Lab 6: 实现 fork 机制

姓名：高铭健

学号：3210102322

日期：2023.1.3

1. 实验目的

- 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

- 计算机（Intel Core i5以上，4GB内存以上）系统
- Ubuntu 22.04.2 LTS

3. 实验原理

3.1 `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。fork 具有以下特点：

- 子 task 和父 task 在不同的内存空间上运行。
- 关于 fork 的返回值：
  - `fork` 成功时，父 task 返回值为子 task 的 pid，子 task 返回值为 0。
  - `fork` 失败时，父 task 返回值为 -1。
- 创建的子 task 需要**深拷贝** `task_struct`，并且调整自己的页表、栈 和 CSR 等信息，同时还需要复制一份在用户态会用到的内存（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。

3.2 `fork` 在 Linux 中的实际应用

Linux 的另一个重要系统调用是 `exec`，它的作用是将进行了该系统调用的 task 换成另一个 task 。这两个系统调用一起，支撑起了 Linux 处理多任务的基础。当我们在 shell 里键入一个程序的目录时，shell（比如 zsh 或 bash）会先进行一次 fork，这时候相当于有两个 shell 正在运行。然后其中的一个 shell 根据 `fork` 的返回值（是否为 0），发现自己和原本的 shell 不同，再调用 `exec` 来把自己给换成另一个程序，这样 shell 外的程序就得以执行了。

4. 实验步骤

4.1 实现 fork()

- 在`trap_handler`中加入`sys_clone`的`syscall`，编号为220，对该调用进行处理：

```
#define SYS_WRITE 64
#define SYS_PID 172
#define SYS_CLONE 220
void trap_handler(unsigned long scause, unsigned long sepc, struct pt_regs *regs) {
    // 通过 `scause` 判断trap类型
    // 如果是interrupt 判断是否是timer interrupt
    // 如果是timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()` 设置下一次时钟中断
    if(scause & 0x8000000000000000){//trap 类型为interrupt
        if(scause == 0x8000000000000005){//timer interrupt
            //printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
            do_timer();
        }
    }
    else if(scause == 8){
        if(regs->reg[17] == SYS_WRITE){
            char *buffer = (char *)(regs->reg[11]);
            int cnt = sys_write(1, buffer, regs->reg[12]);
            regs->sepc += 4;
        }
        else if(regs->reg[17] == SYS_PID){
            uint64 pid = sys_getpid();
            regs->sepc += 4;
            regs->reg[10] = pid;
        }
        else if(regs->reg[17] == SYS_CLONE){
            regs->reg[10] = sys_clone(regs);
            regs->sepc += 4;
        }
    }
    else if(scause == 15 || scause == 12 || scause == 13){
        printk("[S] trap, ");
        printk("scause: %lx, ", scause);
        printk("stval: %lx, ", regs->stval);
        printk("sepc: %lx\n", regs->sepc);
        if(regs->sepc > 0xf00000000){
            while (1);
        }
        do_page_fault(regs);
        return;
    }
    else{
        printk("[S] unhandled trap, ");
```

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

- 在 `_traps` 中的 `jal x1, trap_handler` 后面插入符号`__ret_from_fork`，作为fork出的子进程的返回地址

```
jal trap_handler

   # -----------
       .global __ret_from_fork
   __ret_from_fork:
       ld t0, 288(sp)
       csrw scause, t0
       ld t0, 280(sp)
       csrw stval, t0
       ld t0, 272(sp)
       csrw sscratch, t0
       ld t0, 264(sp)
       csrw sstatus, t0
       ld t0, 256(sp)
       csrw sepc, t0
       ld x0, 0(sp)
       ld x1, 8(sp)
       ......
```

- 实现`sys_clone`函数：
  - 首先为子进程分配一页，将父进程的task页复制到子任务
  - 设置子任务的`thread.ra`为返回地址`__ret_from_fork`；子进程的`thread.sp`根据`(uint64)child_task + PGSIZE`得到页的末尾，然后减去根据父进程`(uint64)(current) + PGSIZE - (uint64)(regs)`计算到的偏移量就可以得到子进程的`pt_regs`的首地址（这样就可以用sp恢复pt_regs）
  - 通在`task`数组中找到下一个一个可用的位置（task[i] = NULL），得到子进程的pid。
  - 前面的`thread.sp`已经得到pt_regs的基地址，用这个基地址加上不同偏移量得到a0, sp, sepc的地址，将对应地址分别设置为0（返回0）；child_task->thread.sp（使恢复时load到的sp依旧是栈顶指针）；regs->sepc + 4（返回后进行下一条指令）
  - 为子进程的用户栈分配一个新页，并将父进程的用户栈内容复制；为子任务的根页表分配一个新页，并将内核的根页表的内容复制到其中。
  - 遍历父任务的`vma`，对每一项的虚拟地址walk pagetable，跑到最后如果地址的最后一位是1，表示已经映射过了，此时对子进程页表进行相同映射，并把那一页的内容复制过来

- 返回子进程PID。

```
unsigned long sys_clone(struct pt_regs *regs) {
    //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
    struct task_struct *child_task = (struct task_struct *)alloc_page();
    memcopy(child_task, current, PGSIZE);
    //将 thread.ra 设置为__ret_from_fork，并正确设置 thread.sp
    child_task->thread.ra = __ret_from_fork;
    child_task->thread.sp = (uint64)child_task + PGSIZE - ((uint64)(current) + PGSIZE - (uint64)(regs));
//    printk("sp = %lx\n",child_task->thread.sp);
    uint64 child_pid;
    for(uint64 i = 0 ; i < NR_TASKS ; i++){
        if(!task[i]){
            child_pid = i;
            task[i] = child_task;
            break;
        }
    }
    child_task->pid = child_pid;
//    printk("child_task->pid = %lx\n", child_task->pid);
    //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
    uint64 *addr_a0 = child_task->thread.sp + 80;
    *(addr_a0) = 0;
    uint64 *addr_sp = child_task->thread.sp + 16;
//    printk("p_sp=%lx  c_sp=%lx\n", regs->reg[2], *(addr_sp));
    *(addr_sp) = child_task->thread.sp;
//    printk("p_sp=%lx  c_sp=%lx\n", regs->reg[2], *(addr_sp));
    uint64 *addr_sepc = child_task->thread.sp + 32*8;
    *(addr_sepc) = regs->sepc + 4;


    //为 child task 申请 user stack，并将 parent task 的 user stack数据复制到其中
    child_task->user_sp = (uint64 *)alloc_page();
    memcopy(child_task->user_sp, current->user_sp, PGSIZE);
    //为 child task 分配一个根页表，并仿照 setup_vm_final 来创建内核空间的映射
    uint64 *new_pgt = (uint64 *)alloc_page();
    memcopy(new_pgt, swapper_pg_dir, PGSIZE);
    child_task->pgd = new_pgt;
    child_task->satp = (csr_read(satp) & 0xfffff00000000000) | (((uint64)new_pgt - PA2VA_OFFSET) >> 12);
//    printk("task->thread.sscratch = %lx\n", child_task->thread.sepc);
    //根据 parent task 的页表和 vma 来分配并拷贝 child task 在用户态会用到的内存
    for(uint64 i = 0 ; i < current->vma_cnt ; i++){
        uint64 offset[3];
        uint64 page[3];
        uint64 va = current->vmas[i].vm_start;
        uint64 va_end = current->vmas[i].vm_end;
        printk("va = %lx\n", va);
        while(va < va_end){
```

```
            offset[2] = (va >> 30) & 0x1FF;
            offset[1] = (va >> 21) & 0x1FF;
            offset[0] = (va >> 12) & 0x1FF;
            if (!(current->pgd[offset[2]])){
                va += PGSIZE;
                continue;
            }
            else{
                uint64 *pgt_next = (uint64 *)(((current->pgd[offset[2]] & 0xfffffffffffffc00) << 2) + PA2VA_OFFSET);
                printk("pgt_next = %lx\n",pgt_next);
                if(!(pgt_next[offset[1]])){
                    va += PGSIZE;
                    continue;
                }
                else{
                    uint64 *pgt_nnext = (uint64 *)(((pgt_next[offset[1]] & 0xfffffffffffffc00) << 2) + PA2VA_OFFSET);
                    printk("pgt_nnext = %lx\n",pgt_nnext[offset[0]]);
                    if(!(pgt_nnext[offset[0]])){
                        va += PGSIZE;
                        continue;
                    }
                    else if(pgt_nnext[offset[0]] & 1){
                        printk("pgt_nnext[offset[0]] = %lx\n", pgt_nnext[offset[0]]);
                        uint64* new_space = (uint64 *)alloc_page();
                        // do mapping
                        uint64 pa = (uint64)new_space - PA2VA_OFFSET;
                        create_mapping(child_task->pgd, va, pa, PGSIZE,31);
                        uint64* src = (uint64 *)(va);
                        memcopy(new_space , src, PGSIZE);
                        printk("in\n");
                    }
                }
            }
            va += PGSIZE;
            printk("va = %lx\n", va);
        }
    }
    //返回子 task 的 pid
    return child_task->pid;
}
```

- 在初始化进程的时候将第一个进程之后的进程都设为NULL（留给子进程分配）

```
for(int i = 1 ; i < NR_TASKS ; i++){
        if(i > 1){
            task[i] = NULL;
            continue;
        }
    ......
```

- 调度的时候遇到为NULL的进程直接跳过

```
void schedule(void) {
    //如果所有运行状态下的线程运行剩余时间都为0，则对 task[1] ~ task[NR_TASKS-1] 的运行剩余时间重新赋值 （使用 rand()） ，之后再重新进行调度。
    //遍历线程指针数组task（不包括 idle ，即 task[0] ）， 在所有运行状态（TASK_RUNNING） 下的线程运行剩余时间最少的线程作为下一个执行的线程
#ifdef SJF
    ......
#else
    int judge_all_0 = 1;
    int select_task = 0;
    for( ; ; ){
        for(int i = NR_TASKS-1 ; i >=1  ; i--){
            if(!task[i]){
                continue;
            }
    .......
```

4.2 编译及测试

- 第一段main函数：

```
switch to [PID = 1 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003fffffffff8, sepc: 0000000000010158
[S] trap, scause: 000000000000000d, stval: 0000000000011978, sepc: 00000000000101f0
[U-PARENT] pid: 1 is running!, global_variable: 0
[U-PARENT] pid: 1 is running!, global_variable: 1
[U-PARENT] pid: 1 is running!, global_variable: 2
[U-PARENT] pid: 1 is running!, global_variable: 3

switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000d, stval: 0000000000011978, sepc: 000000000001018c
[U-CHILD] pid: 2 is running!, global_variable: 0
[U-CHILD] pid: 2 is running!, global_variable: 1
[U-CHILD] pid: 2 is running!, global_variable: 2
[U-CHILD] pid: 2 is running!, global_variable: 3
```

首先输出父进程，然后调度的子进程，此时只会触发一个scause= d的异常，然后输出子进程

```
switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[U-CHILD] pid: 2 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 5
[U-CHILD] pid: 2 is running!, global_variable: 6
[U-CHILD] pid: 2 is running!, global_variable: 7

switch to [PID = 1 COUNTER = 3 PRIORITY = 3]
[U-PARENT] pid: 1 is running!, global_variable: 4
[U-PARENT] pid: 1 is running!, global_variable: 5
[U-PARENT] pid: 1 is running!, global_variable: 6
[U-PARENT] pid: 1 is running!, global_variable: 7
```

然后继续进行调度

- 第二段main函数：

```
...proc_init done!
2023 Hello RISC-V

switch to [PID = 1 COUNTER = 3 PRIORITY = 3]
[S] trap, scause: 000000000000000c, stval: 00000000000100e8, sepc: 00000000000100e8
[S] trap, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0000000000010158
[S] trap, 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
[U-PARENT] pid: 1 is running!, global_variable: 3
[U-PARENT] pid: 1 is running!, global_variable: 4
[U-PARENT] pid: 1 is running!, global_variable: 5
[U-PARENT] pid: 1 is running!, global_variable: 6
[U-PARENT] pid: 1 is running!, global_variable: 7
[U-PARENT] pid: 1 is running!, global_variable: 8
[U-PARENT] pid: 1 is running!, global_variable: 9
[U-PARENT] pid: 1 is running!, global_variable: 10

switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[U-CHILD] pid: 2 is running!, global_variable: 3
[U-CHILD] pid: 2 is running!, global_variable: 4
[U-CHILD] pid: 2 is running!, global_variable: 5
```

fork前先输出当前进程，之后输出父进程，最后调度子进程进行输出（fork之前的进程已经运行到2，所以子进程从3开始运行）

- 第三段main函数：

```
[S] trap, scause: 000000000000000f, stval: 0000003ffffffff8, sepc: 0000000000010158
[S] trap, scause: 000000000000000d, stval: 0000000000011930, sepc: 0000000000010174
[U] pid: 1 is running!, global_variable: 0
[U] pid: 1 is running!, global_variable: 1
[U] pid: 1 is running!, global_variable: 2
[U] pid: 1 is running!, global_variable: 3
[U] pid: 1 is running!, global_variable: 4
[U] pid: 1 is running!, global_variable: 5
[U] pid: 1 is running!, global_variable: 6
[U] pid: 1 is running!, global_variable: 7
[U] pid: 1 is running!, global_variable: 8
[U] pid: 1 is running!, global_variable: 9

switch to [PID = 3 COUNTER = 3 PRIORITY = 3]
[U] pid: 3 is running!, global_variable: 2
[U] pid: 3 is running!, global_variable: 3
[U] pid: 3 is running!, global_variable: 4
[U] pid: 3 is running!, global_variable: 5
[U] pid: 3 is running!, global_variable: 6
[U] pid: 3 is running!, global_variable: 7
[U] pid: 4 is running!, global_variable: 6
[U] pid: 4 is running!, global_variable: 7
[U] pid: 4 is running!, global_variable: 8
[U] pid: 4 is running!, global_variable: 9
```

```
switch to [PID = 2 COUNTER = 3 PRIORITY = 3]
[U] pid: 2 is running!, global_variable: 1
[U] pid: 2 is running!, global_variable: 2
[U] pid: 2 is running!, global_variable: 3
[U] pid: 2 is running!, global_variable: 4
[U] pid: 2 is running!, global_variable: 5
[U] pid: 2 is running!, global_variable: 6
[U] pid: 2 is running!, global_variable: 7
[U] pid: 2 is running!, global_variable: 8
[U] pid: 2 is running!, global_variable: 9

switch to [PID = 4 COUNTER = 3 PRIORITY = 3]
[U] pid: 4 is running!, global_variable: 2
[U] pid: 4 is running!, global_variable: 3
[U] pid: 4 is running!, global_variable: 4
[U] pid: 4 is running!, global_variable: 5
[U] pid: 4 is running!, global_variable: 6
[U] pid: 4 is running!, global_variable: 7
[U] pid: 4 is running!, global_variable: 8
[U] pid: 4 is running!, global_variable: 9
```

进行两次fork后在4个进程间进行调度

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?

   答：复制了task_struct的所有内容，如下图：

```
struct task_struct {
    struct thread_info *thread_info;
    uint64 state;    // 线程状态
    uint64 counter;  // 运行剩余时间
    uint64 priority; // 运行优先级 1最低 10最高
    uint64 pid;      // 线程id
    struct thread_struct thread;
    uint64 satp;
    uint64 kernel_sp;
    uint64 *user_sp;
    pagetable_t pgd;
    uint64 vma_cnt;                     // 下面这个数组里的元素的数量
    struct vm_area_struct vmas[0];          // 为什么可以开大小为 0 的数组?
};
```

   还有task 的内核态栈、进入内核态时的寄存器pt_regs、上一次发生调度调用 `switch_to` 时 `thread_struct` 的信息

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。

   答：应该设成pt_regs的基地址（x0的地址），这样到 `__ret_from_fork` 就可以直接恢复各寄存器

3. 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：因为返回到 `__ret_from_fork` 会恢复 pt_regs的各寄存器，包括 pt_regs中的sp，如果不设置的话运行 `ld x2, 16(sp)` 之后会ld一个错误的值导致后面的寄存器没办法恢复