浙江大学

本科实验报告

课程名称：	操作系统
姓　　名：	夏尤楷
学　　院：	计算机科学与技术学院
系：	计算机科学与技术系
专　　业：	计算机科学与技术
学　　号：	3210104331
指导教师：	夏莹杰

2024 年 1 月 14 日

浙江大学操作系统实验报告

实验名称：fork 机制

电子邮件地址：459510812@qq.com 手机：15058004449

实验地点：玉泉曹光彪西楼 503 实验日期：2024 年 1 月 7 日

一、实验目的和要求

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

二、实验过程

（一）准备工作

修改 task_init 函数中修改为仅初始化一个 task，之后的其他 task 的指针全部设为 NULL，其余的 task 均通过 fork 创建。

```
1. void task_init() {
2.     // 1. 调用 kalloc() 为 idle 分配一个物理页
3.     // 2. 设置 state 为 TASK_RUNNING;
4.     // 3. 由于 idle 不参与调度 可以将其 counter / priority 设置为 0
5.     // 4. 设置 idle 的 pid 为 0
6.     // 5. 将 current 和 task[0] 指向 idle
7.     idle = (struct task_struct*)kalloc();
8.     idle->state = TASK_RUNNING;
9.     idle->counter = 0;
10.    idle->priority = 0;
11.    idle->pid = 0;
12.    current = idle;
13.    task[0] = idle;
14.
15.    //只初始化一个用户态进程
16.    task[1] = (struct task_struct*)kalloc();
17.    task[1]->pid = 1;
18.    task[1]->state = TASK_RUNNING;
19.    task[1]->counter = 0;
20.    task[1]->priority = rand();
21.    task[1]->thread.ra = (uint64)&__dummy;
22.    task[1]->thread.sp = (uint64)task[1] + PGSIZE;
23.    //创建进程自己的页表并拷贝
```

```
    task[1]->pgd = (pagetable_t)alloc_page();
    for (int j = 0; j < 512; ++j)
        task[1]->pgd[j] = swapper_pg_dir[j];
    task[1]->vma_cnt = 0;
    do_mmap(task[1], USER_END - PGSIZE, PGSIZE,
         VM_R_MASK | VM_W_MASK | VM_ANONYM, 0, 0);
    task[1]->thread.sepc = load_program(task[1]);
    printk("[S] Initialized: pid: 1, priority: 1, counter: 0\n");

    for (int i = 2; i < NR_TASKS; ++i)
        task[i] = NULL;

    return;
}
```

（二） 系统调用 sys_clone

在 trap_handler 处理中断的逻辑中，加入系统调用 sys_clone：

```
else if (syscall_id == SYS_CLONE)
{
    ret = sys_clone(regs);
}
```

（三） __ret_from_fork

在_traps 中的 jal trap_handler 后面插入一个符号__ret_from_fork。

（四） sys_clone 的实现

这个系统调用函数被用来创建一个子进程。在其中实现以下几步：

1. 为子进程分配一个空闲的 pid。
2. 创建一个新的子进程 task_struct，把父进程的 task_struct 的所有内容拷贝进去。将子进程的 thread.ra 改为__ret_from_fork，以使得子进程第一次被调度到时返回该符号所在的位置，通过其下的逻辑恢复子进程的寄存器，从而实现到子进程的调度。
3. 计算出 child task 的对应的 pt_regs 的地址，设置其中的值，包括把 x10（a0）设置为 0（子进程 fork 的返回值）、sp（x2）设置成 pt_regs 的地址（子进程栈顶地址）、sepc 加 4（父进程在某一步后进行 fork，子进程直接在父进程的这一步之后继续执行。
4. 为子进程分配一个根页表，将 swapper_pg_dir 的所有内容复制进去，完成子进程的内核空间的映射。
5. 根据父进程的页表和 vma 来分配并拷贝子进程在用户态会用到的内存。

6. 返回子进程的 pid。

代码如下：

```
uint64_t sys_clone(struct pt_regs *regs) {
    // 先申请一个pid, 看看是否成功
    int pid = 0;
    while (pid < NR_TASKS)
        if (task[pid] == NULL)
            break;
        else
            ++pid;
    if (pid == NR_TASKS)
        return -1;

    // 1. 创建一个新的task
    struct task_struct* child_task;

    child_task = (struct task_struct*)kalloc();
    task[pid] = child_task;
    for (int i = 0; i < 1 << 12; ++i)
        ((char *)child_task)[i] = ((char *)current)[i];
    child_task->pid = pid;
    child_task->thread.ra = (uint64)__ret_from_fork;

    // 2. 计算出 child task 的对应的 pt_regs 的地址, 设置其中的值。
    uint64_t offset = (uint64_t)regs - (uint64_t)current;
    struct pt_regs *child_regs =
       (struct pt_regs *)((uint64_t)child_task + offset);
    child_regs->x[10] = 0;
    child_regs->x[2] = (uint64_t)child_regs;
    child_regs->sepc = regs->sepc + 4;

    child_task->thread.sp = (uint64_t)child_regs;

    // 3. 为 child task 分配一个根页表,
     并将 swapper_pg_dir 中的一级内核页表项复制进去
    child_task->pgd = (pagetable_t)alloc_page();
    memset(child_task->pgd, 0, 1 << 12);
    for (int j = 0; j < 512; ++j)
        child_task->pgd[j] = swapper_pg_dir[j];

    // 4. 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 (int i = 0; i < current->vma_cnt; ++i) {
        struct vm_area_struct vma = current->vmas[i];
        for (uint64_t vaddr = PGROUNDDOWN(vma.vm_start);
            vaddr < vma.vm_end; vaddr += PGSIZE) {
            if ((current->pgd[(vaddr >> 30) & 0x1ff] & (PTE_V)) == PTE_V) {
                uint64_t child_page = alloc_page();
                for (int j = 0; j < PGSIZE; ++j)
                    ((char *)child_page)[j] = ((char *)vaddr)[j];
                create_mapping(child_task->pgd, vaddr, child_page - PA2VA_OFFSET,
                    PGSIZE, vma.vm_flags | PTE_U | PTE_V);
            }
        }
    }
    printk("[S] New task: %d\n", pid);

```

```
52.     // 5. 返回子 task 的 pid
53.     return child_task->pid;
54. }
```

（五） 结果演示

使用 getpid.c 中的第一个 main 函数，输出结果如下：

```
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
[S] Initialized: pid: 1, priority: 1, counter: 0
[S-MODE] 2022 Hello RISC-V
[S] Value of sstatus is 8000000000006000
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 4]

[S] switch to [PID = 1 PRIORITY = 1 COUNTER = 4]
[S] Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
[S] SD/SMO page fault.
sepc: 0000000000010124, scause: 000000000000000f, stval: 0000003fffffff8.
[PID = 1] is running, variable: 0
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 10]

[S] switch to [PID = 1 PRIORITY = 1 COUNTER = 10]
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[PID = 1] is running, variable: 1
[S] Supervisor mode time interrupt!
```

使用 getpid.c 中的第二个 main 函数，输出结果如下：

```
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
[S] Initialized: pid: 1, priority: 1, counter: 0
[S-MODE] 2022 Hello RISC-V
[S] Value of sstatus is 8000000000006000
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 4]

[S] switch to [PID = 1 PRIORITY = 1 COUNTER = 4]
[S] Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
[S] SD/SMO page fault.
sepc: 0000000000010158, scause: 000000000000000f, stval: 0000003fffffff8.
[S] New task: 2
[U-PARENT] pid: 1 is running!, global_variable: 0
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 1
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 2
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 3
[S] Supervisor mode time interrupt!

[S] switch to [PID = 2 PRIORITY = 1 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 0
[U-CHILD] pid: 2 is running!, global_variable: 1
[S] Supervisor mode time interrupt!
[S] Supervisor mode time interrupt!
[U-CHILD] pid: 2 is running!, global_variable: 2
[S] Supervisor mode time interrupt!
[U-CHILD] pid: 2 is running!, global_variable: 3
[S] Supervisor mode time interrupt!
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 10]
[S] SET [PID = 2 PRIORITY = 1 COUNTER = 4]

[S] switch to [PID = 2 PRIORITY = 1 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 4
[S] Supervisor mode time interrupt!
```

使用 getpid.c 中的第三个 main 函数，输出结果如下：

```
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
[S] Initialized: pid: 1, priority: 1, counter: 0
[S-MODE] 2022 Hello RISC-V
[S] Value of sstatus is 8000000000006000
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 4]

[S] switch to [PID = 1 PRIORITY = 1 COUNTER = 4]
[S] Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
[S] SD/SMO page fault.
sepc: 0000000000010158, scause: 000000000000000f, stval: 0000003ffffffff8.
[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] New task: 2
[U-PARENT] pid: 1 is running!, global_variable: 3
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 4
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 5
[S] Supervisor mode time interrupt!
[U-PARENT] pid: 1 is running!, global_variable: 6
[S] Supervisor mode time interrupt!

[S] switch to [PID = 2 PRIORITY = 1 COUNTER = 4]
[U-CHILD] pid: 2 is running!, global_variable: 3
[S] Supervisor mode time interrupt!
[U-CHILD] pid: 2 is running!, global_variable: 4
[S] Supervisor mode time interrupt!
[U-CHILD] pid: 2 is running!, global_variable: 5
[S] Supervisor mode time interrupt!
[U-CHILD] pid: 2 is running!, global_variable: 6
[S] Supervisor mode time interrupt!
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 10]
[S] SET [PID = 2 PRIORITY = 1 COUNTER = 4]
```

使用 getpid.c 中的第四个 main 函数，输出结果如下：

```
Boot HART MEDELEG         : 0x000000000000b109
...buddy_init done!
[S] Initialized: pid: 1, priority: 1, counter: 0
[S-MODE] 2022 Hello RISC-V
[S] Value of sstatus is 8000000000006000
[S] SET [PID = 1 PRIORITY = 1 COUNTER = 4]

[S] switch to [PID = 1 PRIORITY = 1 COUNTER = 4]
[S] Instruction page fault.
sepc: 00000000000100e8, scause: 000000000000000c, stval: 00000000000100e8.
[S] SD/SMO page fault.
sepc: 0000000000010158, scause: 000000000000000f, stval: 0000003ffffffff8.
[U] pid: 1 is running!, global_variable: 0
[S] New task: 2
[U] pid: 1 is running!, global_variable: 1
[S] New task: 3
[U] pid: 1 is running!, global_variable: 2
[S] Supervisor mode time interrupt!
[U] pid: 1 is running!, global_variable: 3
[S] Supervisor mode time interrupt!
[U] pid: 1 is running!, global_variable: 4
[S] Supervisor mode time interrupt!
[U] pid: 1 is running!, global_variable: 5
[S] Supervisor mode time interrupt!

[S] switch to [PID = 2 PRIORITY = 1 COUNTER = 4]
[U] pid: 2 is running!, global_variable: 1
[S] New task: 4
[U] pid: 2 is running!, global_variable: 2
[S] Supervisor mode time interrupt!
[U] pid: 2 is running!, global_variable: 3
[S] Supervisor mode time interrupt!
[U] pid: 2 is running!, global_variable: 4
[S] Supervisor mode time interrupt!
[U] pid: 2 is running!, global_variable: 5
[S] Supervisor mode time interrupt!

[S] switch to [PID = 3 PRIORITY = 1 COUNTER = 4]
[U] pid: 3 is running!, global_variable: 2
[S] Supervisor mode time interrupt!
[U] pid: 3 is running!, global_variable: 3
```

三、讨论和心得

这次实验让我对 fork 机制有了更深入和全面的了解。尤其是在 fork()函数的返回值方面，父进程返回值是子进程的 pid，而子进程的返回值是 0，我以前一直都无法理解同一个函数在不同进程中是如何做到返回值不同的。通过这次实验，我明白了，原来是在子进程恢复用来存储返回值的寄存器 a0 之前，把用来恢复 a0 寄存器的数值偷换成 0，从而改变了子进程的返回值。

不过，fork()对应的理论课内容早早已经讲完，而其在实验中的实现要等到学期快结束时的最后一个实验，对于这种脱节总感觉有些遗憾。

四、思考题

1. 复制了进程的线程信息、线程状态、运行剩余时间、优先级、线程 id、线程调度相关寄存器、一级虚拟页表地址（页表内容未复制）、vma 的数量和各 vma 的内容。
2. 应该设置成子进程的 pt_regs 指针值。这个就是子进程当前的栈顶地址，由于子进程又开了一个栈，它的栈顶地址就并非从父进程复制来的被保存下来的 sp（父进程栈顶）的值了，所以另外设置成子进程的 pt_regs 指针值，以使子进程栈与父进程栈区分开来。
3. 因为由于子进程直接复制了父进程栈的内容，导致子进程在被调度到并跳出 fork 时用来恢复各寄存器的值就是父进程在调用 fork 时压入栈的各寄存器的值。由于子进程通过__ret_from_fork 借用_traps 中恢复寄存器的逻辑来跳出，其中包含恢复 sp 寄存器的过程。如果不修改，恢复后的 sp 会是父进程的栈顶而非子进程自身的栈顶，使得父子进程依然互相干涉。

五、附录

无。