操作系统研讨课 实验报告

代瀚堃 2019K8009929051

一、实验中遇到的问题

1. 不清楚操作系统处理例外的整个流程

这部分的任务书和 FAQ 我看了很多遍以后还是不懂，所以决定先从代码入手，才梳理清楚了各个函数的调用关系，并在试探-调试的过程中理解了操作系统处理例外的整个机制。

2. 不清楚屏幕驱动中各个函数的作用

由于一开始所有程序都运行在内核态，移动光标使用的是 vt100_move_cursor 函数，所以在将用户态和内核态隔离开后，我仍将 vt100_move_cursor 封装为 sys_move_cursor 系统调用，而 printf 在内核中的对应函数为 screen_write。导致用户态程序在使用 printf 时写缓冲区溢出，将内核中锁的数目 lock_num 覆盖，所以后来访问锁数组时访问到了非法区域。事实上，vt100_move_cursor 是供内核态使用的，它调用了 printk 函数，可以直接写串口；而真正应该使用的 screen_move_cursor 与 screen_write 函数一起，先写缓冲区，再在某个特定的时候通过 screen_reflush 刷新到屏幕上，刷新的过程中才会写串口。由于 vt100_move_cursor 并不会更改 screen_cursor_x 和 screen_cursor_y 的值，所以原来的写法中，screen_cursor_x 和 screen_cursor_y 的值并不如我们的预期，所以很有可能会越界。

3. fork 时栈指针设置错误

fork_task 中的父进程调用 fork 后创建一个子进程，然而当父进程通过 sys_yield 退出后，计数变量的值就不再改变，而子进程则能够一直增加。一个可能的原因是我创建子进程的过程中，对 ready_queue 的操作不正确，于是我仔细检查了 ready_queue 中各个节点的各个域，发现并没有问题，父进程确实是可以被调度到的。由于任务书的最后强调了 sys_fork 可能会影响子/父进程的 sp 和 fp 寄存器，于是我查看反汇编出来的结果：

```
0000000050203c90 <fork_task>:
    50203c90:   1101                addi    sp,sp,-32
    50203c92:   ec06                sd  ra,24(sp)
    50203c94:   e822                sd  s0,16(sp)
    50203c96:   1000                addi    s0,sp,32
    50203c98:   479d                li  a5,7
    50203c9a:   fef42223            sw  a5,-28(s0)
    50203c9e:   57fd                li  a5,-1
    50203ca0:   fef42423            sw  a5,-24(s0)
    50203ca4:   fe042623            sw  zero,-20(s0)
    50203ca8:   fe842783            lw  a5,-24(s0)
    50203cac:   2781                sext.w  a5,a5
    50203cae:   c3a9                beqz    a5,50203cf0 <fork_task+0x60>
    50203cb0:   fe442783            lw  a5,-28(s0)
    50203cb4:   85be                mv  a1,a5
    50203cb6:   4505                li  a0,1
```

```
    50203cb8:   db3ff0ef            jal ra,50203a6a <sys_move_cursor>
```

对应的 C 代码如下：

```
void fork_task(void){
    int i;
    int print_location = 7;
    int pid=-1;
    for(i=0;;i++){
        if(pid!=0){
            sys_move_cursor(1, print_location);
            printf("> [TASK] This is parent task(%d).", i);
            sys_move_cursor(1, print_location+1);
            printf("Please enter a priority and create a child task: ");
            sys_yield();
        }else{
            sys_move_cursor(1, print_location+2);
            printf("> [TASK] This is child task(%d).",i);
            sys_yield();
        }
        char c=sys_getchar();
        if(c>=0&&c<=127){//c should be 0 ~ 9
            sys_move_cursor(51, print_location+1);
            printf("%c",c);
            pid=sys_fork();
            if(pid==0){
                sys_prior(c-'0');
            }
        }
    }
}
```

也就是说，编译器为 fork_task 函数开了一个 32 字节的栈帧，存放局部变量 i, print_location 和 pid，返回地址和 s0 的旧值。而对栈中变量的寻址是通过 fp (s0)指针来实现的，所以我们在系统函数 do_fork 中只修改上下文中的 sp 还不够，如果子进程的上下文中保存的 fp 仍然是父进程的 fp，最后子进程在运行时仍然会在父进程的空间内取数。我们看到子进程的计数变量的值大约是其他任务的 2 倍，也验证了上述想法。

二、还有待解决的问题

1. 在解决上面的第 3 个问题时，我们在 do_fork 中不仅要修改子进程上下文中 sp 的值，还需要修改 fp 的值为栈帧高地址（基地址），但用户并不一定总是通过 fp 来对栈帧索引，fp (s0)中可能存放其他值，所以将上下文中的 fp 修改为用户栈的基地址具有一定的局限性，另一方面，用户还有可能在栈中存放一些与栈指针有关的数据（比如 fp 的旧值），这在我们复制堆栈数据时会被迁移到子进程的栈，并且我们无法识别出这些数据是否真的和父进程的栈有关。这些问题在目前看来还没有很好的解决方案。