Lab5 实验

练习 0：

1、实验五在用 meld 进行合并的时候一定要小心，因为前面四个实验的部分函数不能直接复制到实验五中，还要进行一些改进，合并的时候需要保留实验五的代码注释。具体来看有这么以下三个文件需要额外修改：trap.c、vmm.c 和 proc.c，其中 vmm.c 的修改是 challenge 的内容。

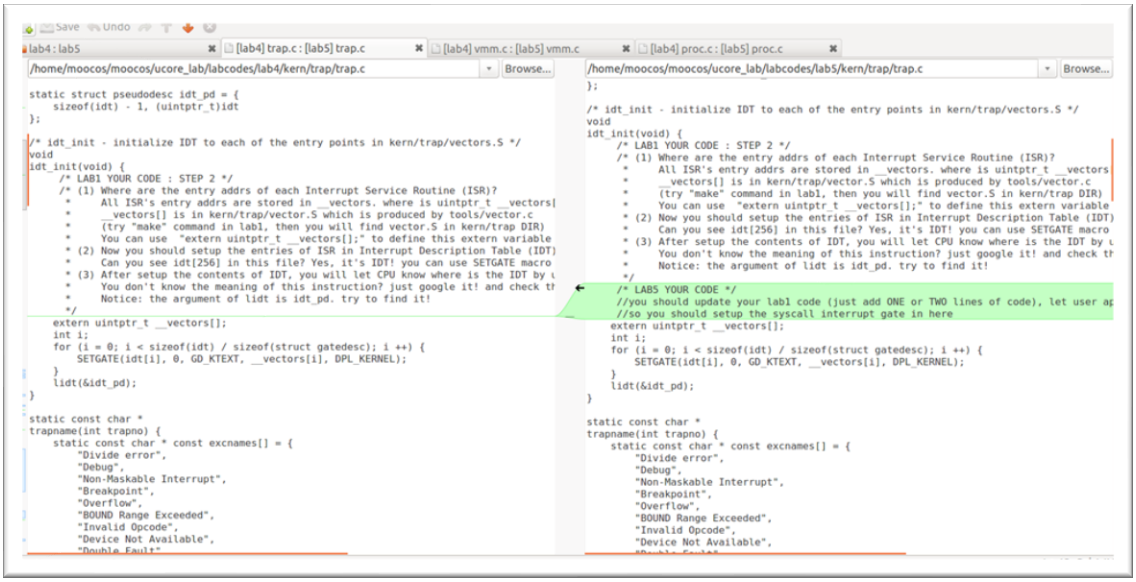

图 1　meld 合并过程中需要额外修改的文件

2、trap.c 的修改主要是两个地方，一个是在初始化中断符描述符表时设置一个特定中断号的中断门，专门用于用户进程访问系统调用；第二个是每 100 个时钟周期，将当前占用 CPU 的进程的 need_resched 设置为 1 。

3 、proc.c 主要的修改也是两个地方，一个是 alloc_proc 函数，需要增加初始化的工作，相对来说比较简单；另一个是 do_fork 函数，在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程）。

```
    if((proc = alloc_proc()) == NULL)
    {
        goto fork_out;
    }
    else
    {
        assert(current->wait_state == 0);//make sure current process's wait_state is 0
        proc->parent = current;
    }
    if(setup_kstack(proc) != 0)
    {
        goto bad_fork_cleanup_proc;
    }
    if(copy_mm(clone_flags,proc) != 0)
    {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc,stack,tf);
    proc->pid = get_pid();
    hash_proc(proc);
    set_links(proc);//insert proc_struct into proc_list, set the relation links of process
    wakeup_proc(proc);
    ret = proc->pid;
    //nr_process++;
fork_out:
    return ret;
```

图 2　do_fork 函数

练习 1：

1、练习 1 让我们补充用 load_icode 函数的内容，该函数的主要工作就是给用户进程建立一个能够让用户进程正常运行的用户环境。我们需要补充的是最后一步，即重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让 CPU 转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。这道题的难度在于最后一个赋值操作，即对 tf_eflags 进行赋值，使得系统能够产生中断，在这之前都比较简单，于是有下面的第一版代码。

```
//(6) setup trapframe for user environment
struct trapframe *tf = current->tf;
memset(tf, 0, sizeof(struct trapframe));
/* LAB5:EXERCISE1 YOUR CODE
 * should set tf_cs,tf_ds,tf_es,tf_ss,tf_esp,tf_eip,tf_eflags
 * NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel. Sc
 *          tf_cs should be USER_CS segment (see memlayout.h)
 *          tf_ds=tf_es=tf_ss should be USER_DS segment
 *          tf_esp should be the top addr of user stack (USTACKTOP)
 *          tf_eip should be the entry point of this binary program (elf->e_entry)
 *          tf_eflags should be set to enable computer to produce Interrupt
 */
tf->tf_cs = USER_CS;
tf->tf_ds = tf->tf_es = tf->tf_ss = USER_DS;
tf->tf_esp = USTACKTOP;
tf->tf_eip = elf->e_entry;
tf->tf_eflags =
ret = 0;
```

图 3　练习 1 第一版代码

下面开始寻找 tf_eflags 的正确赋值。这需要对所有的文件比较熟悉，知道 tf_eflags 的取值集合在哪个文件里。很幸运的是，我之前做实验的时候找过相关内容，该寄存器的取值集合在 mmu.h 文件里，宏定义和注释如下图。

```
/* Eflags register */
#define FL_CF           0x00000001  // Carry Flag
#define FL_PF           0x00000004  // Parity Flag
#define FL_AF           0x00000010  // Auxiliary carry Flag
#define FL_ZF           0x00000040  // Zero Flag
#define FL_SF           0x00000080  // Sign Flag
#define FL_TF           0x00000100  // Trap Flag
#define FL_IF           0x00000200  // Interrupt Flag
#define FL_DF           0x00000400  // Direction Flag
#define FL_OF           0x00000800  // Overflow Flag
#define FL_IOPL_MASK    0x00003000  // I/O Privilege Level bitmask
#define FL_IOPL_0       0x00000000  //   IOPL == 0
#define FL_IOPL_1       0x00001000  //   IOPL == 1
#define FL_IOPL_2       0x00002000  //   IOPL == 2
#define FL_IOPL_3       0x00003000  //   IOPL == 3
#define FL_NT           0x00004000  // Nested Task
#define FL_RF           0x00010000  // Resume Flag
#define FL_VM           0x00020000  // Virtual 8086 mode
#define FL_AC           0x00040000  // Alignment Check
#define FL_VIF          0x00080000  // Virtual Interrupt Flag
#define FL_VIP          0x00100000  // Virtual Interrupt Pending
#define FL_ID           0x00200000  // ID flag
```

图 4　eflags 取值集合

通过注释，我们就能发现，中断的 Flag 是 FL_IF，另外一个很有意思的事情是 FL_VIP，其注释显示的是 Virtual Interrupt Flag，中文名称是虚拟中断标志，但是我不明白是干什么的，所以我问老师了……然后，代码就搞定了。

```
/* Eflags register */
#define FL_CF           0x00000001  // Carry Flag
#define FL_PF           0x00000004  // Parity Flag
#define FL_AF           0x00000010  // Auxiliary carry Flag
#define FL_ZF           0x00000040  // Zero Flag
#define FL_SF           0x00000080  // Sign Flag
#define FL_TF           0x00000100  // Trap Flag
#define FL_IF           0x00000200  // Interrupt Flag
#define FL_DF           0x00000400  // Direction Flag
#define FL_OF           0x00000800  // Overflow Flag
#define FL_IOPL_MASK    0x00003000  // I/O Privilege Level bitmask
#define FL_IOPL_0       0x00000000  //   IOPL == 0
#define FL_IOPL_1       0x00001000  //   IOPL == 1
#define FL_IOPL_2       0x00002000  //   IOPL == 2
#define FL_IOPL_3       0x00003000  //   IOPL == 3
#define FL_NT           0x00004000  // Nested Task
#define FL_RF           0x00010000  // Resume Flag
#define FL_VM           0x00020000  // Virtual 8086 mode
#define FL_AC           0x00040000  // Alignment Check
#define FL_VIF          0x00080000  // Virtual Interrupt Flag
#define FL_VIP          0x00100000  // Virtual Interrupt Pending
#define FL_ID           0x00200000  // ID flag
```

图 5　eflags 取值集合截图

2、do_execv 函数来完成用户进程的创建工作，此函数首先为加载新的执行码做好用户态内存空间清空准备，然后通过调用 load_icode 函数加载应用程序执行码到当前进程的新创建的用户态虚拟空间中，建立用户进程正常运行的用户环境。最后 initproc 按着产生系统调用的函数调用路径原路返回，执行中断返回指令“iret”，将切换到用户进程的第一条指令处开始执行。

练习 2：

1、copy_range 函数的作用是 copy content of memory (start, end) of one process

A to another process B，在实现过程中是在没什么好说的。

```
         * (1) find src_kvaddr: the kernel virtual address of page
         * (2) find dst_kvaddr: the kernel virtual address of npage
         * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
         * (4) build the map of phy addr of  nage with the linear addr start
         */
        memcpy(page2kva(npage),page2kva(page),PGSIZE);
        ret=page_insert(to,npage,start,perm);
        assert(ret == 0);
        }
        start += PGSIZE;
    } while (start != 0 && start < end);
    return 0;
}
```

图 6　copy_range 函数

2、Copy on Write 机制原理：执行 fork 系统调用进行复制的时候，父进程（A）不会将整个内存中的内容复制给子进程（B），而是暂时共享相同的物理内存页，并将其设置为只读；当其中的一个进程比如 A 需要修改某个物理内存页时，再重新分配一个新的物理页，将共享的物理内存页复制到新的物理内存页上，这样两个进程都有各自的物理内存页了，且一个进程所做的修改不会被另外一个进程可见。

3、Copy on Write 机制实现：根据上面的分析，主要需要修改两部分。一个是在 do_fork 函数进行复制的时候，不进行实际的复制，而是只将父进程和子进程映射到同一个物理内存空间上，然后将这片物理内存空间设置成只读和共享。当其中一个进程试图写某个共享页面时，就会产生页访问异常，从而进行缺页处理。第二个是在 Page Fault 时，如果发现有进程试图写只读页面且这个页面是共享的话，则分配一个新的物理页给该进程，将当前物理页复制过去；同时检查原来的物理页是否仍然是共享，如若不是则删掉共享标志，同时恢复写标志。

练习 3：

1、正常的系统调用都需要先执行正常的中断处理流程，将控制权交给操作系统，然后操作系统来执行相应的函数（后面我所提到的都是函数）。

依次分析，do_fork 函数的作用是创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同，即父进程创建了一个子进程，将子进程的状态设置为 RUNNALB；

do_execve 函数的作用练习 1 已经说明过，该函数不会影响当前进程的执行状态；

do_wait 函数的作用是遍历 proc_list，根据输入参数寻找指定 pid 或任意 pid 的子进程，如果没找到，直接返回错误信息。如果找到子进程，状态为 ZOMBIE，则释放子进程占用的资源并返回，如果状态不为 ZOMBIE，则将当前进程的状态设置为 SLEEPING，然后调用 schedule 函数，从而进入等待状态；

do_exit 函数的作用是释放页表项记录的物理内存以及内存管理结构的占用内存，将当前进程的状态设置为 ZOMBIE，然后唤醒父进程，并调用 schedule 函数，等待父进程回收剩下的资源，最终彻底结束子进程。

2、唯一需要注意的是父进程退出时需检查所有子进程是否退出，如果是，可以直接退出，否则将该进程状态设置为 SLEEPING，等待子进程退出的系统调用。

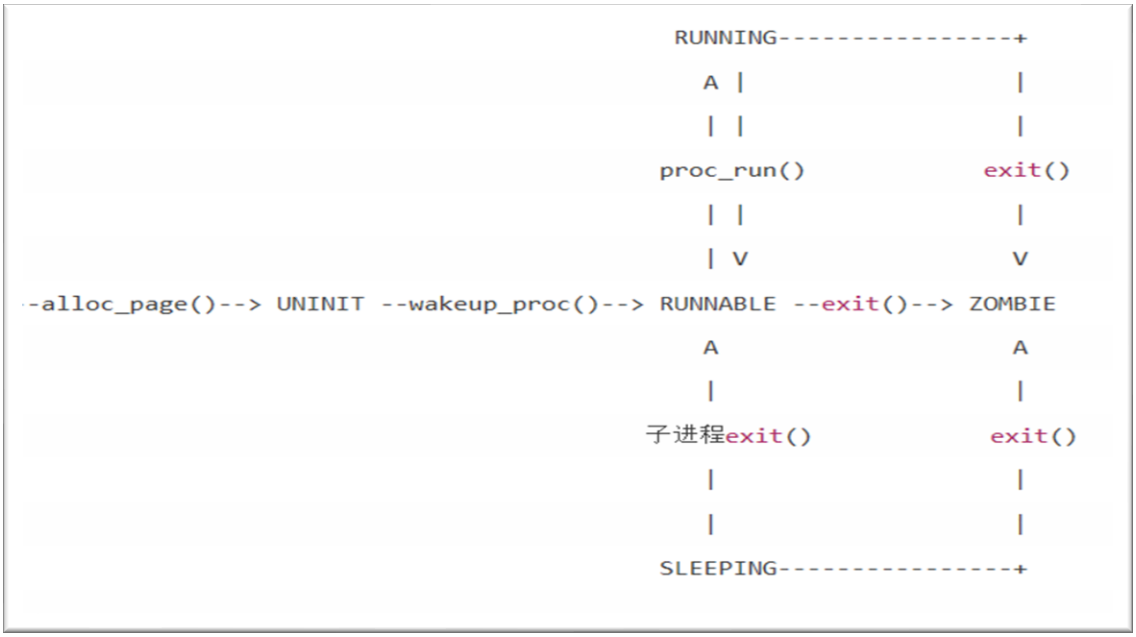

图 7　一个用户态进程的执行状态生命周期图