

实验7：fork，exit，wait系统调用的实现

姓名：王永锋

学号：16337237

实验7：fork，exit，wait系统调用的实现

- 1 实验目的
- 2 本实验完成的功能
- 3 基本原理
 - 3.1 fork的实现
 - 3.2 wait 的实现
 - 3.3 exit的实现
- 4 遇到的问题
 - 4.1 一个有关子进程地址空间与父进程地址空间不一致的问题。
 - 4.2 一个小问题
- 5 测试过程
- 6 实验感想

1 实验目的

- 1. 完善进程控制模型
- 2. 实现一个5状态进程模型

2 本实验完成的功能

通过这一个实验，我为自己实现的操作系统中增加了三个系统调用。

- 1. fork
- 2. exit
- 3. wait

这三个系统调用，建立在原先的进程间通信的机制上，通过向“mm”进程发送消息来实现。具体地说，就fork系统调用而言，父进程在调用fork系统调用后，操作系统能够找到一块空闲的进程控制块，将父进程的信息复制到子进程中，并且向mm进程发送消息以分配子进程的内核空间。而其他的系统调用也都类似。

实现这几个系统调用后，我编写了一个测试程序，来测试这几个系统调用的正确性，验证了自己实现的五状态进程模型。

3 基本原理

3.1 fork的实现

- 1. 主要工作：向mm进程发送一个 `fork` 消息，主要工作由mm进程完成
- 2. mm进程收到消息，判断是fork后，调用`do_fork函数
 - 1. 寻找到一个空闲的进程控制块（通过遍历PCB数组，寻找空闲的块就可以实现），确定子进程pid
 - 2. 从消息中，确定父进程pid。
 - 3. 复制发送消息者的pcb到子进程中。

4. 对子进程复制过来的PCB做出适当的调整

1. 分配新的栈空间给子进程（通过一个很简单的内存分配程序）。
 2. 将父进程的栈空间复制到子进程中。
 3. 修改子进程pid（刚刚复制PCB的时候覆盖了原有的pid）。
 4. 设置父进程pid号。
5. 由于此时父进程和子进程均阻塞在等待接收消息的状态，而该函数结束后只会向父进程发送消息，因此在do_fork的最后还需要向新生成的子进程发送消息，解除子进程的阻塞状态。

执行的代码可见 `mm/forkexit.c`

```
PUBLIC int do_fork()
{
    /* find a free slot in proc_table */
    proc_task_struct_t * cur_proc_table = g_pcb_table;
    int cur_empty_pcb_pid = 0;
    for (int i = 5; i < _PROC_NUM; i++){
        if (cur_proc_table[i].status == _PROC_EMPTY){
            cur_empty_pcb_pid = i;
            break;
        }
    }
    int father_pid = mm_msg.source;
    proc_task_struct_t * father_proc = &g_pcb_table[father_pid];
    int child_pid = cur_empty_pcb_pid;
    proc_task_struct_t * child_proc = &g_pcb_table[child_pid];
    com_printk("pid(%d) : do fork! fork to pid(%d)\n", father_pid, child_pid);

    /* 得到了 cur_empty_pcb_pid, 等下就把进程控制块复制一遍, 放到这个pcb上 */
    com_memncpy(child_proc, father_proc, sizeof(proc_task_struct_t));

    /* 分配栈空间, 并将父进程的栈复制到子进程的栈中 */
    uint32_t new_stack = mm_alloc_mem_default(cur_empty_pcb_pid);
    // 栈复制 源地址
    uint32_t src = (uint32_t)father_proc->kernel_stack;
    // 栈偏移量
    uint32_t offset = father_proc->stack_base + sizeof(proc_regs_t) - src;
    // 栈复制 目的地址
    uint32_t dest = new_stack + sizeof(proc_regs_t) - offset;
    com_memncpy((void *)dest, (void *)src, offset);

    /* 修改子进程控制块, 子进程使用新栈 */
    child_proc->kernel_stack = (void *)dest;
    child_proc->pid = child_pid;
    child_proc->p_parent = father_pid;

    /* child PID will be returned to the parent proc */
    mm_msg.PID = child_pid;

    /* birth of the child */
    message_t m;
    m.type = SYSCALL_RET;
    m.RETVAL = 0;
    m.PID = 0;
    msg_send_recv(SEND, child_pid, &m);

    return 0;
}
```

3.2 wait 的实现

对于用户的wait，我是这样子实现的，具体的 `do_wait` 操作，在 `mm/forkexit.c` 中实现。

1. 用户调用 `user_wait` 函数，该函数会向 `TASK_MM` 发送 `WAIT` 消息。
2. `TASK_MM` 收到 `WAIT` 消息，调用 `do_wait` 函数。
 1. 从消息中，得知调用 `WAIT` 的进程的pid
 2. 遍历PCB进程控制块列表，寻找该进程的所有子进程，如果找到了子进程，且该子进程处于HANGING状态，就调用 `cleanup` 函数清除该进程。
 3. 如果没有处于HANGING的子进程，就将进程状态设置为 `WAITING`，阻塞住。
 4. 如果没有子进程，就没有比较阻塞了，直接向进程发送消息返回 `NO_TASK`。

```
PUBLIC void do_wait()
{
    int pid = mm_msg.source;

    int i;
    int children = 0;
    proc_task_struct_t * p_proc = g_pcb_table;
    /*
    遍历进程控制块列表，寻找该进程的所有子进程
    如果找到了子进程，并且该子进程还处于hanging状态，就清除该进程
    */
    for (i = 0; i < _PROC_NUM; i++, p_proc++) {
        if (p_proc->p_parent == pid) {
            children++;
            com_printk("pid(%d) : do wait! I have children pid(%d) status(%d)\n", pid, p_proc->
            >pid, p_proc->p_flags);
            if (p_proc->p_flags & HANGING) {
                cleanup(p_proc);
                return;
            }
        }
    }
    if (children) {
        /* has children, but no child is HANGING */
        g_pcb_table[pid].p_flags |= WAITING;
    }
    else {
        /* no child at all */
        com_printk("pid(%d) : do wait! I don't have children\n", pid);
        message_t msg;
        msg.type = SYSCALL_RET;
        msg.PID = NO_TASK;
        msg_send_recv(SEND, pid, &msg);
    }
}
```

3.3 exit的实现

exit与wait是相互对应的操作。子进程的exit对应着父进程的wait，只有子进程完全exit后，父进程的wait才可以接触阻塞，继续向下执行。

对于用户的exit，我是这样子实现的，具体的 `do_exit` 操作，在 `mm/forkexit.c` 中实现。

1. 用户调用 `user_exit` 函数，该函数会向 `TASK_MM` 发送 `EXIT` 消息。
2. `TASK_MM` 收到 `EXIT` 消息，调用 `do_EXIT` 函数。
 1. 从消息中，得知调用 `EXIT` 的进程的pid
 2. 从PCB中得知该进程的父进程，判断父进程是否处于 `WAITING` 状态
 1. 如果处于 `WAITING` 状态，则直接清除子进程
 2. 如果没有在 `WAITING`，则将当前进程设置为 `HANGING` 状态，等待父进程调用 `wait`
 3. 如果该进程还有子进程，将这些子进程的父进程设置为 `INIT` 进程。

```
PUBLIC void do_exit(int status)
{
    int i;
    int pid = mm_msg.source; /* PID of caller */
    int parent_pid = g_pcb_table[pid].p_parent;
    com_printk("pid(%d) : do exit! My father is pid(%d)\n", pid, parent_pid);
    proc_task_struct_t * p = &g_pcb_table[pid];

    /* 应该有一个清楚内存的函数 */

    p->exit_status = status;

    if (g_pcb_table[parent_pid].p_flags & WAITING) { /* parent is waiting */
        g_pcb_table[parent_pid].p_flags &= ~WAITING;
        cleanup(&g_pcb_table[parent_pid]);
    }
    else { /* parent is not waiting */
        g_pcb_table[pid].p_flags |= HANGING;
    }

    /* if the proc has any child, make INIT the new parent */
    for (i = 0; i < _PROC_NUM; i++) {
        if (g_pcb_table[i].p_parent == pid) { /* is a child */
            g_pcb_table[i].p_parent = TASK_INIT;
            if ((g_pcb_table[TASK_INIT].p_flags & WAITING) &&
                (g_pcb_table[i].p_flags & HANGING)) {
                g_pcb_table[TASK_INIT].p_flags &= ~WAITING;
                cleanup(&g_pcb_table[i]);
            }
        }
    }
}
```

4 遇到的问题

4.1 一个有关子进程地址空间与父进程地址空间不一致的问题。

我曾出现过一个问题：子进程返回的pid值不对。

```
PUBLIC int user_fork()
{
    message_t msg;
    msg.type = FORK;

    msg_send_recv(BOTH, TASK_MM, &msg);
    // assert(msg.type == SYSCALL_RET);
    // assert(msg.RETVAL == 0);

    return msg.PID;
}
```

在用户的fork函数中，我使用了一个FORK消息来发送，同时该消息也兼顾接受返回值的作用。问题的出现在于，这是一个父进程中的局部变量，在父进程的局部堆栈中。当我派生出子进程后，该子进程中的堆栈存有的仍然是父进程的message绝对地址。因此返回的消息仍然在使用着父进程的内存，但是在C语言编译的过程中，该消息是使用esp来索引访问的，子进程的esp与父进程是不同的（给子进程申请了一段新的堆栈）。

因此，对于子进程而言，mm进程返回的消息复制到了父进程的内存空间中，而子进程使用相对寻址，因此子进程无法访问到mm进程返回的消息，也就是说，访问的地址是错误的，这就导致了子进程返回的pid值不对。

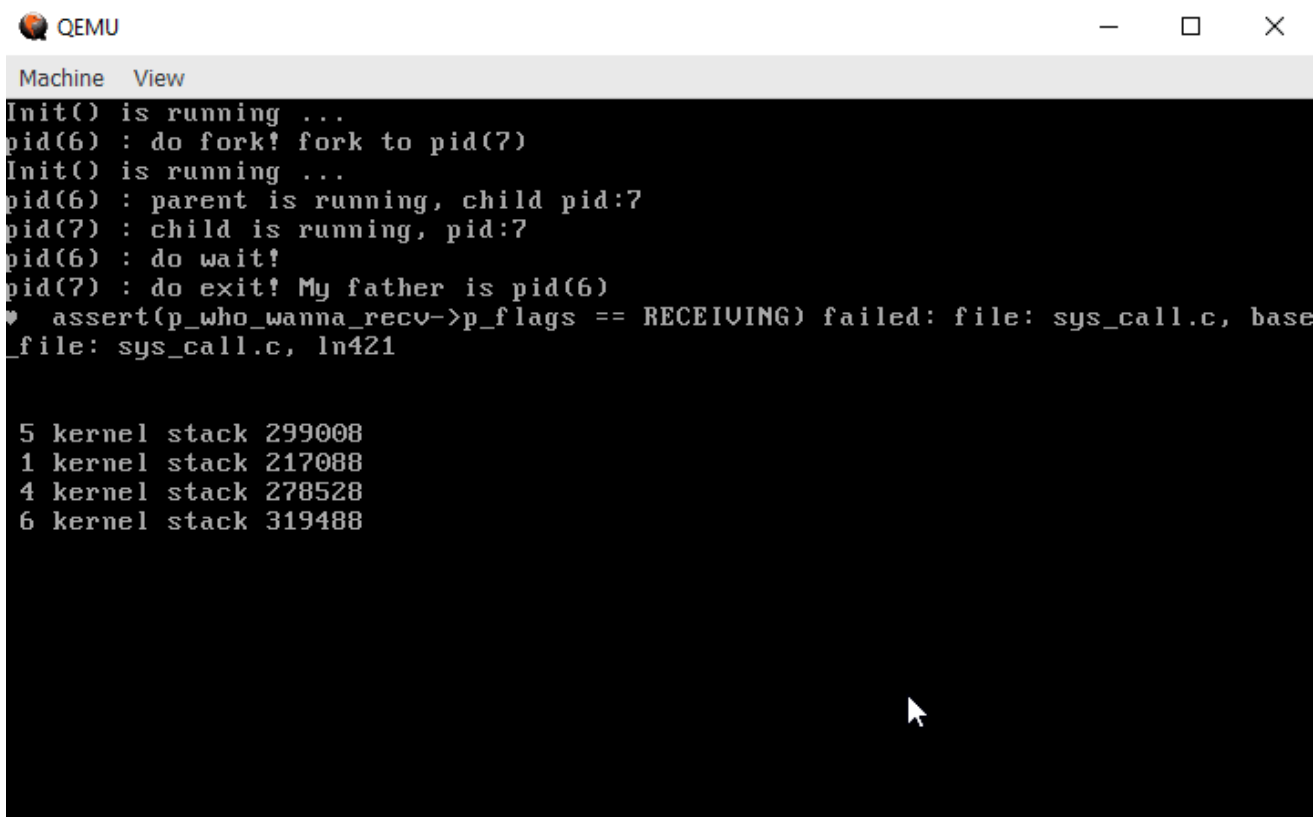
最根本的问题在于子进程和父进程的地址空间不一致。如果有页表的话，其实可以做到通过更换页表的方式，达到地址空间不变，而使用不同的物理内存。不过由于我目前的页表还比较简单，虽然开启了页表，但是初始化为“线性地址等于物理地址”的形式。因此这个问题我暂时还比较难解决。

我的解决方法是：将fork所使用的消息变量，改为全局变量，这样子就避免了相对寻址导致的问题。

也就是说，将上面的代码修改为：

```
message_t fork_msg; // 全局！！
PUBLIC int user_fork()
{
    fork_msg.type = FORK;
    msg_send_recv(BOTH, TASK_MM, &fork_msg);
    return fork_msg.PID;
}
```

4.2 一个小问题

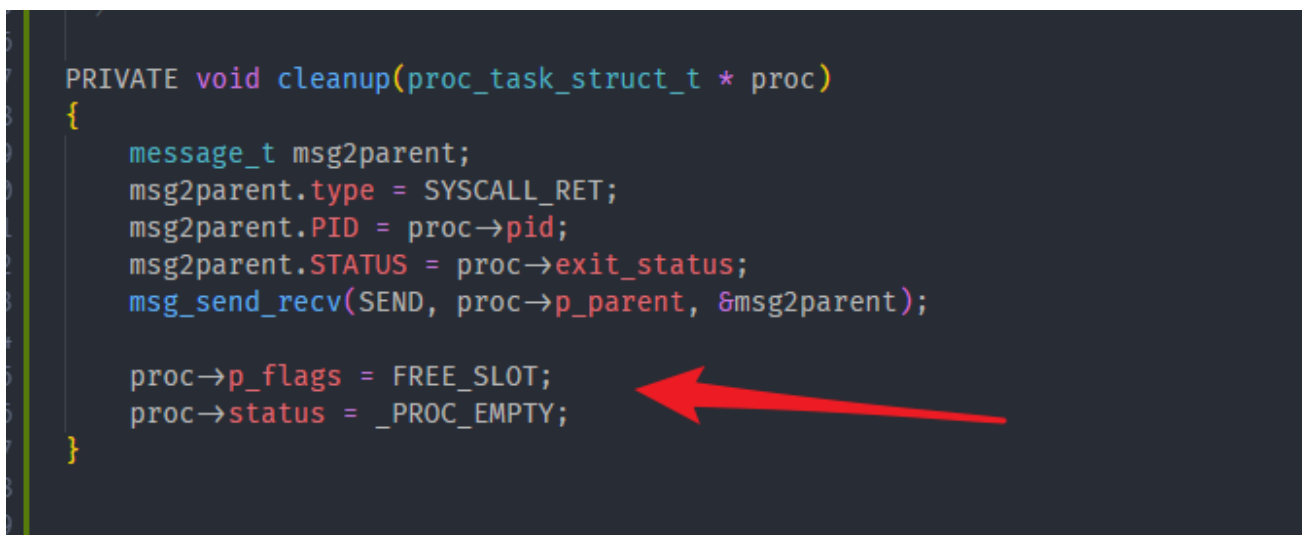


```
Machine  View
Init() is running ...
pid(6) : do fork! fork to pid(7)
Init() is running ...
pid(6) : parent is running, child pid:7
pid(7) : child is running, pid:7
pid(6) : do wait!
pid(7) : do exit! My father is pid(6)
♥ assert(p_who_wanna_recv->p_flags == RECEIVING) failed: file: sys_call.c, base
_file: sys_call.c, ln421

5 kernel stack 299008
1 kernel stack 217088
4 kernel stack 278528
6 kernel stack 319488
```

在我编写wait和exit的时候，发生了这样的问题，代码中的assert检测到了问题，及时提示消息，停止了系统的进一步运行。后来发现这个是因为我忘记了在清楚进程控制块的时候，往status打上一个已空的标记。

补上这个就好了

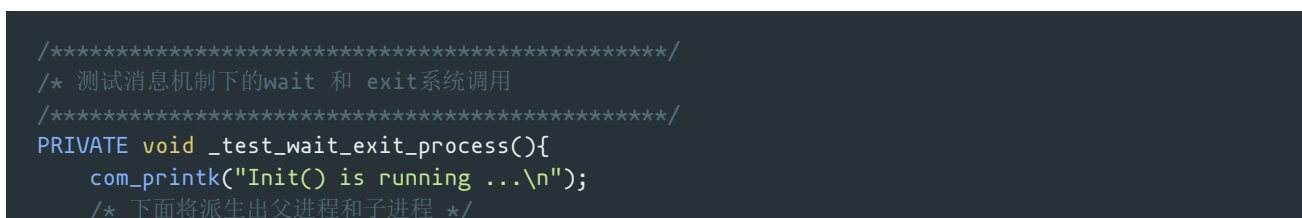


```
PRIVATE void cleanup(proc_task_struct_t * proc)
{
    message_t msg2parent;
    msg2parent.type = SYSCALL_RET;
    msg2parent.PID = proc->pid;
    msg2parent.STATUS = proc->exit_status;
    msg_send_recv(SEND, proc->p_parent, &msg2parent);

    proc->p_flags = FREE_SLOT;
    proc->status = _PROC_EMPTY;
}
```

5 测试过程

我使用了这样的代码进行测试，可见 `test/test.c`



```
/* *****
/* 测试消息机制下的wait 和 exit系统调用
/* *****
PRIVATE void _test_wait_exit_process(){
    com_printk("Init() is running ...\n");
    /* 下面将派生出父进程和子进程 */
```

```

int pid = user_fork();

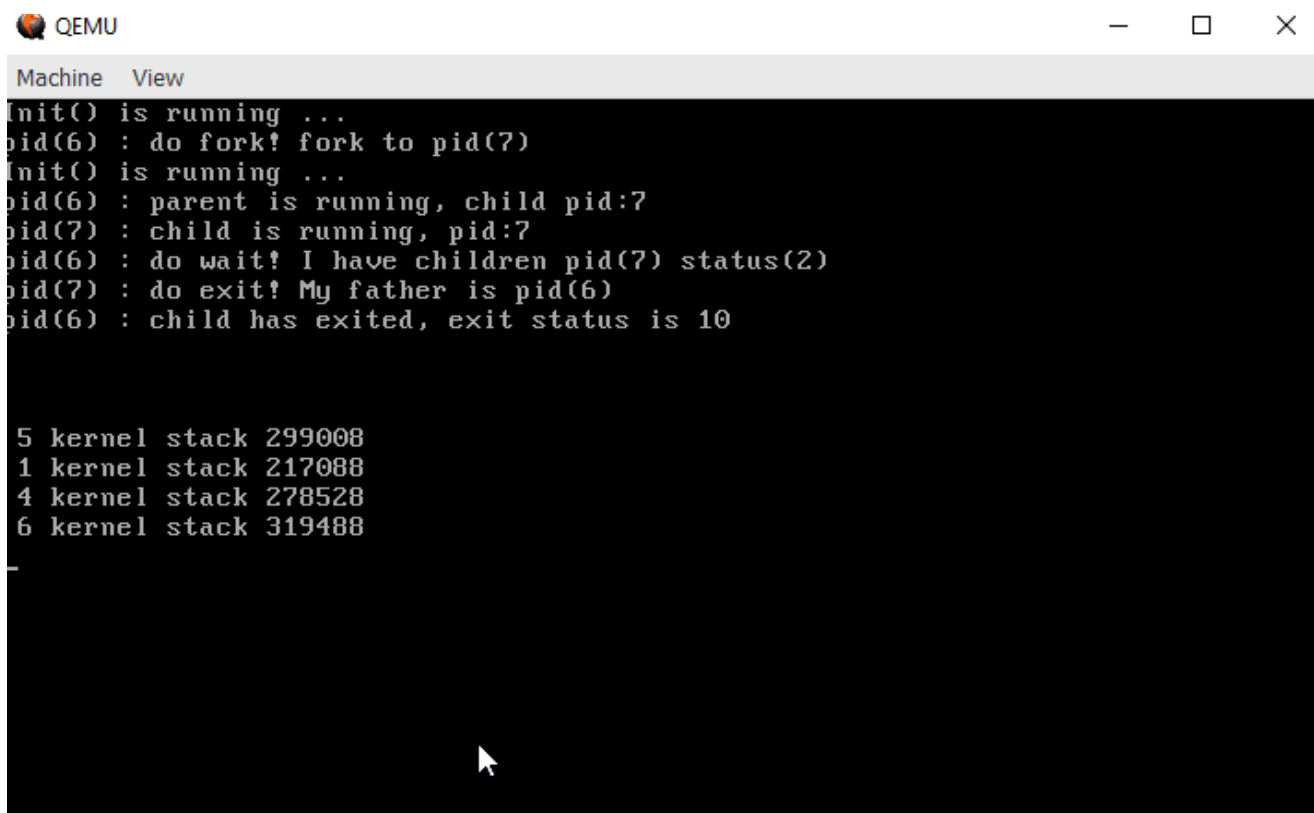
if (pid != 0) { /* parent process */
    com_printk("pid(%d) : parent is running, child pid:%d\n", user_get_pid(), pid);
    int status;
    user_wait(&status);
    com_printk("pid(%d) : child has exited, exit status is %d\n", user_get_pid(), status);
}

else { /* child process */
    com_printk("pid(%d) : child is running, pid:%d\n", user_get_pid(), user_get_pid());
    user_exit(10);
}
while (1){}
}

```

这部分代码做的是派生出两个父进程和子进程，然后父进程调用 `user_wait` 等待子进程，子进程调用 `user_exit` 让父进程从wait中返回，父进程从而打印出子进程的退出状态。

程序的运行结果如下图所示，可以看见，既成功地创建了父进程和子进程，同时父进程能够等待子进程的退出，并且获取子进程的退出状态。



The image shows a QEMU window with a terminal output. The output displays the execution of a program that forks a child process, waits for it to exit, and prints the exit status. Below the main output, there is a list of kernel stack addresses for different processes.

```

Machine  View
init() is running ...
pid(6) : do fork! fork to pid(7)
init() is running ...
pid(6) : parent is running, child pid:7
pid(7) : child is running, pid:7
pid(6) : do wait! I have children pid(7) status(2)
pid(7) : do exit! My father is pid(6)
pid(6) : child has exited, exit status is 10

5 kernel stack 299008
1 kernel stack 217088
4 kernel stack 278528
6 kernel stack 319488

```

6 实验感想

这一次我的实验实现了fork，exit，wait三个系统调用。

这几个函数虽然说是系统调用，不过还是一个包装在发送接收消息上的接口。这一次实验我主要的代码贡献在设置多一个MM进程，这个进程能够帮助我简单的管理内存，同时处理一些fork，exit，wait相关的事务。

这一次的难点主要在栈段的复制上。我自己的进程栈段的大小，以及起始位置，都需要自己去算出来，然后再使用 `com_memncpy` 函数进行复制，在复制的过程中一旦出错，也许整个子进程就再也回不来了。幸运的是，我应该是前期对这个比较谨慎，后面一次调试成功。反倒是一些我本来没有认为是问题的细节，卡了我很久。

编写这一次代码的思路主要来自于orange的实现，不过由于有很多地方和orange并不兼容，fork函数其实是完全的重写了，orange设置了ldt，也对gdt表做了一些设置，我自己的操作系统简单起见，其实并没有去设置。但是改动这些代码的话，错综复杂的依赖关系，又让我很担心会不会出事，也就不敢改了。

在编写操作系统的过程中，我越发的体会到，完成一个完善的系统，真的能力要求特别高。我一直让我的操作系统更加优雅，简洁，但是在编写的过程中，越来越多的历史遗留问题，让我自己写的代码依赖性越来越强。这些依赖性，比如说，一些模块内常量与操作系统常量的混杂，比如说模块内私有函数与操作系统公有函数的混杂。再比如说，一些不得不设置的全局变量。再加上我的操作系统对页表的支持并不是特别好，在进程切换的时候并不能够顺便切换页表实现地址空间的不变，导致我的一些操作不能够很优美的实现。

其实我本来想写一下文件系统的。我一直很着迷于linux的VFS（虚拟文件系统），想到各种操作都可以抽象为文件的读写，就觉得很优雅，很想用在自己的操作系统上。了解过一些具体的实现，但是放到自己的操作系统上，还是有很多很难解决的问题需要我去花费大量的时间去解决。到现在为止我还没有将键盘驱动放到操作系统，导致目前用户的交互性仍不够好，甚至比以前我在实模式下编写的操作系统还不如。不过，就发展的潜力而言，当然是目前我现在已经进入保护模式的操作系统潜力大了，别的不说，就可用地址空间而言就胜过实模式一大截。