# 16 级计科 7 班: 操作系统原理实验 #7(保护模式)

Due on Wednesday, June 13, 2018

凌应标 周一 *9-10* 节

**颜彬**

**16337269**

# Content

# 1 实验目的

在实验六或更后的原型基础上, 进化你的原型操作系统, 原型保留原有特征的基础上, 设计满足下列要求的新原型操作系统:

实现控制的基本原语 do_fork() 、do_wait() 、do_exit() 、blocked() 和 wakeup()

内核实现三系统调用 fork() 、wait() 和 exit() , 并在 c 库中封装相关的系统调用

编写一个 c 语言程序, 实现多进程合作的应用程序

# 2 实验过程

## 2.1 fork 逻辑

代码 1: C 库中封装的 fork

```
#define fork() \
({\
    cli();\
    int32_t __fork_ret;\
    __asm__(\
        "movl $0x02, %%eax\n"\
        "int $0x80\n"\
        : "=r"(__fork_ret)\
        :\
    );\
    sti();\
    __fork_ret;\
})
```

如代码1所示，将 fork 封装到 C 函数库中。在进入 fork 的前后要关中断和开中断。

本代码最重要的细节在于，它不以函数的形式实现。这是因为函数结束时的语句一般是 leave 和 ret。这两条语句会对 esp 的值有影响。但是在 fork 完后，子进程会继承父进程的 eip（在本项目的实现中，是这样处理的）。所以子进程被调度后，会继续执行 fork 函数的后半部分。为了避免 fork 的 leave 和 ret 弄乱子进程的 esp 和 ebp，fork 不应实现成函数调用。

另一种实现的方法是，修改子进程的 eip, 使得子进程被调度时，返回到 fork 的下一条语句中。

由于 fork 有返回值，故这里利用了 C 语言的 ({...}) 语法。这个语法中，C 程序会把... 中的代码全部执行完，然后把... 中的最后一个表达式作为整个表达式的值。这样，在程序调用 int pid = fork(); 时，可以恰好把 pid 赋为 fork 的返回值。

### 2.2 sys_fork 逻辑

在 fork 中，通过调用 0x80 中断的 0x2 号功能实现 fork。如代码2所示。

代码 2: sys_fork 逻辑

```
fn_ptr sys_call_table[] = {
    test_print, print_hello, sys_fork, sys_wait, sys_exit
};

...

int sys_fork() {
    printks("[debug]fork\n");
    int32_t pindex = first_empty_pcb();
    if (pindex == -1) {
        return -1;
    }
    copy_process(pindex, current);
    return PCB_List[current].register_image.eax;
}
```

sys_call_table 是一个函数数组，存储着若干函数的首地址。在调用 0x80 号中断时，功能号会成为索引，跳转到该地址表中。所以 fork 的 0x2 功能号会跳转到函数 sys_fork. 其中数组类型 fn_ptr 是一个 typedef, 类型是 int (*) ()，即返回 int 的，不接受参数的函数。

在 sys_fork 中，首先调用 first_empty_pcb 得到第一个空闲的 pcb 表。然后调用 copy_process ，将当前的 pcb 表复制到空闲的 pcb 表中。随后显式地返回当前进程的 eax 的值。

### 2.3 copy_process 逻辑

如代码3所示。

首先将源 PCB 中的寄存器副本复制到目的 PCB 中。源和目的的 eax 需要根据 fork 的语义特殊处理。

目的 esp 和 ebp 需要特殊地计算。程序为每个（内核）进程分配固定的栈空间。所以 PCB 索引和栈空间可以相互计算的。在 fork 后，旧 esp 和 ebp 相对于旧栈的位置，等于新 esp 和新 ebp 相对于新栈的位置。利用这个条件，可以求出新 esp 和新 ebp。

由于实验 6 中，进程切换的寄存器都是通过在栈中 Pop 来恢复的（实现细节问题），故此处需要把部分寄存器中 PCB 中手动复制到栈中，覆盖掉旧的内容。这是通过计算栈中寄存器（如 eax）和栈顶的相对位置来实现的。

最后，设置 pid, parent_pid, 以及进程的状态。

代码 3: copy_process 的主要代码

```
void copy_process(int32_t dst_index, int32_t src_index) {
    struct RegisterImage* dst = &PCB_List[dst_index].register_image;
    struct RegisterImage* src = &PCB_List[src_index].register_image;
    int32_t new_pid = last_pid++;

    src->eax = new_pid;
    dst->eax = 1;
    dst->ecx = src->ecx;
    ...
    dst->cs = src->cs;

    int32_t esp_offset = ProcessStack(src_index) - src->esp;
    int32_t ebp_offset = ProcessStack(src_index) - src->ebp;
    _rev_memcpy(
        (void*)ProcessStack(dst_index),
        (void*)ProcessStack(src_index), 1024
    );
    dst->esp = ProcessStack(dst_index) - esp_offset;
    dst->ebp = ProcessStack(dst_index) - ebp_offset;

    uint32_t* pesp = (uint32_t*)dst->esp;
    *(pesp + OFFSET_EAX) = dst->eax;
    ...
    *(pesp + OFFSET_EAX) = dst->eax;

    PCB_List[dst_index].state = TASK_INTERRUPTIBLE;
    PCB_List[dst_index].pid = new_pid;
    PCB_List[dst_index].parent_id = src_index;
}
```

### 2.4 __rev__memcpy 的实现

_rev_memcpy 是反向内存复制的函数。它专门用来在 fork 中复制栈的值。

由于栈从高地址增长到低地址，采用反向内存复制会更便于编程。该函数暂不考虑跨段的问题。

## 3 实验结果

### 3.1 fork 测试

测试样例的代码如4所示。

在主进程中，通过 fork 产生进程 2 和 3. 在进程 2 和 3 中，分别再调用 fork，产生进程 4 和 5. 通过在各个分支中输出 111 到 666，可以检验 fork 的确正确工作了。

程序初始化时，会利用实验 6 的成果，手动产生新进程。在新进程中，同样调用 fork，并输出 777 和 888。当在屏幕上最终显示 111 到 888，且恰好显示一次时，说明所有的 fork 都正常工作。

代码 4: 测试样例代码

```
int main(){
    sti();
    int id = fork();
    if (id == 1) {
        printks("111\n");
        int id2 = fork();
        if (id2 == 1) {
            printks("333\n");
        } else {
            printks("444\n");
        }
    } else {
        printks("222\n");
        int id2 = fork();
        if (id2 == 1) {
            printks("555\n");
        } else {
            printks("666\n");
        }
    }
    while(1);
    return;
}
void second_process() {
    int id = fork();
    if (id == 1) {
        printks("777\n");
    } else {
        printks("888\n");
    }
    while(1);
}
```

fork 正常工作的情景如图1所示。

### 3.2 wait 和 exit 测试

在输出 888 的分支之前加入 wait 指令，888 将不会输出，如图2所示。

但是在 777 的分支中加入 exit 指令后，由于子进程的 exit 解除了父进程的阻塞，888 分支又可以再次输出到屏幕上。如图3所示。

值得注意的是，wait + exit 的情况和一开始的图1不同。区别在于，各个进程被调度的顺序是不同的。

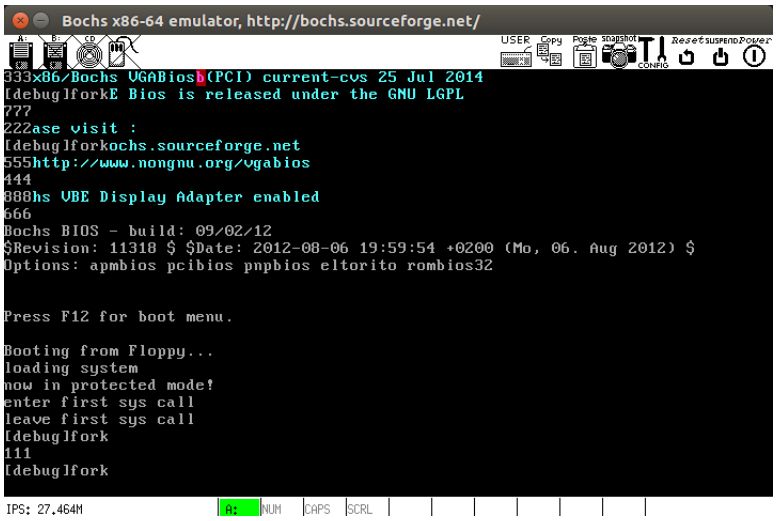


图 1: 进程切换和 fork 结合的测试样例

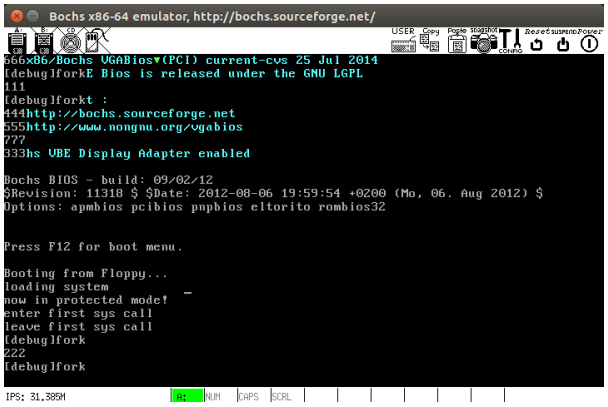


图 2: 加入一条 wait 指令的图片

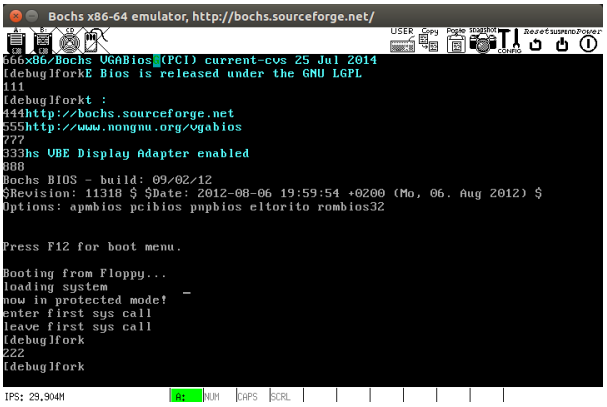


图 3: 再加入 exit 指令后的图片

# 4 实验总结

## 4.1 实验心得

实验七拖得比较晚才交，因为我尝试了保护模式，在保护模式下重做了前面的几个实验，并最终实现了实验 7. 保护模式耗费了我不少时间。其中无数的试错和重写，让我的代码改了又改。

这个实验我感觉略有困难，难度不亚于实验 6。这是因为我之前参考了 Linux 的保护模式的实现，其在中断方面和进程切换方面，思路和课内的思路（也是大部分同学的思路）都不一样。这导致了我在做实验 7 时，几乎把进程切换和系统调用又再重写了一遍。其中还陷入许多常规保护错误中。debug 比较困难。

遇到最坑的一点是，汇编代码中的 label, 在 C 程序中，需要用 &label 的方式取得它的地址。我在做栈切换时，需要取得汇编中栈的起始地址，但一直不知道要加 & 号，导致取出来的值非常奇怪（实际上取出来的是 label 标签处内存的值）。

在 fork 中把栈复制后，需要把新进程的 esp 和 ebp 正确地设置，由于历史实现的问题，进程在恢复时，除了 ss 和 sp 以外的所有寄存器的值都是从栈中恢复的。所以 copy_process 还要负责把栈中的某个位置的寄存器的值正确地修改。

# 附录 A 参考文献

1. https://blog.csdn.net/longintchar/article/details/50602851
   16 位和 32 位汇编指令的不同（尤其是 push 指令）
2. https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1
   GCC 内嵌汇编的书写。