实验六：实现时间片轮转的二态进程模型

实验目的

1. 学习多道程序与CPU分时技术
2. 掌握操作系统内核的二态进程模型设计与实现方法
3. 掌握进程表示方法
4. 掌握时间片轮转调度的实现

实验要求

1. 了解操作系统内核的二态进程模型
2. 扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。
3. 修改时钟中断处理程序，调用时间片轮转调度算法。
4. 设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。
5. 修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。
6. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

实验方案

实验环境

硬件：个人计算机

操作系统：Linux Manjaro

虚拟机软件：Bochs

实验的模式：x86_64长模式

实验开发工具

开发环境：Linux

语言工具：x86汇编语言、C语言

编译器：gcc

汇编器：as（gcc里的汇编器）

汇编调试工具：bochsdbg

链接器：ld

反汇编工具：objdump

打包文件工具：ar

磁盘映像文件浏览编辑工具：bless（Linux下类似于WinHex的工具）

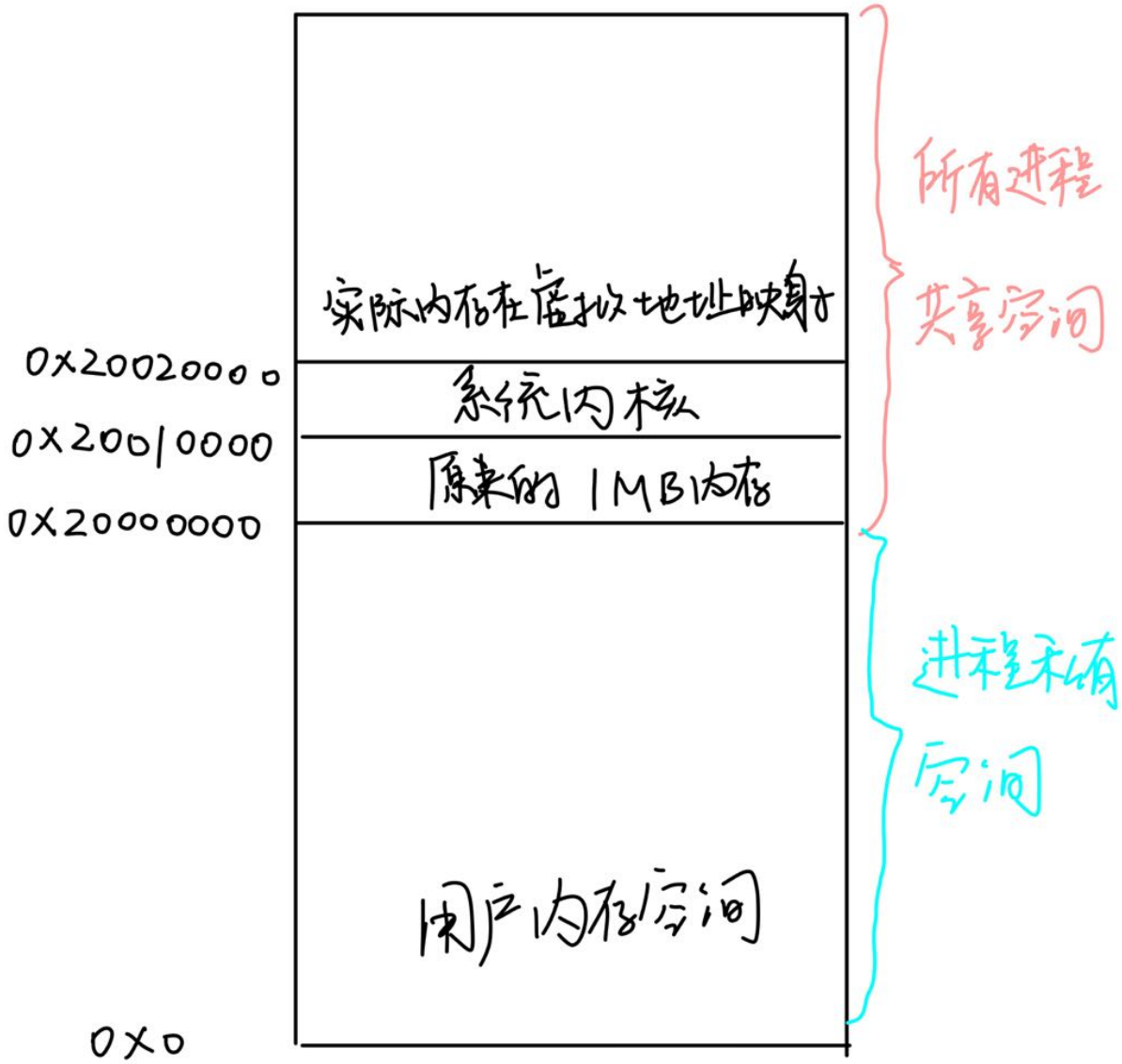
代码编辑器：Visual Studio Code

程序设计

根据实验要求，本次实验主要的工作如下：

- 重新设计长模式下的虚拟内存分配
- 增加虚拟内存管理
- 修改实验五设计的PCB数据结构
- 修改函数save()和restart()，用于保存进程的PCB结构和恢复PCB结构
- 增加进程调度者，实现时间片轮算法
- 增加一个多进程运行系统调用

虚拟内存分配图



考虑到本操作系统实验的简洁性，我重新限制了整个操作系统虚拟内存地址空间大小——1G。接着，因为操作系统里涉及到进程页表的切换、新页框的动态分配等等，我先是把原来的1MB内存空间映射到虚拟地址 `0x20000000` - `0x20010000`，而系统内核所处在的内存地址是 `0x20010000` - `0x20020000`。而剩下的物理内存（起始物理地址为 `0x30000`），放到 `0x20030000`。

这样就得到一个公式：逻辑地址（虚拟地址） = 物理地址 + 0x20000000。

这么设计的话，我们可以在虚拟地址空间中，轻松地控制内存实际的物理地址，为后面页框的分配奠定基础。

虚拟内存管理

这部分主要有两个函数`Kmalloc`和`Kfree`。前者用于分配新的页框，后者用于销毁已有的页框。

Kmalloc()

页框分配算法我利用的是原理课上讲到的`first-fit`算法。从头开始扫描内存表，直到第一次遇到空闲的。

该函数主要流程：

- 从`0x20020000`内存地址开始进行扫描
- 判断页表项里的Present位，直到遇到该位为0的项停止
- 返回对应的物理地址

源代码：

```
/* 请求新的页框,返回的是空闲的物理页框号 */
uint64_t Kmalloc() {
    for (uint64_t i = FreeMem; i < Memory_Size; i += Page_Size) {
        /* 每个页表第0项跳过 */
        if (i % FreeMem == 0)
            continue;

        /* 寻找该页对应的页表地址 */
        uint64_t virt_addr = PHY2VIR(i);
        uint64_t *pt_addr = PT_Addr(virt_addr);

        /* 判断它的Present位是否为0 */
        if (!(pt_addr[PT_Index(virt_addr)] & PRESENTED)) {
            pt_addr[PT_Index(virt_addr)] |= PRESENTED;
            return i;
        }
    }
}
```

注：为了方便内存管理，对于每512个页，我取了第一页作为页表进行管理，故这里的第一个判断条件是为了跳过页表

Kfree()

这个函数是上面函数响应的释放资源函数。而释放操作，就是把Present位置为0即可。

源代码：

```
/* 释放输入的物理页框号 */
void Kmfree(uint64_t mem_addr) {
    uint64_t virt_addr = PHY2VIR(mem_addr);
    uint64_t *pt_addr = PT_Addr(virt_addr);

    /* 将页表里该项置为Unpresented */
    pt_addr[PT_Index(virt_addr)] -= 1;
}
```

多进程模型

PCB数据结构

参考课本给出简化版进程控制块

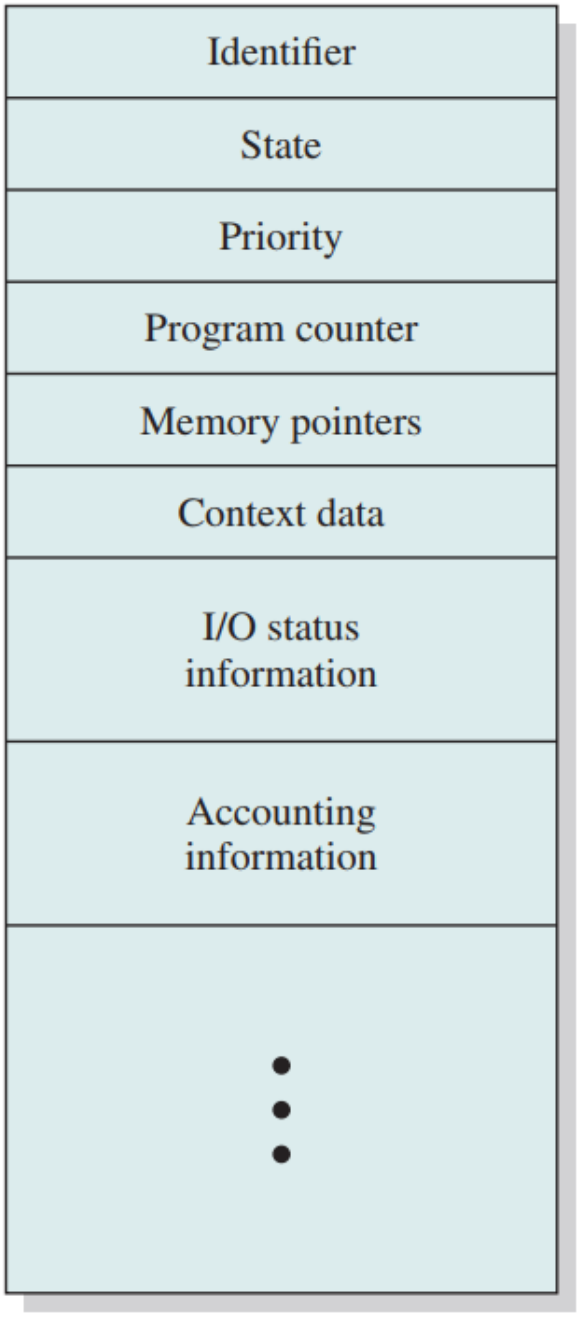


Figure 3.1 Simplified Process Control Block

整个结构体：

```
struct Proc {
    struct TrapFrame *tf; /* 进程上下文 */
    pde_t *pd;  /* 页表 */
    int proc_state; /* 进程状态 */
    uint64_t proc_id; /* 进程上下文 */
};

struct TrapFrame {
    uint64_t rax;
    uint64_t rbx;
    uint64_t rcx;
```

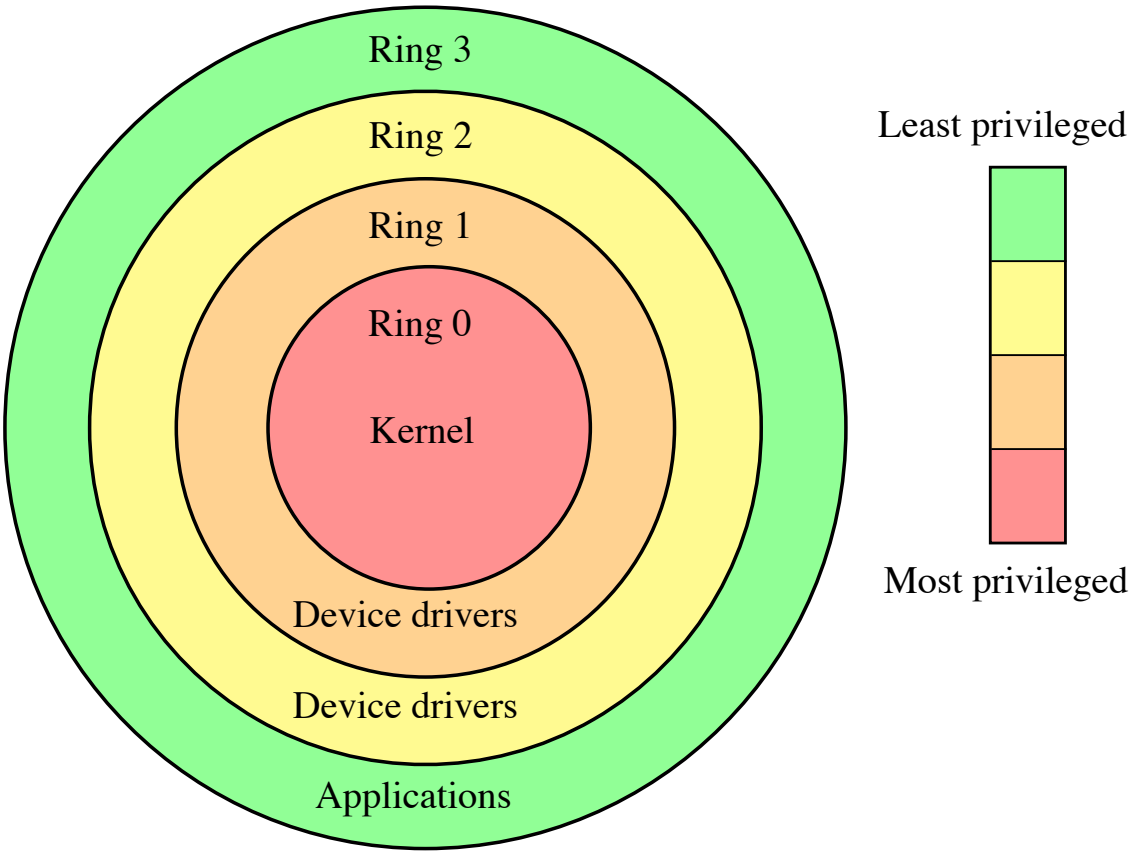
```
    uint64_t rdx;
    uint64_t rsi;
    uint64_t rdi;
    uint64_t rbp;
    uint64_t r8;
    uint64_t r9;
    uint64_t r10;
    uint64_t r11;
    uint64_t r12;
    uint64_t r13;
    uint64_t r14;
    uint64_t r15;

    /* 中断返回iret时堆栈的情况 */
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
    uint64_t rsp;
    uint64_t ss;
};
```

从上面可以看出，我设计的PCB和课本给出的基本相同，可以满足多进程模型的实现。

进程权限设计

相比于实模式，保护模式引入了特权级的概念，不同特权级之间数据和代码以及一些汇编指令使用等方面都是不一样的。

分级保护域：



根据上图，我给内核代码段设置了Ring 0的特权级，而在用户进程里设置了Ring 3。这样处理，可以利用保护模式的功能，使得系统内核的数据对于用户是不可访问的，保证系统的安全性。

而从Ring 0特权级转移到Ring 3特权级这一过程，就引出了我对于`restart`函数的设计。

save()和restart()对于多进程模型的修改

因为本次实验引入了特权级的概念（之前一直默认是在Ring 0的环境下执行），就必须考虑Ring 0到Ring 3和Ring 3到Ring 0这两个过程了。

restart()

根据文档，前者的实现主要依赖`ret`指令和`iret`指令。这里我选择了`iret`指令。下面是restart()函数。

```
restart:
    # EOI
    movb $0x20, %al
    out %al, $0x20

    # 恢复原来的寄存器
    pop %rax
    pop %rbx
    pop %rcx
    pop %rdx
    pop %rsi
    pop %rdi
```

```
    pop %rbp
    pop %r8
    pop %r9
    pop %r10
    pop %r11
    pop %r12
    pop %r13
    pop %r14
    pop %r15
    iretq
```

上面的`pop`是按照前面`struct TrapFrame`的顺序。值得注意的是，该结构体中还剩下下面五项：

```
    /* 中断返回iret时堆栈的情况 */
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
    uint64_t rsp;
    uint64_t ss;
```

这部分的设计则跟`iret`的实现有关系。

下图是中断调用后堆栈的情况。

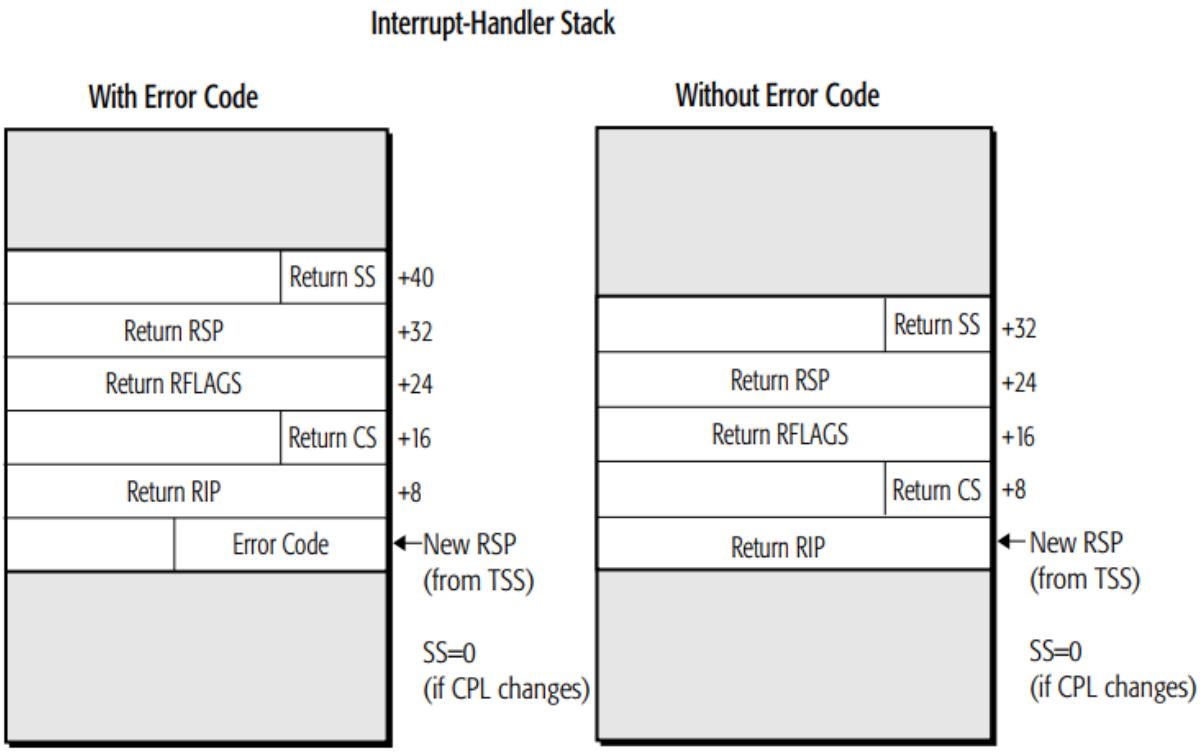


Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege

根据上图，在`iret`返回的时候，会依次将这些寄存器的值进行出栈。因此，这就可以修改上图中的CS选择子，从而更改特权级。同时，`RSP`寄存器的修改，也可以实现进程堆栈切换，构成完全的进程上下文切换。

save()

而save()函数正好跟这个过程相反：

```
save:
    # 记录调用者的返回地址
    pop temp_addr

    # 保存所有的寄存器
    push %r15
    push %r14
    push %r13
    push %r12
    push %r11
    push %r10
    push %r9
    push %r8
    push %rbp
    push %rdi
    push %rsi
    push %rdx
    push %rcx
    push %rbx
    push %rax

    movq %rsp, %rax

    # 恢复返回地址
    push temp_addr
    ret
```

TSS寄存器设置

要实现Ring 3到Ring 0的切换，只需要配置中断向量表项里的CS选择子的RPL即可。但是从用户进程切换会内核代码，考虑到安全性因素（因为用户的堆栈可能无效等等之类的原因），需要同时切换到系统堆栈，这就需要用到长模式里的TSS寄存器了。

根据官方文档的描述：

3. If a privilege change occurs, the target DPL is used as an index into the long-mode TSS to select a new stack pointer (RSP).

如果进入中断的时候发生特权级转换，在切换的时候，CPU会自动载入预先设定好的`RSP`的值，即进行堆栈切换。

而TSS的格式如下

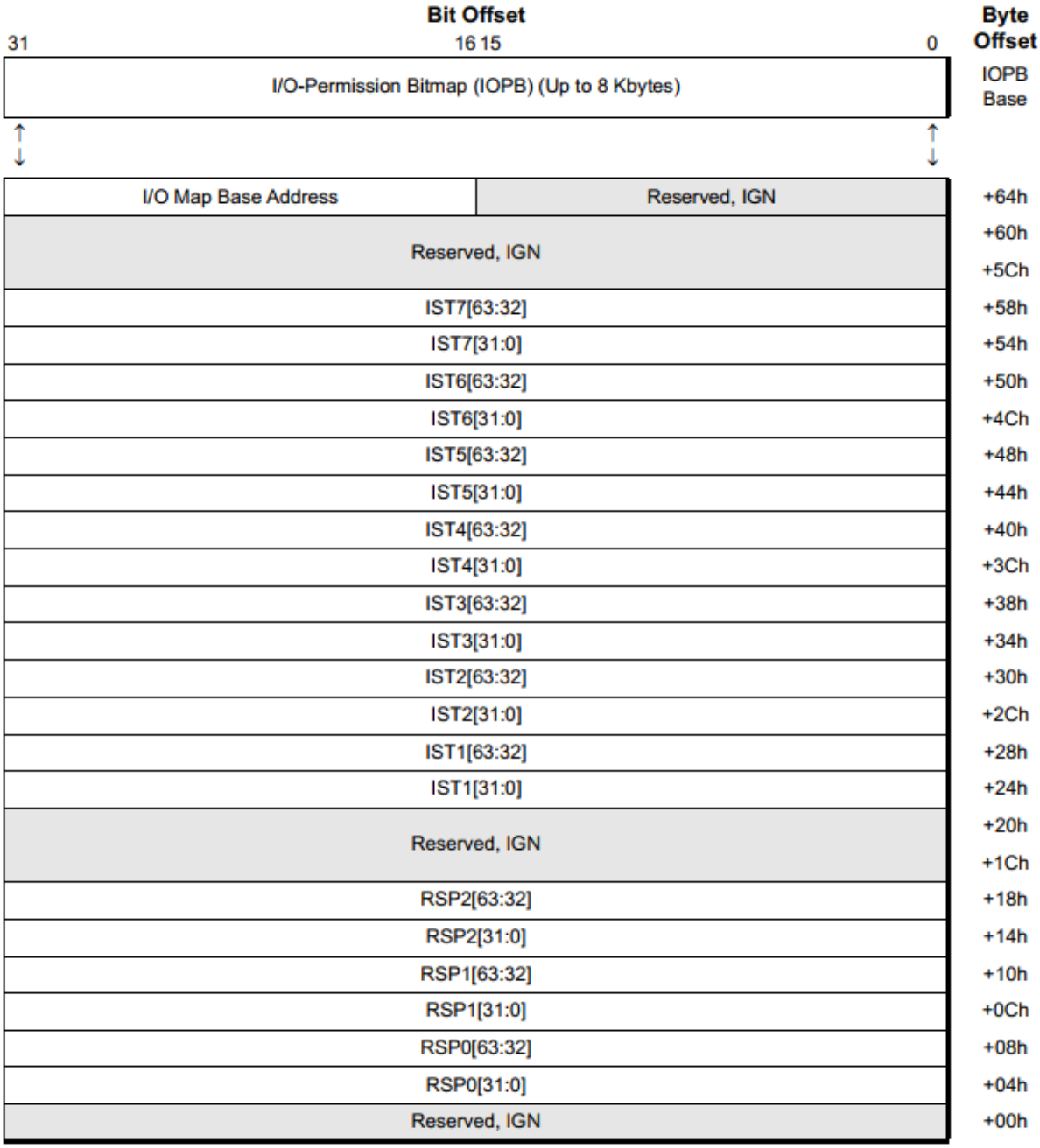


Figure 12-8. Long Mode TSS Format

我们需要实现的是Ring 3到Ring 0的转换，只需要配置RSP0即可。

初始化进程

完成了前面一系列基础函数的铺垫以后，我们终于可以着手开始实现多进程了。

系统实现流程

正如我在实验五中设定的，考虑到shell是与用户进行交互的，可以算是一个应用程序。因此，我把shell程序从内核代码转变为一个特殊的用户进程。因此，整个系统的实现流程变为了：

引导程序：

- 切换模式进入保护模式
- 设置虚拟内存

- 进入长模式
- 加载系统内核到内存
- 跳转到内核主程序

内核：

- 初始化TSS
- 初始化时钟中断（无敌风火轮）
- 初始化键盘中断（用于处理输入）
- 初始化系统调用（int 0x80)
- 进入shell程序

shell：

- 打印提示符
- 等待用户输入
- 使用系统调用exec加载用户程序

内核：

- 创建相应进程的PCB
- 控制权交给进程调度程序Scheduler
- 加载Ready进程

用户程序：

- ...
- `exit`把控制权交回给内核

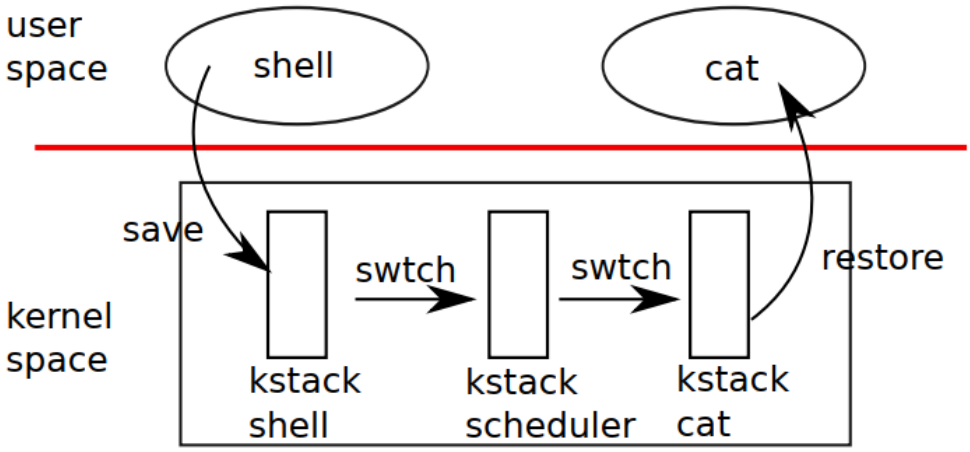
内核：

- 回收进程资源
- 进程调度程序Scheduler选择状态为Ready的进程进行加载

shell：

- ...

经过转变后，剩下的用户进程和shell的关系就变为了：shell进程通过系统调用`exec`加载用户进程。即如下图（参考的是xv6的设计）：



创建第一个进程

从上面的流程可以看出，`shell`就是我们的第一个进程。下面我们就要考虑在内核里创建第一个进程，并加载执行该进程。

具体流程：

- 初始化系统进程表
- 设置进程号0的进程表
- 为该进程申请一个新的页表
- 为进程的代码段、数据段和堆栈段申请页框
- 加载进程的代码段和数据段到刚刚申请的页框
- 设置进程的上下文
- 更换当前的Page Table Directory
- 设置TSS，确保save能够保存到进程表里的上下文对应的地址
- 将进程表里的上下文地址加载到`rsp`寄存器
- 跳转到restart

```
/* 初始化第一个进程 pid = 0 */
void InitProcess() {
    /* 初始化系统进程表 */
    for (int i = 0; i < Max_Pro_Num; ++i) {
        process_table[i].proc_state = 0;
        process_table[i].tf = TrapFrameMem + i * 100 * sizeof(struct TrapFrame);
    }

    /* 设置进程号0的进程表 */
    process_table[0].proc_id = 0;
    uint64_t pd_addr = Kmalloc();
    process_table[0].pd = PHY2VIR(pd_addr);

    /* 申请页表 */
    uint64_t pt_addr = Kmalloc();
    process_table[0].pd[0] = (pt_addr | Uncached);
```

```
    uint64_t *pt = PHY2VIR(pt_addr);

    /* 申请页框,并加载进程到内存0x0 */
    pt[0] = Kmalloc() | Uncached;
    pt[1] = Kmalloc() | Uncached;
    pt[511] = Kmalloc() | Uncached;
    readsect(PHY2VIR(pt[0] - Uncached), SHELL_SEC, SHELL_NUM);

    /* 设置进程的上下文 */
    process_table[0].tf->cs = 0x1b;
    process_table[0].tf->rip = 0x0;
    process_table[0].tf->rflags = 0x3202;
    process_table[0].tf->ss = 0x13;
    process_table[0].tf->rsp = 0x1fffff;
    running_pid = 0;

    /* 更换Page Table Directory */
    ChangePd(pd_addr);

    /* 跳转运行 */
    void *shell_code_addr = 0x0;
    process_table[0].proc_state = RUNNING;
    tss[0] = (uint64_t)(process_table[0].tf) + sizeof(struct TrapFrame);
    __asm__("mov %0, %%rsp\n\t"::"r"(process_table[0].tf):);
    __asm__("jmp restart\n\t");
}
```

经过前面的设计，现在我的系统是分页式内存管理机制。因此，更换页表可以保证进程间地址空间不同，从而起到进程隔离的保护目的。页表的设计初衷就是加强系统保护机制，与特权级的引入相辅相成。

在保护之余，这里还有内核代码的共享机制。因为系统服务面向所有用户进程的，故系统内核对用户是可见的。虽然如此，用户却只能访问内核中只读的代码，而不能读取或者修改系统数据（因为特权级的限制）。下面是更换页表的代码：

```
/* 更换Page Table Directory,传入参数为物理地址 */
void ChangePd(uint64_t new_pd_addr) {
    /* 原始PDP地址 */
    uint64_t *pdp = PHY2VIR(PDPE_BASE);
    pde_t *new_pd = PHY2VIR(new_pd_addr);
    pde_t *pdt = PHY2VIR(PDT_BASE);

    /* 将内核内存空间复制到用户进程内存空间 */
    for (int i = 256; i < 512; ++i)
        new_pd[i] = pdt[i];

    /* 更换当前CPU的页表 */
    pdp[0] = new_pd_addr | Uncached;

    /* 更新TLB */
    ChangePageTable();
}
```

最下面的更新TLB，是为了刷新CPU的MMU里的缓存，避免内存读取错误。

进程的运行

进程的运行就依赖`restart`函数。通过更换CPU的上下文，再利用`iret`的机制，我们就可以直接从内核切换到shell进程了。

创建新的进程

事实上，创建新进程的方式和创建第一个进程的过程非常相似，下面就直接放出代码了。

```
/* 创建新的进程 sec_no:在硬盘的扇区号 num_of_sec:要连续读取的扇区数,返回进程号 */
int CreateProcess(int sec_no, short num_of_sec) {
    int i;
    for (i = 0; i < Max_Pro_Num; ++i) {
        if (process_table[i].proc_state == UNUSED)
            break;
    }

    /* 设置进程表 */
    process_table[i].proc_id = i;
    uint64_t pd_addr = Kmalloc();
    process_table[i].pd = PHY2VIR(pd_addr);

    /* 为进程申请页表 */
    uint64_t pt_addr = Kmalloc();
    process_table[i].pd[0] = (pt_addr | Uncached);
    uint64_t *pt = PHY2VIR(pt_addr);

    /* 申请页框给内存0x0 */
    pt[0] = Kmalloc() | Uncached;
    pt[1] = Kmalloc() | Uncached;
    pt[511] = Kmalloc() | Uncached;
    readsect((void*)(PHY2VIR(pt[0] - Uncached)), sec_no, num_of_sec);

    /* 设置进程上下文 */
    process_table[i].tf->cs = 0x1b;
    process_table[i].tf->rip = 0x0;
    process_table[i].tf->rflags = 0x3202;
    process_table[i].tf->ss = 0x13;
    process_table[i].tf->rsp = 0x1fffff;
    process_table[i].proc_state = READY;
    return i;
}
```

因为还没有文件系统，只能利用扇区号和扇区数来加载文件了。

回收进程资源

这个过程就是回收相应的PCB结构。主要有两个过程：设置对应的进程表项状态为UNUSED、回收进程申请的页框。

```
void DestroyProcess(uint64_t pid) {
    /* 设置进程表 */
    process_table[pid].proc_state = UNUSED;
    pde_t pd_addr = (uint64_t)process_table[pid].pd;
    uint64_t pt_addr = process_table[pid].pd[0] - Uncached;
    uint64_t *pt = PHY2VIR(pt_addr);

    Kmfree(pt[0] - Uncached);
    Kmfree(pt[1] - Uncached);
    Kmfree(pt[511] - Uncached);
    Kmfree(pt_addr);
    Kmfree(pd_addr);
}
```

进程调度Scheduler

这个程序采用的是时间片轮的算法（即每个时间片轮执行一个进程）。在此基础上，我稍微增加了优先级的概念，即优先执行正在进行的进程以外的进程。同时，因为这是个二态多进程，还没有阻塞这个状态，我只能暂时把shell设为最低优先级，保证在其它进程执行完之前，不会执行shell进程。程序流程：

- 建立进程运行队列
- 将shell设为最低优先级
- 判断除了shell是不是只剩下一个进程了
    - 如果是，继续执行该进程
    - 反之，加载队列中第一个进程

```
void Scheduler() {
    /* 建立进程运行队列 */
    int count = 0;
    int ready_process[Max_Pro_Num];
    for (int i = (running_pid + 1) % Max_Pro_Num; i != running_pid; i = (i + 1) %
Max_Pro_Num) {
        if (process_table[i].proc_state == READY && i != 0) {
            ready_process[count++] = i;
        }
    }

    /* shell设为最低优先级 */
    ready_process[count] = 0;

    /* 判断除了shell是不是只剩下一个进程了 */
    if (process_table[running_pid].proc_state == RUNNING) {
        if (ready_process[0] == 0)
            return;
```

```
        process_table[running_pid].proc_state = READY;
    }

    /* 加载进程 */
    int i = ready_process[0];
    process_table[i].proc_state = RUNNING;
    running_pid = i;
    ChangePd(VIR2PHY((uint64_t)(process_table[i].pd)));
    tss[0] = (uint64_t)(process_table[i].tf) + sizeof(struct TrapFrame);
    asm("mov %0, %%rsp\n\tjmp restart\n\t"::"r"(process_table[i].tf):);

}
```

重新设计进程加载系统调用exec

因为封装好了前面的函数，这部分变得异常的简单，主要流程是：

- 创建新的进程
- 跳转到Scheduler进行进程调度

```
/* 指明要加载的用户程序扇区号和所需扇区数 */
void sys_exec(struct TrapFrame *tf, unsigned int sec_no, unsigned int num_of_sec) {
    /* 创建进程 */
    CreateProcess(sec_no, num_of_sec);
    Scheduler();
}
```

重新设计进程返回系统调用exit

同样的，因为封装好了前面的函数，这部分也变得很简单，主要流程是：

- 回收进程资源
- 跳转到Scheduler进行进程调度

```
void sys_exit() {
    /* 销毁进程 */
    DestroyProcess(running_pid);
    Scheduler();
}
```

多进程载入multiexec

为了实现多个进程同时运行的效果，我新增了一个系统调用号，一次性可以创建指定个数的进程。

与exec不同之处在于，这个系统调用一次性会创建多个进程

```
void sys_multiexec(uint64_t proc_num, uint32_t *sec_no, uint32_t *num_of_sec) {
    /* 创建多个进程 */
    for (int i = 0; i < proc_num; ++i) {
        CreateProcess(sec_no[i], num_of_sec[i]);
    }
    Scheduler();
}
```

实验过程和结果

（1）修改实验5的内核代码，定义进程控制块PCB类型

修改后的PCB：

```
struct Proc {
    struct TrapFrame *tf; /* 进程上下文 */
    pde_t *pd;  /* 页表 */
    int proc_state; /* 进程状态 */
    uint64_t proc_id; /* 进程上下文 */
};

struct TrapFrame {
    uint64_t rax;
    uint64_t rbx;
    uint64_t rcx;
    uint64_t rdx;
    uint64_t rsi;
    uint64_t rdi;
    uint64_t rbp;
    uint64_t r8;
    uint64_t r9;
    uint64_t r10;
    uint64_t r11;
    uint64_t r12;
    uint64_t r13;
    uint64_t r14;
    uint64_t r15;

    /* 中断返回iret时堆栈的情况 */
    uint64_t rip;
    uint64_t cs;
    uint64_t rflags;
    uint64_t rsp;
    uint64_t ss;
};
```

具体设计思路已经在程序设计部分阐述过了，这里就不在赘述了。

（2）增加一条命令可同时执行多个用户程序

这条命令对应的系统调用就是`multiexec`

代码编写

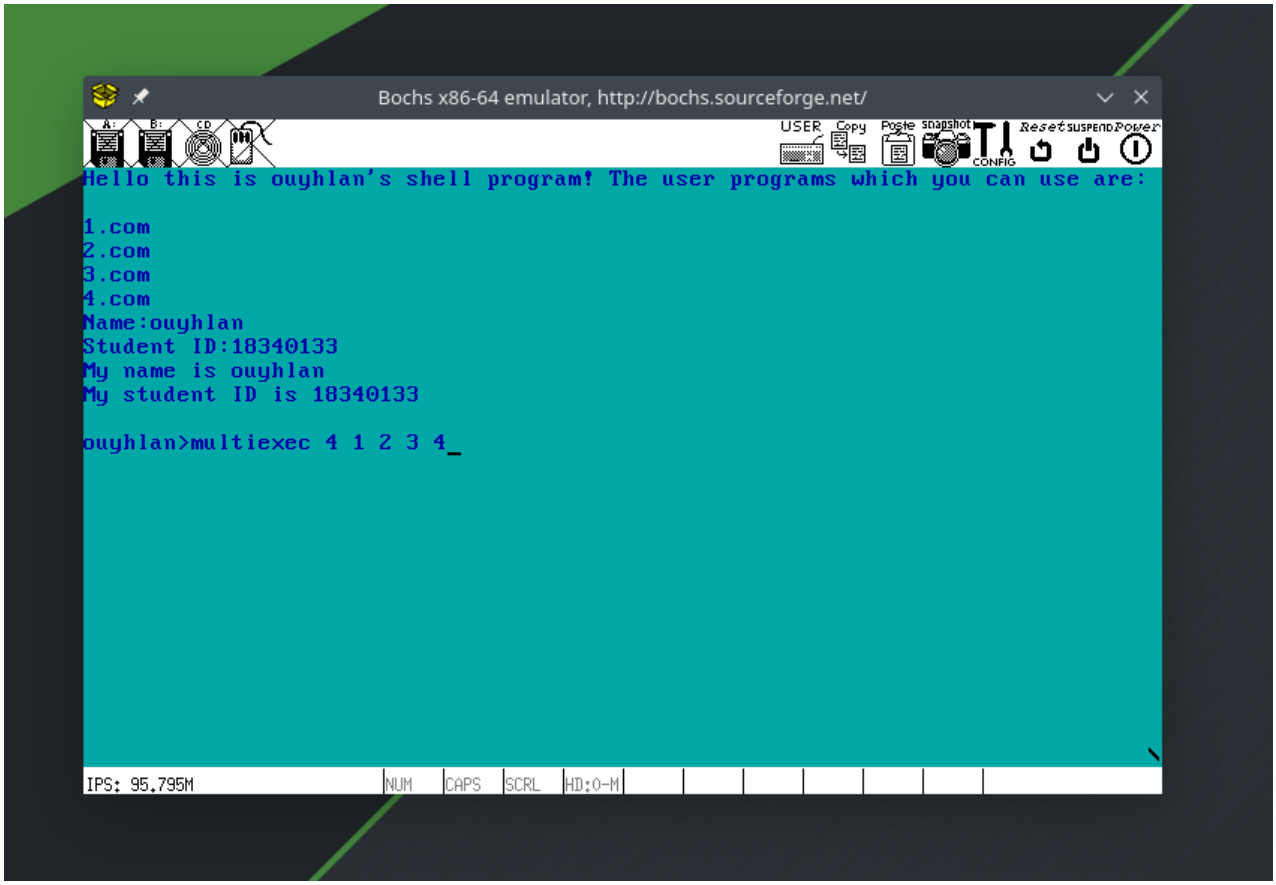
```
void sys_multiexec(uint64_t proc_num, uint32_t *sec_no, uint32_t *num_of_sec) {
    /* 创建多个进程 */
    for (int i = 0; i < proc_num; ++i) {
        CreateProcess(sec_no[i], num_of_sec[i]);
    }
    Scheduler();
}
```

运行截图

命令格式：

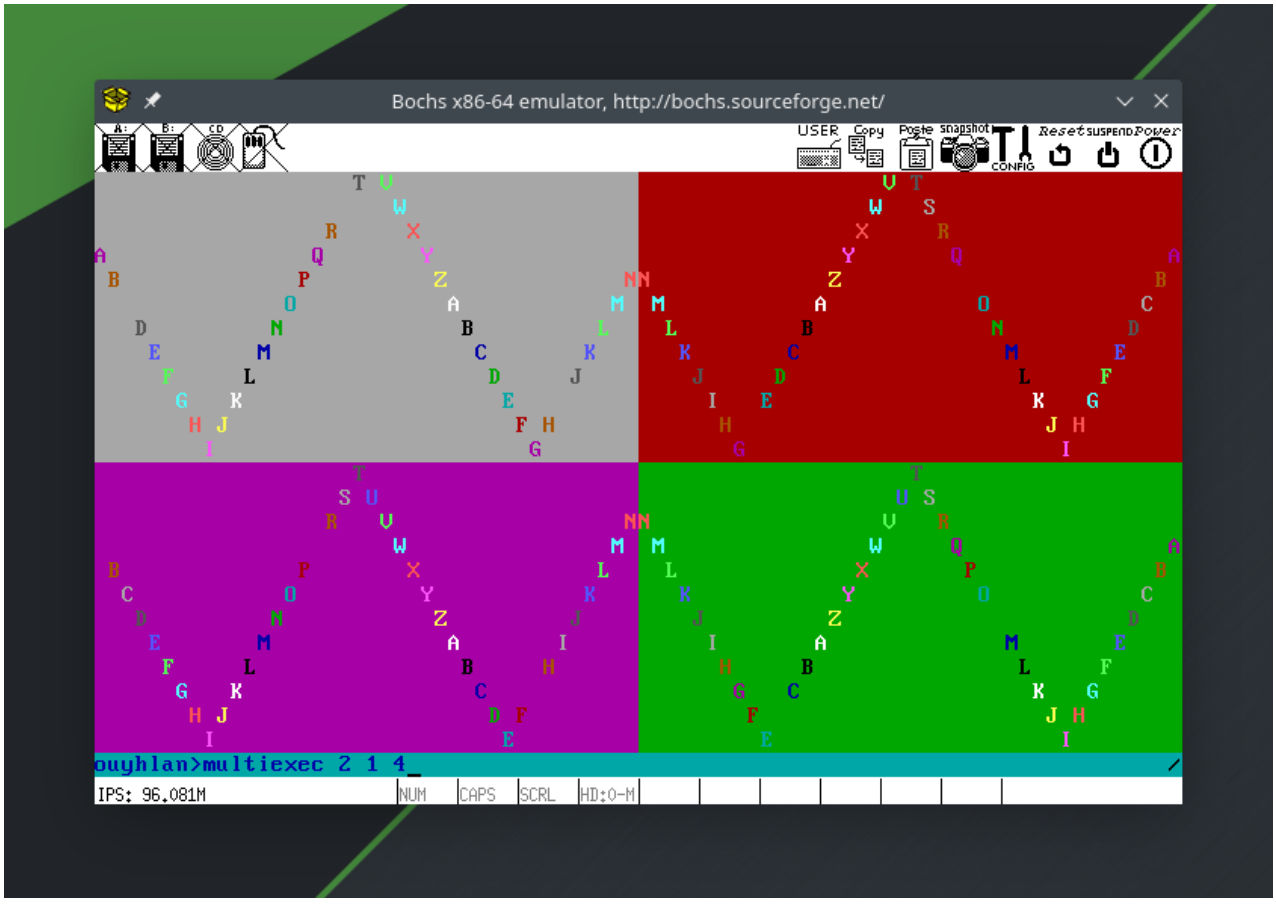
```
multiexec [num_of_progress] [program_id ...]
```

首先是运行4个进程

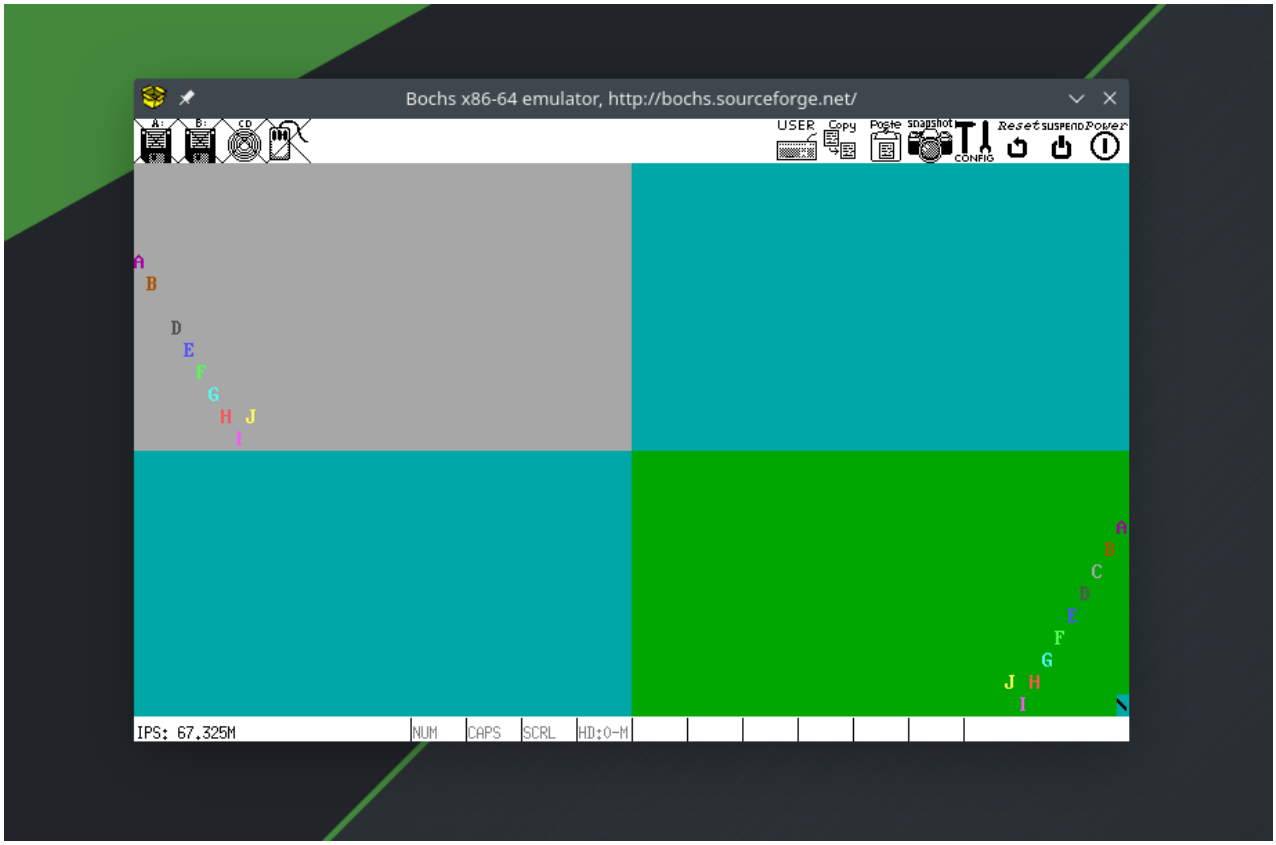


在这里我为了视觉上更加直观，给4个进程设置了不同的背景色：

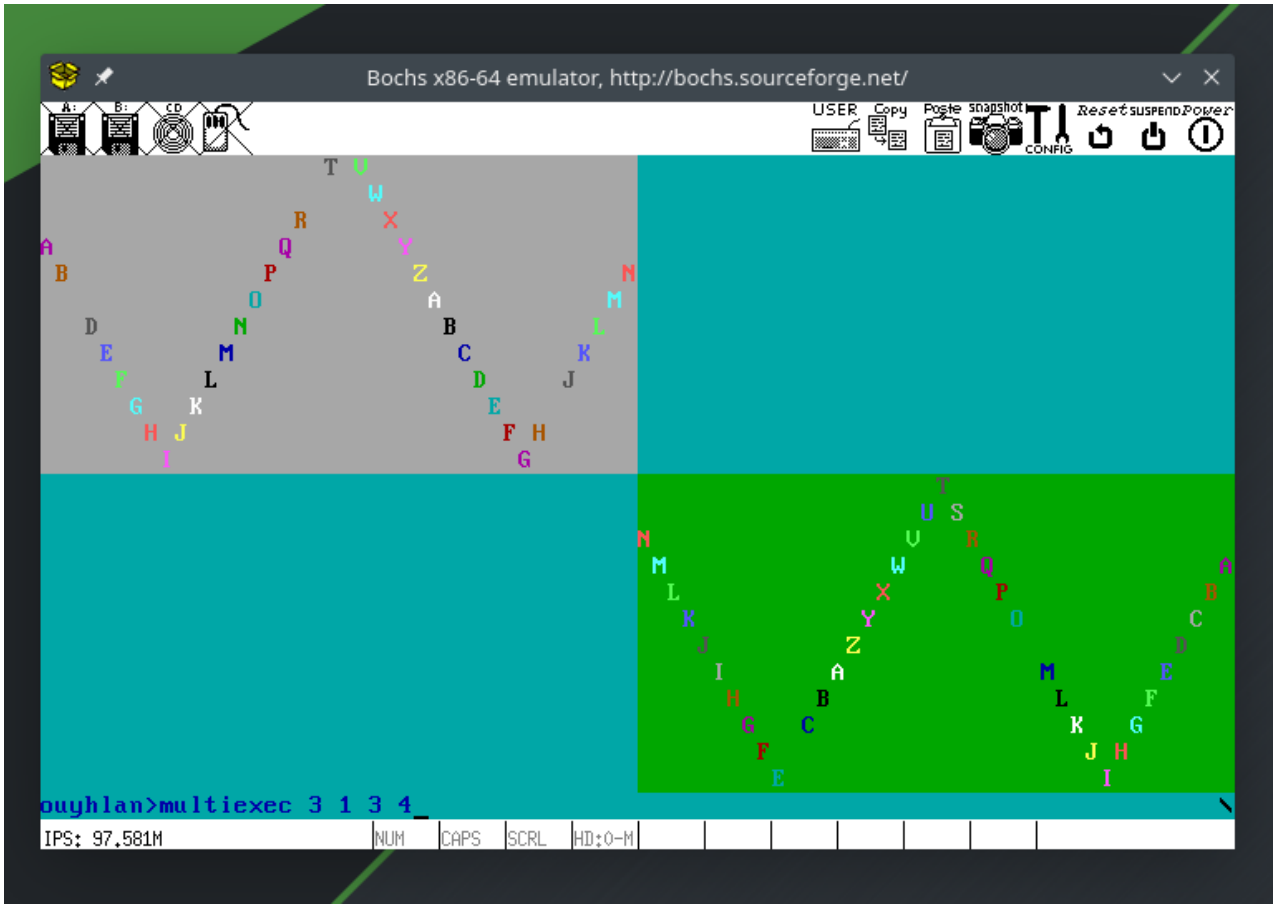
这是运行两个进程的命令



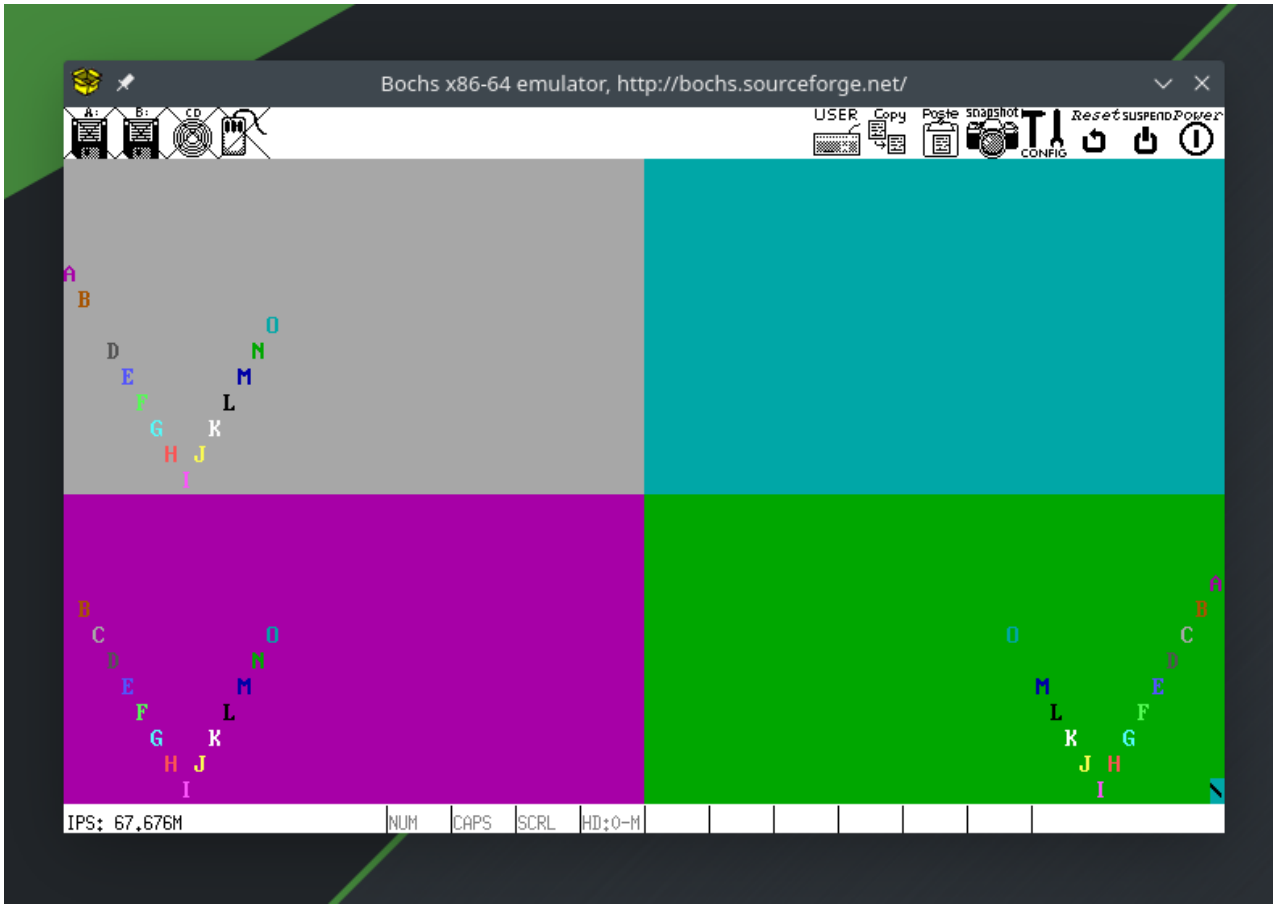
运行效果：



运行三个进程命令



运行效果：



可以看出，我设计的多进程模型能够正常运行，且效果很好。右下角的无敌风火轮一直在转，证明时间片轮算法一直在起作用。

更多详细的运行示例在视频里。

（3）修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程

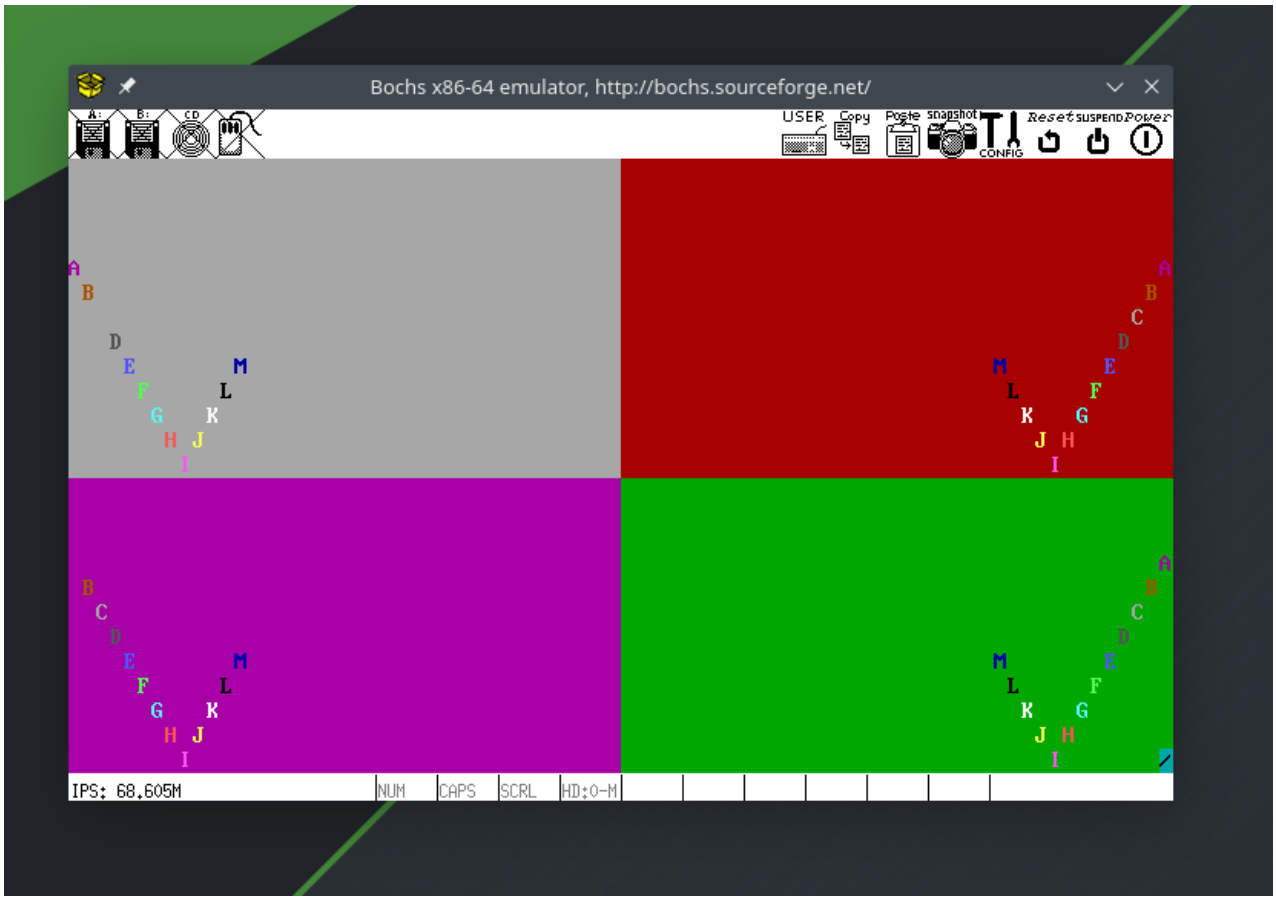
这部分代码就是由原来的无敌风火轮显示，再加上上面设计的调度者程序组成。

时钟中断代码如下：

```
void CcHandler(struct TrapFrame *tf) {
    /* 无敌风火轮 */
    static int cc_index = 0;

    char *hot_wheels_shape = "|/\\";
    unsigned char *vga_addr = 0x200b8000;

    /* 显示风火轮 */
    vga_addr[((24 * 80) + 79) * 2] = hot_wheels_shape[cc_index];
    vga_addr[((24 * 80) + 79) * 2 + 1] = 0x30;

    /* 更新风火轮状态 */
    cc_index = (cc_index + 1) % 3;
```

```
    /* 进程调度程序 */
    Scheduler();
}
```

实现效果：



可以看到右下角的风火轮，且整个屏幕被四个不同的进程占据着，展示了修改的效果。

（4）内核增加进程调度过程

我在调度器的实现上，跟老师的稍微有点不同。我还引入了优先级的概念，使得整个时间片轮算法运行得更加合理。

具体思路在程序设计部分有详细地阐述了，这里就只是贴出代码：

```
void Scheduler() {
    /* 建立进程运行队列 */
    int count = 0;
    int ready_process[Max_Pro_Num];
    for (int i = (running_pid + 1) % Max_Pro_Num; i != running_pid; i = (i + 1) %
Max_Pro_Num) {
        if (process_table[i].proc_state == READY && i != 0) {
            ready_process[count++] = i;
        }
    }
```

```
    /* shell设为最低优先级 */
    ready_process[count] = 0;

    /* 判断除了shell是不是只剩下一个进程了 */
    if (process_table[running_pid].proc_state == RUNNING) {
        if (ready_process[0] == 0)
            return;
        process_table[running_pid].proc_state = READY;
    }

    /* 加载进程 */
    int i = ready_process[0];
    process_table[i].proc_state = RUNNING;
    running_pid = i;
    ChangePd(VIR2PHY((uint64_t)(process_table[i].pd)));
    tss[0] = (uint64_t)(process_table[i].tf) + sizeof(struct TrapFrame);
    asm("mov %0, %%rsp\n\tjmp restart\n\t"::"r"(process_table[i].tf):);

}
```

（5）修改save()和restart()两个汇编过程

save()和restart()两个过程的修改思路也已经在前面的程序设计中提及了，这里只是贴出代码：

```
save:
    # 记录调用者的返回地址
    pop temp_addr

    # 保存所有的寄存器
    push %r15
    push %r14
    push %r13
    push %r12
    push %r11
    push %r10
    push %r9
    push %r8
    push %rbp
    push %rdi
    push %rsi
    push %rdx
    push %rcx
    push %rbx
    push %rax

    movq %rsp, %rax

    # 恢复返回地址
    push temp_addr
    ret
```

```
restart:
    # EOI
    movb $0x20, %al
    out %al, $0x20

    # 恢复原来的寄存器
    pop %rax
    pop %rbx
    pop %rcx
    pop %rdx
    pop %rsi
    pop %rdi
    pop %rbp
    pop %r8
    pop %r9
    pop %r10
    pop %r11
    pop %r12
    pop %r13
    pop %r14
    pop %r15
    iretq
```

实验中遇到的问题

虚拟地址设置

因为我重新修改了整个原型操作系统的虚拟内存部分，在更换页表的时候，会因为没有正确使用物理地址或虚拟地址，而导致该地址在系统无效，使得系统无法运行下去。

缓存机制带来的影响

同时，在本次实验里，我发现了TLB的作用。一开始的时候，因为我为了整个系统的简单性，并没有按照长模式的4层页表结构更换页表。因此，最初我是没有调整CR3寄存器的值（只修改了最后两层的页表）。

但是我发现，系统对应的内存好像是有问题。在切换shell进程和第一个用户程序的时候，原型操作系统并没有显示用户程序1的弹球过程，而是又从开头执行了一次shell程序。这是让我很困惑的一点。

最后，经过思考和理论课学到的知识，我认为是缓存带来的影响。经过查阅官方文档：

- Updates to the CR3 register cause the entire TLB to be invalidated *except* for global pages. The CR3 register can be updated with the MOV CR3 instruction. CR3 is also updated during a task switch, with the updated CR3 value read from the TSS of the new task.

我只要对CR3寄存器进行写操作就可以了：

```
ChangePageTable:
    movq $0x9000, %rax
    movq $rax, %cr3
    ret
```

经过测试，问题确实是出现在这里。

系统调用时参数传递问题

这部分主要是传递指针时候有问题。因为用户传递给系统参数的时候，用的是自己进程里的地址，但是在系统内核操作的时候，两者之间的地址空间可能是不一样的。因此，解决方案要么是使得两个虚拟地址空间相同，要么是将所有参数复制到内核指定位置。

也因为这个原因，我开始理解为什么Linux里的`execve()`函数的参数，必须要以NULL作为结尾了。

实验总结

这次是我在操作系统实验课的第六个实验。在本次实验里，我修改了整个原型操作系统的虚拟内存框架，增加了内存分配和内存回收机制、增加了特权级的变换以及进程调度等等。虽然我的操作环境是长模式，但实际上长模式跟保护模式的机制非常相似——虚拟内存+特权级保护。本次实验就是利用了保护模式里的很多系统保护特性，使得整个原型操作系统变得更加安全有序。

同时，我利用了TSS寄存器，实现了进程切换时，堆栈的切换。这样，使得进程之间的独立性、进程与系统的独立性和安全性都有了大幅度的增强。用户态（Ring 3）和系统态（Ring 0）的相互切换，也是明确了各个进程对内存访问的权限，既共享了系统内核代码段，又保护了系统数据段不被读取和篡改。

虽然本次实验引入了页式内存管理机制，但我还没有实现elf格式文件的加载，也没有完全实现段页式内存管理机制，算是美中不足之吧。后面我会参考一下xv6操作系统在这部分的实现，将完整的段页式内存管理机制移植到我的原型操作系统。

最后是一点是对于实验内容的建议吧。我觉得老师提供的实验要求文档可以适当地调整一下顺序：save和restart函数的修改应该紧接在PCB进程控制块，再接着是进程调度过程。完成了这些以后，才是多进程加载和时钟中断的修改吧。即整体的顺序应该为：

- 定义进程控制块PCB
- 修改save()和restart()
- 内核增加进程调度过程
- 修改时钟中断处理程序
- 增加一条命令可同时执行多个用户程序

按照这个顺序的话，整体实验过程也会比较流畅，难度曲线也会更加平滑吧。