操作系统原理实验

实验五

中断机制编程技术

版本 2

姓名： 吴侃

学号： 14348134

班别： 2014 级计算机系一班

日期： 2016.04.08 – 2016.04.20

目录

零、特色先览

本次实验的亮点包括:

1. 实现进程的多个状态: 就绪态，运行态，挂起态，阻塞态，终止态能方便地完成有效状态的切换
2. PCB 表基本完全由 C++管理
3. 实现了 top 指令，能够显示全部进程及其详细信息
4. 能够通过用户指令唤醒，挂起，结束单个进程
5. 实现 int 20h 中断，可以在用户程序中调用此中断，结束该程序，返回 Shell,不会影响其他程序.
6. 实现 clock()函数，使计时精确到秒.
7. 支持用户指令调用中断(有权限管理)
8. 使用了加锁机制管理共享变量.

以下为版本 2 新增功能：

(搜索“以下为版本 2 新增内容：”能知道哪里进行了修改，谢谢☺)

9. 实现了 fork()函数
10. 使用端口机制实现了进程间的通信，并且带有信号量
11. 32 位 G++与 16 位 NASM 交叉编译时中断处理程序的实现

尝试实现内存管理(版本号 448)，但没有成功，现在提交的版本号为 371.

一、实验目的

学习中断机制知识，掌握中断处理程序设计的要求

二、实验要求

1. 实现一个逆时针逐步填满整个屏幕的程序

2. 编写键盘中断响应程序，当键盘有按键时，在某个地方显示 OUCH!OUCH!

3. 实现 33h,34h,35h,36h 号中断，调用 4 个用户程序（为了后续可扩展性， 我将它们改了中断编号）

4. 扩展系统调用，编写一个测试系统调用的用户程序.

三、实验环境与工具

（一）实验环境

物理机操作系统: Arch Linux 4.4.5-1

调试使用虚拟机: qemu-system-i386, bochs

虚拟机软件: VMware Workstation 12 Pro

虚拟机配置: CPU: i7-4702MQ @ 2.20GHz，使用单核单线程

内存:4 MB

硬盘:32 MB

（二）实验工具

编辑器: Vim 7.4

汇编工具: NASM 2.11.08

C++编译器: g++ 5.3.0

链接工具: GNU ld 2.26.0.20160302

构建工具: GNU Make 4.1

调试工具: Bochs x86 Emulator 2.6.8

虚拟机: qemu-system-i386

dosbox 0.74

VMWare Workstation 12 Pro

合并文件: dd

四、实验方案

（1）中断的调用与 iret

调用中断:

int 中断号

当调用中断时，先往栈压入标志寄存器(flags)，CS，IP，然后跳转。

中断返回:

中断程序中，执行 iret 时，将 IP,CS,Flags 依次出栈，会返回到

位置 CS:IP。

（2）中断开启与屏蔽

开启为 sti，屏蔽为 cli. 当一个中断程序要调用另一个中断时，使用 sti 开启中断.

注意: 不可屏蔽中断无法用 cli 屏蔽.

（3）中断向量表

在第二个实验中，我已经实现一个中断，能够判断是否按下某键，决定是否返回 Shell.

16 位实模式的中断向量表的第 i 个中断的地址:0x0000:i * 4

然后中断程序的 偏移量，段地址 各占两个字节

可以修改这段内存，实现自编中断.

（4）保存原中断程序方法

可以将原中断程序的偏移量保存到某个变量，在实现自编中断时，pushf 并且 call

far 原中断地址，即可.

（5）时钟中断

当时钟发生中断时，会调用 08h 中断.

需要发送 EOI 到主 8529A 和从 8529A:

mov al,20h

out 20h,al

out 0A0h,al

设置时钟中断频率:

需要 1 秒中断 x 次:

mov ax, 1193182/x

out 40h,al

mov al,ah

out 40h,al

注意，使用 out 时，只能用 al 寄存器依次输入.

我使用了时钟中断实现分时系统,

分时系统的实现见第三次 GCC+NASM 交叉编译的实验报告,

大概原理是保存当前寄存器的内容到某段内存，进程调度,

从某段内存读取数据写入寄存器，跳转.

（6）键盘中断

当键盘有按键或释放按键事件时，调用 09h 中断.这里保留了原中断程序，每次调用 09h 中断时，INT09H_FLAG 在 0 或 1 中变换,该变量可以被 C++获取.根据该值，决定是否显示 OUCH!OUCH!

（7）INT 33h,34h,35h,36h 中断

当调用这些中断时，将在变量 INT_INFO 中设置要调用的程序的

编号. 当内核发现 INT_INFO 不为 0 时，调用对应用户程序，并置 INT_INFO 为 0

（8）加锁机制

在多进程管理中，需要对共享变量 INT_INFO 运用加锁机制保护，当该值为 0 时，检测到该状态的程序可以修改 INT_INFO 的值. 该操作为类似原子操作. 当该值为非零时，程序只能等待该值变为 0

（9）系统调用

int 20h, int 33, int 34, int 35, int 36 都是系统调用， 能够修改某处内存的值。这个过程对于用户程序是不可见的， 由内核来完成操作。

（10）进程的多个状态

进程的状态用 PCB 表中的 STATE 值表示，使用时钟中断中的 WKCN_TIMER 例程,

在调度程序附近对进程的状态进行自动变换. 也可以在 C++中, 对进程状态进行修改.

调度程序只会选择 Running 状态的进程; 会对被标记为 Dead 的进程进行销毁,标记状态为 0. 对于就绪态的进程，会将其转换为运行态.

杀死进程，只需设置对应进程的状态为 Dead.

20h 中断，根据 RunID(当前运行的进程 ID)，将对应进程的状态设置为 Dead.

（11）Shell 中调用中断:

首先获取中断程序的段地址和偏移量，然后压入标志寄存器，远程调用该中断程序.

（12）逆时针将字符填充屏幕:

这个很简单，改一下字符转移的方向，设置边界值，当坐标的某个值遇到边界值时，进行转向．使用一个变量进行计数，当画了 25*80 个字符时，退出程序．由于我的分时系统使用了时钟中断，因此将这个程序变成了普通的用户程序.

（13）top 实现

由于 16 位实模式的 C++程序中的指针实际有效位为 16 位，即不包括段地址．因此，要读取任意内存的变量，需要使用内嵌汇编，将该内存的内容写入 C++程序的某个变量中.

top 的实现即将 PCB 段对应程序信息写入一个结构体中，然后输出对应信息.

以下为版本 2 新增内容：

(14) fork()函数

fork()的原理是将父进程所占的内存段复制到新的空间,并复制父进程的 PCB 表。需要修改的信息为 PCB 表中，段寄存器的值。这个过程中,我觉得最困难的是 IP 寄存器的值的处理。我的处理方式是：在 fork()函数刚开始执行时，调用 8 号中断(时钟中断)，保存当前父进程的寄存器值（使对应 PCB 为最新），此时 PCB 中的 IP 值指向调用 8 号中断的下一条指令。接下来重新取最新的 PCB 值，进入一条判断语句。经过判断 PCB 中的值，可以区分出父进程和子进程，若是父进程,创建新进程,再返回 0;若是子进程,直接返回自己的 pid.

(15) 端口通信

我的实现方式为由一个内存段管理端口到一个变量的段地址和偏移地址,并且有对应端口属性值(如信号量,数据大小)。这里使用了 3 号端口记录内存段分配值，5 号端口为 IO 端口。当从 5 号端口写入字符串时,内核将会检测到该信息,并将字符串输出,并将内容清空。

(16) 32 位 G++与 16 位 NASM 交叉编译的 C++中断处理程序

经反汇编发现,G++在有-m16 参数下编译出的文件仍是 32 位文件，其中的 iret 弹出的是 eip,ecs,eflags; 当使用 16 位 NASM 编译出的

程序调用 C++编写的中断处理程序时，压入的是 flags,cs,ip.

因此 ,我编写了 CPP_INT_END 和 CPP_INT_LEAVE 两个宏来分别处理 C++中不带全局变量和带全局变量的中断处理程序。

原理为将 16 位的值扩展为 32 位，高位填零。

(17) 系统调用

这里实现了多个系统调用。

```
;21H 中断
;AH = 00h, 切换 ShellMode 到 al 状态
;AH = 01h, 切换进程状态到 al
;AH = 02h, 得到当前进程 ID
;AH = 03h, 返回 PCB_SEGMENT
;AH = 04h, 返回 PROG_SEGMENT
;AH = 05h, 返回 MSG_SEGMENT
;AH = 06h, 返回 MaxRunNum
;AH = 07h, 停止时钟
;AH = 08h, 开启时钟
;AH = 09h, ++RunNum

;22H 进程, 进程通信
;ah = 00h 读
```

;ah = 01h 写

;ah = 02h 信号量设置(bh=0, 清零; bh=1, 加 1; bh=2, 减 1; bh=3, 设置为 bl 值)

;ah = 03h 设置端口

;ah = 04h 关闭端口

;ah = 05h 只返回信号量

;al = 端口值

;基地址 bx, 缓存大小 cx, 段地址 dx

;返回信号量(ax)

//24h 中断

//转换大小写

//功能号: ah

//偏移量: bx

//段地址: dx

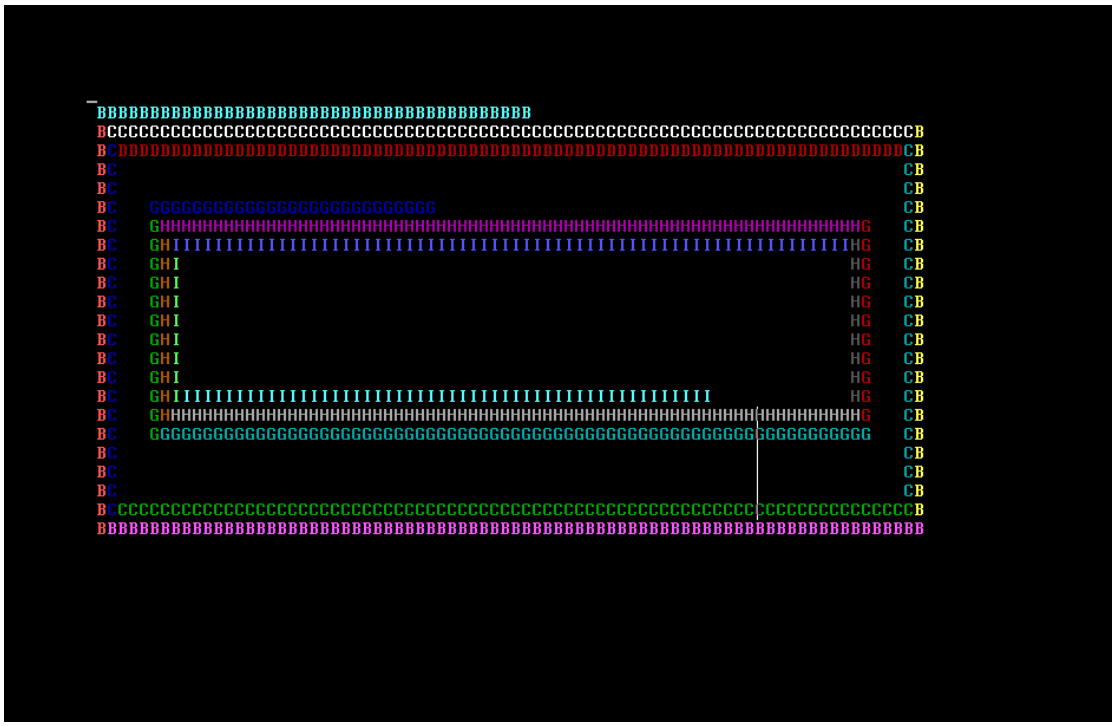
//ah = 0, 全部转小写

//ah = 1, 全部转大写

还有 20h 中断（返回 Shell），23h 中断（打印一个简单的字符串）等等

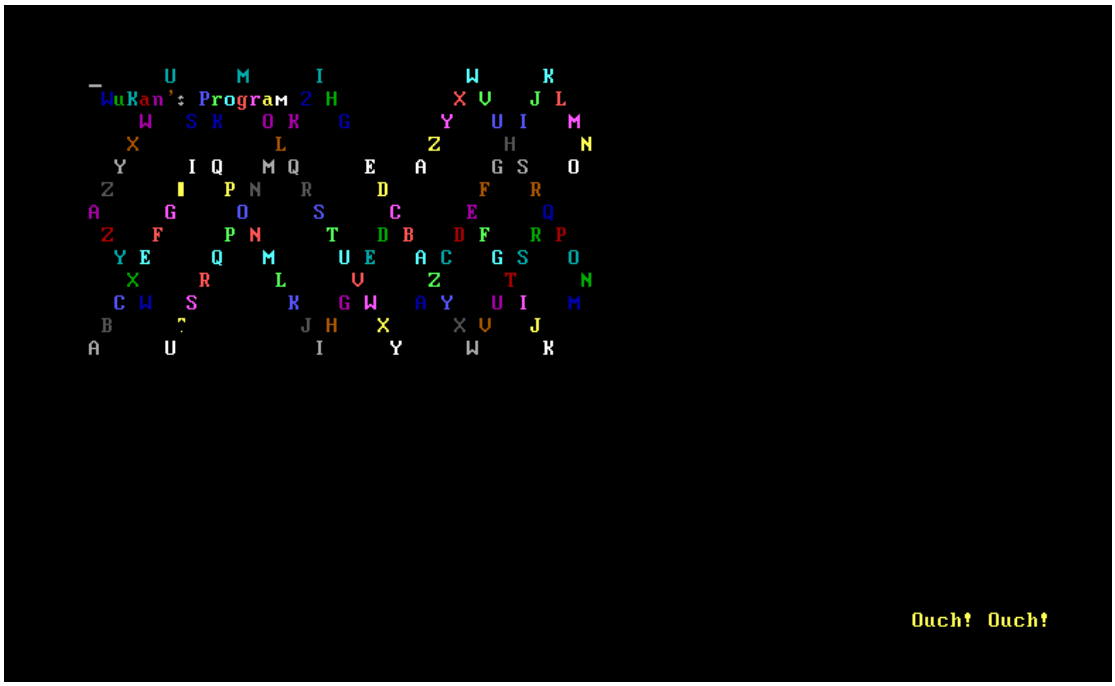
五、实验操作

使用两个进程运行 box 程序(在 Shell 输入 box, Esc 返回， 再输入 box, 按 r 切换到用户程序模式)：

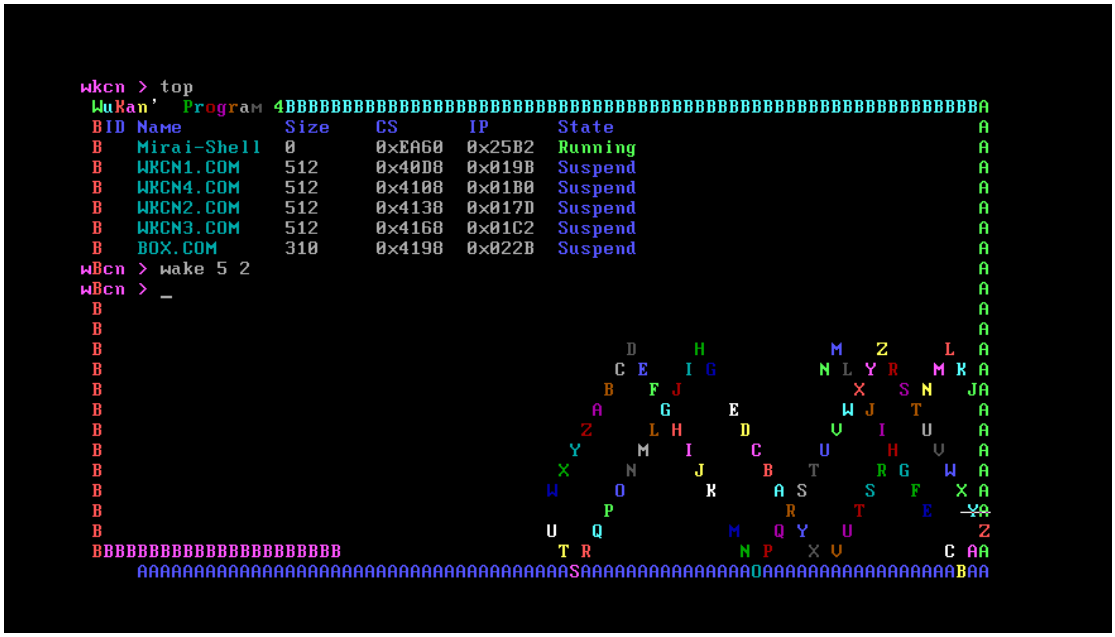


可以看到两个字符在逆时针运动并填充屏幕

在用户程序模式中按键， 右下角显示 OUCH!OUCH!



在 Shell 模式 3 下， 使用 wake xxx 唤醒一些进程（可以批量唤醒，可以看到一些用户程序和 Shell 同步运行， 使用 top 命令可以看见进程状态）



使用 kill, wake, suspend 可以杀死、唤醒、挂起某个进程， 这里不作展示。

top 命令：

可以看见进程状态

在 Shell 模式下输入 inttest，可以运行 inttest.com，依次调用 33h，34h，35h，36h 号中断， 将会同步运行四个用户程序。

输入 int 33h，可以运行 33h 号中断。

以下为版本 2 新增内容：

23 号中断

```
MiraiOS 0.2 #371
You can input 'help' to get more info
wkcn > int 23h
Execute interrupt 0x23
I'm an interrupt written by C++
wkcn > _
```

端口测试：porttest (使用 22h 号中断)

测试 0 号 IO 端口

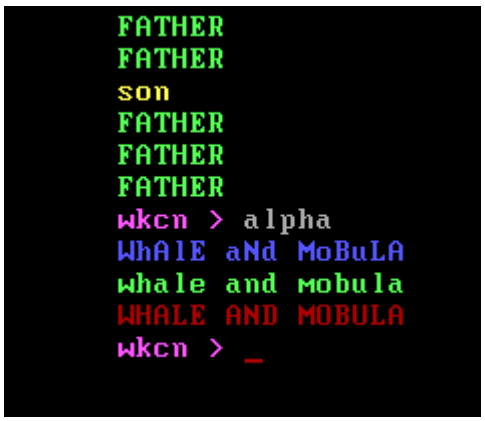
```
MiraiOS 0.2 #371
You can input 'help' to get more
wkcn > int 23h
Execute interrupt 0x23
I'm an interrupt written by C++
wkcn > testport
wkcn > Command not found, Input '
wkcn > porttest
wkcn > GoodJob!
wkcn > _
```

fork()测试： 使用 fork2 程序测试

```
wkcn > porttest
wkcn > GoodJob!
wkcn > fork2
son
son
son
FATHER
son
son
FATHER
son
FATHER
FATHER
son
FATHER
FATHER
FATHER
wkcn > _
```

```
1  #include "include/io.h"
2  #include "include/task.h"
3
4  ostream cout;
5  int main(){
6      int i=0;
7      for(i=0;i<3;i++){
8          uint8_t fpid=fork();
9          if(fpid==0)
10             PrintStr("son", YELLOW);
11         else
12             PrintStr("FATHER",LGREEN);
13         PrintStr(NEWLINE);
14     }
15     return 0;
16
17 }
18
```

24h 号中断：大小写转换测试 alpha



六、小结

这次实现比较简单，这段时间我想起很多年前我就查找过自己制作操作系统的资料，第一次使用 Linux 系统，也是误以为操作系统可以像动态语言一样轻易地修改(我学的第一个编程语言是动态语言 Ruby，那时候还没有动态语言的概念).

我觉得这 ,段时间我的进步很大，了解了很多东西，有时候会连续花很长时间做操作系统实验，可能是因为觉得做出的东西很有成就感，但是有时遇到莫名其妙的问题，比如无缘无故崩溃，使用 Bochs 反编译也很难找出问题. 我在第二次实验中实现了自定义中断，第三次实验中用时钟中断实现了分时系统. 本来以为对中断的调用理解得挺好了，但是当使用手动 push 中断处理程序的段地址，偏移量后，使用远程返回 retf，出现了问题，只能使用 call far 解决.

关于键盘中断，我发现当修改 09h 而不使用原中断处理程序时，

会导致 16h 读取字符的功能失效. 思考原因, 可能是当键盘发生 09h 中断时, 将扫描码存入某个缓冲区； 当调用 16h 中断时, 会从这个缓冲区取扫描码.

我尝试过自行编写 16h, 发现通过扫描码获取 Ascii 码是很困难的, 而且 16h 有很多个功能, 实现起来很困难, 为此保留了 09h 的原有中断处理程序.

我尝试让 C++函数作为中断处理程序, 使用一个汇编程序对 C++函数进行包装(由于中断处理程序使用 iret 返回, 比普通函数 retf 返回多了一个标志寄存器值弹出), 但是失败了, 现在还没研究具体原因, 初步怀疑是段寄存器被修改了.

这次比较自豪的事为: 实现了 top 指令, 可以对单个进程进行管理, 使用了 C++程序管理 PCB.

我查看了我的 Github 关于操作系统实验提交次数, 差不多达 100 多次了, 代码增加量也从实验 3 时的 6000 左右到现在的 18 万, (每次实验我都会复制上一个实验的代码), 但也说明我付出了很多.

以下为版本 2 新增内容：

在版本 2 中，我尝试做得更好，实现了 fork()函数和进程通信。对于我来说，进程通信的实现问题不大，但是 fork()函数还是很难实现的，我想了很久才想出使用现在的解决方式。

我也尝试实现内存管理，但失败了。我尝试了使用 C++编写的中断处理程序运行内存管理函数，跳转位置正确，但最终进入了一个 FXXX 段，不断循环，原因不明。也遇到了一些必须打印\r 字符才能正常的

问题。然后我尝试在指定内存段中放置内存管理表，但也失败了(运行多次 ls 会崩溃)。我尝试改回原来的实现方式，还是会崩溃。无奈之下，只能从版本 448 退回 371. 我的计划为先实现内存管理，再尝试进入保护模式或者实现图形化界面。

附 Github 提交记录：