操作系统原理实验

实验三

用 C 和汇编实现操作系统内核

姓名： 吴侃

学号： 14348134

班别： 2014 级计算机系一班

日期： 2016.03.22 – 2016.03.30

目录

零、特色先览

这次设计的操作系统有很大的亮点：

该系统为**分时系统**，同时支持**批处理**

在内存允许下，该分时系统支持**动态创建进程**

可以**根据进程的 ID 值杀死进程**，也可以一次**杀死全部进程**

有良好的控制台界面，方便的命令行，用户界面切换方式

支持**检查指令**，判断正确性

使用了 **g++**和 **nasm** 交叉编译，运行的环境为 **16 进制实模式**

使用了 C++**内嵌汇编(MASM)**

完全自主编写，支持函数互调

一、实验目的

将实验二的原型操作系统分离为引导程序和 MYOS 内核，由引导程序加载内核，用 C 和汇编实现操作系统内核。

二、实验要求

1、扩展内核汇编代码，增加一些有用的输入输出函数，供 C 模块中调用

2、提供用户程序返回内核的一种解决方案

3、在内核的 C 模块中实现增加批处理能力

在磁盘上建立一个表，记录用户程序的存储安排

可以在控制台命令查到用户程序的信息，如程序名、字节数、在磁盘映像文件中的位置等

设计一种命令，命令中可加载多个用户程序，依次执行，并能在控制台发出命令

在引导系统前，将一组命令存放在磁盘映像中，系统可以解释执行

三、实验环境与工具

3.1 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

注意：由于 qemu, bochs 和 VMware Workstation 模拟的 CPU 速率不同，会导致动画的播放观赏性不好。为了兼容这三个平台，将系统的时钟调到了每秒 20 次中断，并且调整了各个用户程序的刷新频率。

在编写系统的过程中，让动画效果适应 qemu 和 Bochs

在后期编写中，将动画效果适应 VMware Workstation，因此在 qemu 和 bochs 上会反映很慢。

会在之后的实验中，用系统或用户时钟解决这个问题。

3.2 使用工具：

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

合并文件：　dd

四、实验方案

4.1 C++与 nasm 交叉编译

nasm 可以将汇编代码编译为 .o 目标文件

g++也可以将 C++代码编译为 .o 目标文件

（ 这里不使用 C 语言，我的理解是，从编译原理的角度讲，当 C++代码不使用 C++与 C 想比较的特性时，编译出的指令效率和 C 语言是一样的，并且 C++的编写比 C 更方便 ）

生成 .o 目标文件后，将它们链接(这里使用 ld)，生成 .bin 文件，这个文件为不包含头区域的二进制纯指令。

使用 ld 链接时，可以选择参数-Ttext offset，在声明生成的代码将会放在什么区域，类似汇编中的 org 声明。

4.2 互调函数/例程:

要使 nasm 的例程能被 C++调用，需要以下步骤:

以本系统的杀死进程函数为例：

1、在 nasm 汇编代码的该例程前声明 global 函数名，即 global KillProg

2、在 c++代码中声明 extern "C" void KillProg(osi);

(这里的 osi 即类型 uint16_t，之所以起别名是为了将来把程序扩展到 32 位)

C++会将从最后一个参数到第一个参数依次压栈，而 nasm 中的 ax 为函数的返回值。

4.3 nasm 调用 C++函数

在 nasm 汇编代码前声明 extern 函数名

然后使用 call 函数名 即可

4.4 注意事项

g++和 nasm 中的函数互调不用考虑下划线_。

由于 g++编译出的文件不是真正的 16 位指令，难免会产生不兼容的形象。

有以下解决方法:

1、nasm 汇编编写为 C++调用的例程时，返回时用 o32 ret

2、C++的函数带参数时，在前面声明__attribute__((regparm(x)))

其中 x 为参数个数(不包含带默认参数的参数)，这段话告诉编译器，使用 x 个寄存器存储参数，这样可以避免压栈大小不一致带来的问题。

4.5 共享变量

nasm 汇编中的变量，声明 global 变量名后，在 C++中写 extern "C" 变量类型 变量名 即可共享。

4.6 引导

本系统的 loader 为主引导程序，该程序运行后，将内核从软盘写入内存的 0:0x7e00 处,

并跳转到该位置执行内核。

内核是由 nasm 代码和 c++代码混合而成，首先进行基本的寄存

器设值，中断的声明，

然后跳转到 C++编写的 Shell 中。

4.7 分时系统

分时系统是本次实验的最大亮点，它的分时功能由内核 kernel 的汇编胡分实现，拥有一个可扩展的 PCB 表，在内存的允许下可以创建任意多的进程。(由于技术原因，暂时限定最大进程数)

同一个程序可以创建多个进程。

核心描述:

这里使用了 08H 时钟中断，每次时钟中断发生时，

a. 屏蔽中断，

b. 计算出当前进程的 PCB 表位置，将所有寄存器的值保存到该 PCB 中(PCB 为内存的一段区域)

c. 进入调度系统，一般方式为切换到下一进程

d. 恢复新的进程的原寄存器值

f. 取消屏蔽中断

g. 中断返回

这里用到了很巧妙的方法，当中断发生时，flags，cs，ip 依次压栈，在第 d 步时，将栈中的这三个值更换为新进程的对应值，然后中断返回。

4.8 创建进程

内部有一个 ID 计数器: ProgressIDAssigner

通过其计数，可以确定不同的程序的内存分配段地址

将程序写入内存后，修改 PCB 表，进程数加一

之后实验会通过标记来判断空闲内存

4.9 删除进程

用 PCB 表中最后一个进程的数据覆盖要删除的进程

进程数减一

4.10 Shell

使用了屏幕输出，光标控制，键盘输入等中断

滚屏效果:

主要使用了 10h 号中断的 0Eh 功能，可以实现滚屏效果，而这个中断打印的字是无色的，

因此使用 10h 号中断.获取光标的位置，计算在 B800 内存区域的对应位置，用彩色字符替换。

C++实现了一个叫 buf 的缓冲区，用来记录指令的输入，但输入

回车时，调用 Execute 函数进行指令分析，然后执行。

4.11 批处理

有一个 batchList 列表，它可以存储要执行的程序的序号，依次执行。

4.12 返回 Shell

由于是分时系统，实现的原理为一个 Shell 进程和多个用户进程，用户进程受 Shell 管理。

有两种方式返回 Shell

1。当 Shell 判断到 Esc 键按下时，从用户进程界面切换到 Shell。

2。 当 Shell 判断 Ctrl+Z 按下时，将所有用户进程杀死，返回 Shell。

五、实验操作

可用指令：

指令名称	功能
r	回到用户进程界面
ls	列出所有用户程序
cls	清屏
top	显示用户程序状态
kill	杀死一个用户进程，如 kill 3
killall	杀死所有用户进程
uname	显示操作系统信息

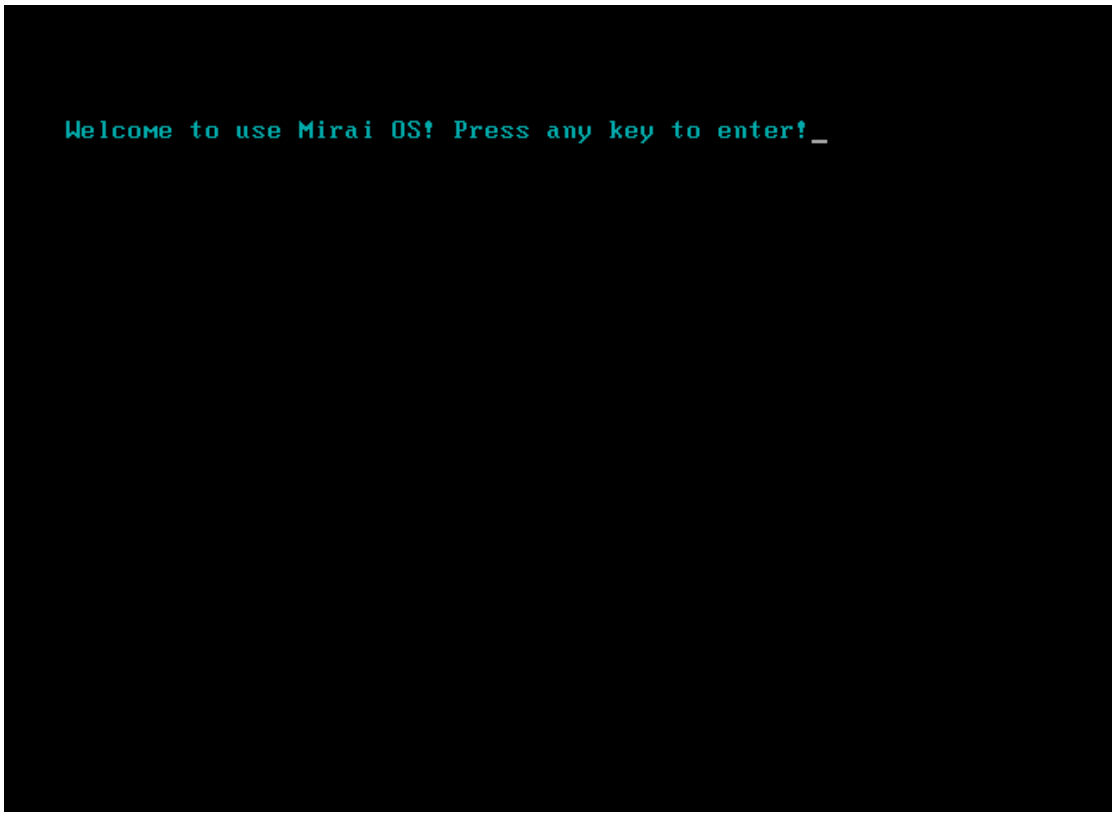
特殊按键：

ESC: 返回 Shell 但是不杀死进程
Ctrl+Z: 返回 Shell 并且杀死所有进程

目前程序的编号为 1~5

输入单个数字， 运行某个程序； 输入一串数字（无间隔）， 批处理这批程序。

加载界面：



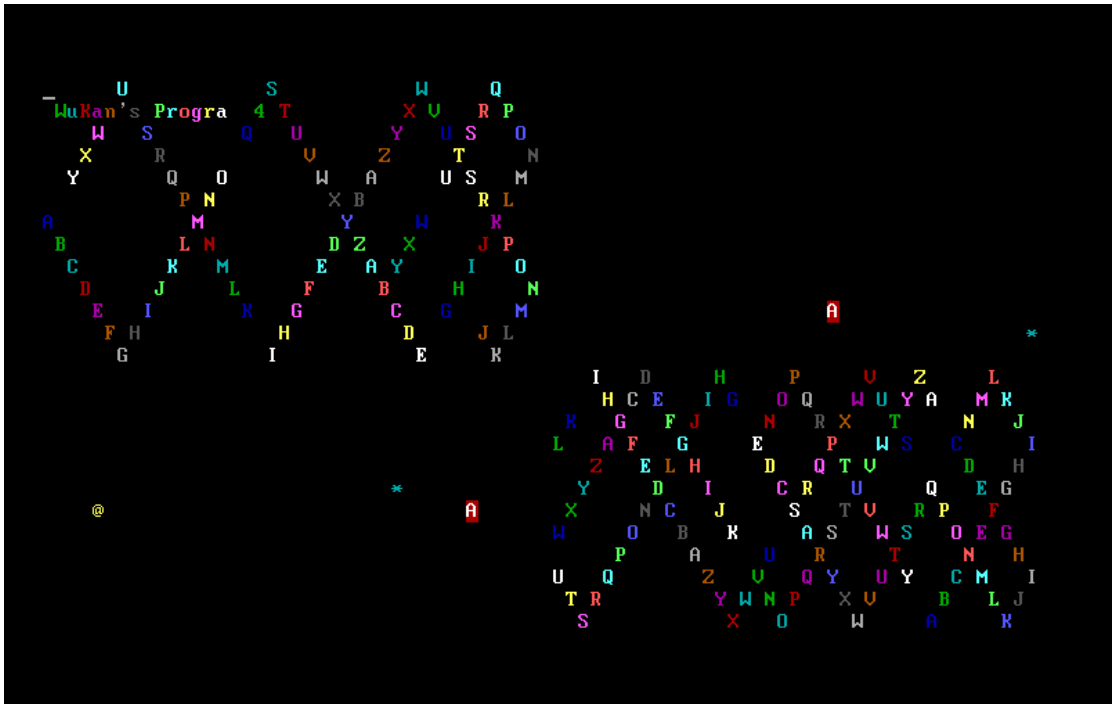
Shell 界面：输入了一些指令

```
MiraiOS 0.1
You can input 'help' to get more info
wkcn > help
Input a 1~5 for parallel running. A stream nums for serial running
Commands:
r         Go to look user processes
ls        list all programs
cls       Clear Screen
top       View all running processes
kill      Kill a process, ex: kill 3
killall   Kill all Processes
uname     Show os info
Keys:
Esc       Back to Shell but not kill processes
Ctrl+Z    Back to Shell and kill all processes
wkcn > uname
wkcn > MiraiOS 0.1
wkcn > ls
Name Size  Pos   Description
1    512    /    Quad 1 45-angle char
2    512    /    Quad 2 45-angle char
3    512    /    Quad 3 45-angle char
4    512    /    Quad 4 45-angle char
5    512    /    Print My Name
wkcn > _
```

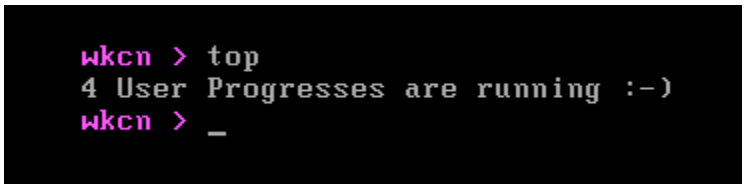
输入 1，按 ESC ;输入 2，按 ESC ;输入 3，按 ESC ;输入 4，按 ESC ；

在 Shell 界面输入 r，回车。(以上输入数字后需按回车)

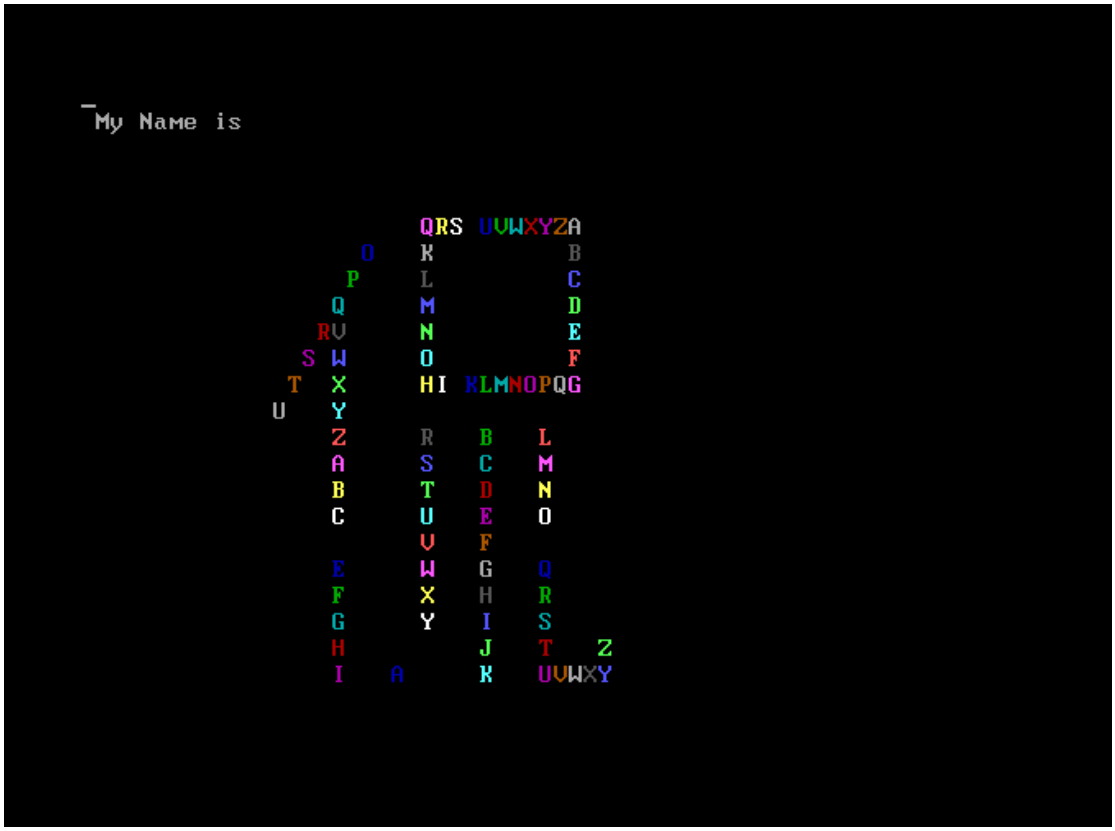
可看到四个程序在并行执行：



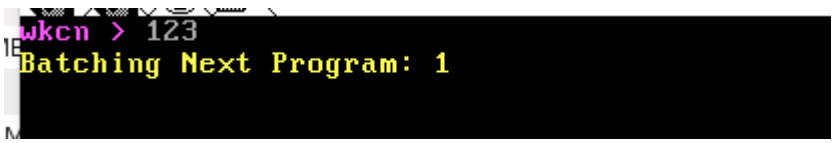
Ctrl + Z 退出， 输入 top 查看用户进程状态：



输入 5， 回车， 打印我的名字：

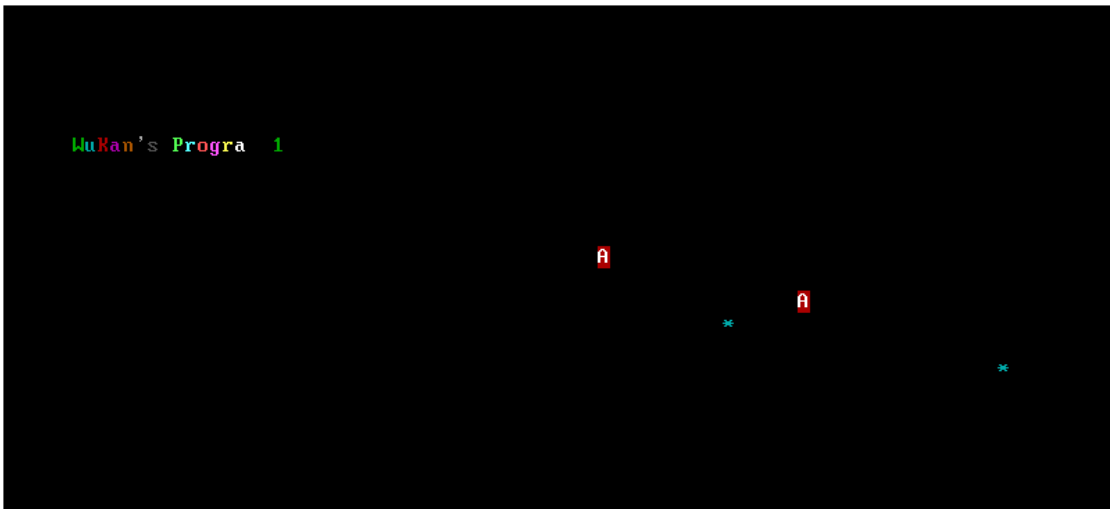


Ctrl+Z 退出， 输入 123, 进行批处理



当用户进程执行时，可按 Esc 保留该进程到下一步；或者 Ctrl+Z，运行下一个进程。

按 Ctrl+Z 或输入 killall 杀死所有进程后，输入 1 回车，ESC，再输入 1 回车，可看到程序 1 在两个不同的进程中执行。



六 小结

这次实验，我耗的精力巨大，根据我 Github 上**私有库**的提交记录，这次实验的提交次数高达 30 次，连续提交天数 9 天。 代码增加了 3000 多行。

我的目标是实现一个分时系统，在这段时间中，虽然我也学习了保护模式的编写，但考虑到实现难度和可用时间，我选择了首先实现分时系统。

我一早就想实现分时系统了，无奈参考资料很少， 大部分分时系统的实现都是基于高级语言的，我也想过用高级语言实现分时系统，

但觉得透明性太多，不知道内部寄存器如何被修改。

实现分时系统，我首先遇到的问题是，如何使用时钟中断?

我查了大量资料，可是很多资料是从硬件层面上介绍硬件接口，连一个简单的范例都很难找。

我测试时钟中断是否有效，是使用极其简单的方式，写一个简单的程序看字符是否会被修改。

学习使用时钟中断的过程中，我也在学习 C 语言和 nasm 的交叉编译，其中掉进了很多坑。从中我的最大收获是了解了 qemu, bochs 是如何使用的, Makefile 是如何编写的。

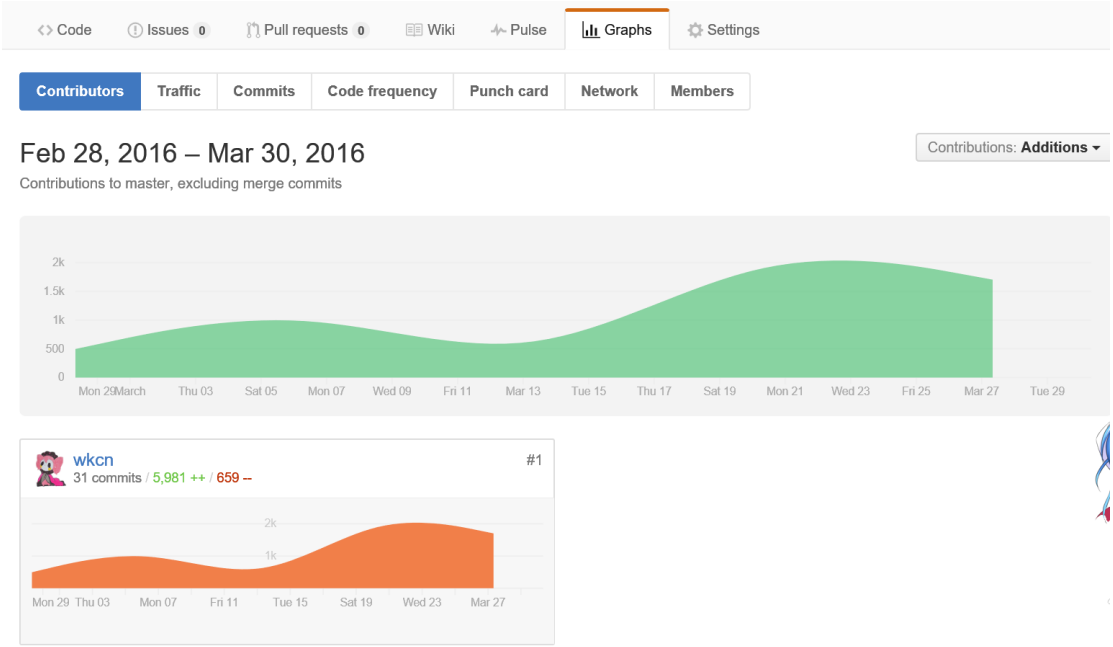
学习交叉编译，花了差不多一周的时间，遇到了比如找不到_start 入口的链接问题， 压栈的不兼容性，指令占用过多扇区的处理。

分时系统的实现也是从很简单的程序开始，首先测试只有一个进程时，该进程是否能运行正常。 最好笑的是，我发现了 nasm 编译的一个 bug，它有时会少编译一条指令，我的解决方式是在缺失指令前加 nop。

这次光调试，用 bochs 的次数就有几百次，期间还多次使用反汇编查看指令。

这次实验我付出了很多，同时也收获很多。

附 Github 私有库提交记录：



参考文献：

1. <x86 PC 汇编语言，设计与接口> - 作者：（美国）马兹迪（Muhammad Ali Mazidi） （美国）考西（Danny Causey） （美国）马兹迪（Janice Gillispie Mazidi） 译者：高升 合著者：王筱珍
2. <x86 汇编语言-从实模式到保护模式> - 李忠 王晓波 余 洁 著

3. <使用 GNU Binutils、 GCC 和 NASM 编译操作系统> - 蔡日骏
4. Leasunhy OS
5. Ling OS
6. Dev OS
7. 多个内嵌汇编教程