## 项目概述和开发环境配置

这是一篇阐述如何在基于Intel x86架构的IBM PC机及其兼容计算机上构建一个简单

的操作系统内核的本科毕业设计论文。我将带领大家一起来探索x86 CPU的保护模式下操作系统内核的编写方法，一起感受一次完整的探索过程。虽然这个小内核和一个具有商业价值的操作系统内核相较而言依旧相差甚远。但是通过这样的探索，相信我们能充分的理解x86保护模式的运行方式和操作系统的基本原理，而这恰恰是传统的通过理论教学和阅读书籍的方式难以获得的深刻体验。

言归正传，开始我的介绍。工欲善其事，必先利其器，我先来阐述下开发环境和相关的工具配置。

### 工作环境

Windows 和Linux 之争由来已久，我不想在这篇论文里针对这个问题再费口舌，我

的工作环境选择Linux。使用Linux的原因很简单，这里有可以自由使用的一系列的开源软件能很好的协助我们的开发和调试工作，而在Windows下缺乏相应的免费工具。虽然我的构建环境使用的是Ubuntu-16.04-i386-Desktop，但是这不影响大家在对项目验证时对于Linux发行版的选择，因为使用的命令基本上都是相同的。

### 开发语言

### 开发工具

接着是选择开发使用的工具了，这个我简单罗列出来。首先C语言的编译器肯定使用

gcc，链接器自然也就是ld了。同时大项目自然也少不了GNU make这个构建工具。至于

汇编编译器我们选择nasm这个开源免费的编译器，以便使用大多数读者习惯的Intel风格的汇编语法。不过考虑到需要在一些C语言代码中内联汇编指令，而gcc使用的是AT&T风格的汇编语法，所以我还是稍稍学习掌握了一部分的AT&T风格的语法。这些就是开发使用的基本工具了，其他的工具我会在使用的时候再介绍。

我们写用户级别程序自然可以直接运行，现在是要写一个操作系统内核。我们在哪里运行它？我们可以使用虚拟机。不过我们这次使用的不是大多数人熟悉的Vmware或者Virtual Box，而是一款叫做bochs的虚拟机。为什么呢？因为有调试的需要。我们需要一个能调试其上运行着的操作系统的虚拟机，而bochs是个不错的选择。选择另一款叫做qemu的虚拟机也支持调试，但本着简单易用的原则，这里不选择qemu。

bochs的安装方法很简单，以Ubuntu为例，只需执行以下命令即可。

sudo apt-get install bochs bochs-x

## 总体设计

MOS目前支持的硬件环境是基于Intel 80386以上的计算机系统。更多的硬件相关内容（比如保护模式等）将随着实现MOS的过程逐渐展开介绍。那我们准备如何一步一步实现MOS呢？按照一个操作系统的开发过程，我们可以有如下的开发步骤：

1. bootloader：它将负责从启动设备中寻找ELF格式的内核可执行文件kernel.bin，并将其读入内存，随后将机器跳转至kernel.bin处开始执行。
2. 内核雏形：
3. 进程管理：
4. 输入/输出系统：
5. 进程间通信：
6. 文件系统：理解文件系统的具体实现，与进程管理和内存管理等的关系，缓存对操作系统IO访问的性能改进，虚拟文件系统（VFS）、buffer cache和disk driver之间的关系。
7. 内存管理：
8. C运行时库与简易Shell：

其中每个开发步骤都是建立在上一个步骤之上的，就像搭积木，从一个一个小木块，最终搭出来一个小房子。在搭房子的过程中，完成从理解操作系统原理到实践操作系统设计与实现的探索过程。这个房子最终的建筑架构和建设进度如下图所示：

## BootLoader

一个操作系统从开机到运行，大概是“引导→加载内核进入内存→跳入保护模式→开始执行内核”这个历程。在内核开始执行之前，bootloader必须已将内核加载入内存，并且已将保护模式准备妥当。我们知道，每个启动设备的引导扇区只有区区的512字节大小，如果bootloader的工作全部交给引导扇区来做，很有可能不够用。我的解决方案是：被引导扇区加载进内存的不一定就是操作系统内核，可以把这些工作交给一个单独的模块loader，而小小的boot模块仅负责加载loader并交由它继续引导，loader的大小随意，可以很灵活地执行其他任务。

启动设备可以是软盘、硬盘、网络等等，具体选择何种设备启动由CMOS设置决定的。由于我们可以依赖的调试工具匮乏、经验尚浅，为了方便调试，也为了将loader和kernel的写入启动盘简单易行，这里不妨以MSDOS FAT12格式的软盘镜像作为启动设备。

软盘启动的过程如下：

1. BIOS将引导扇区读入到内存0000：7c00处；
2. 跳转到0000：7c00处开始执行引导代码；
3. boot从软盘中找到loader，并将其读入内存；
4. 跳转到loader继续执行引导代码；
5. loader从软盘中找到kernel，并将其读入内存；
6. 跳转到kernel开始执行，到此软盘引导系统启动结束；

下面我分别从boot和loader详细介绍。

### Boot

我们想让软盘以FAT12格式被MSDOS识别，必须在引导扇区中添加BPB等头信息，如下：

BS\_OEMName DB 'Menguozi' ; OEM String, 必须 8 个字节

BPB\_BytsPerSec DW 512 ; 每扇区字节数

BPB\_SecPerClus DB 1 ; 每簇多少扇区

BPB\_RsvdSecCnt DW 1 ; Boot 记录占用多少扇区

BPB\_NumFATs DB 2 ; 共有多少 FAT 表

BPB\_RootEntCnt DW 224 ; 根目录文件数最大值

BPB\_TotSec16 DW 2880 ; 逻辑扇区总数

BPB\_Media DB 0xF0 ; 媒体描述符

BPB\_FATSz16 DW 9 ; 每FAT扇区数

BPB\_SecPerTrk DW 18 ; 每磁道扇区数

BPB\_NumHeads DW 2 ; 磁头数(面数)

BPB\_HiddSec DD 0 ; 隐藏扇区数

BPB\_TotSec32 DD 0 ; 如果 wTotalSectorCount 是 0 由这个值记录扇区数

BS\_DrvNum DB 0 ; 中断 13 的驱动器号

BS\_Reserved1 DB 0 ; 未使用

BS\_BootSig DB 29h ; 扩展引导标记 (29h)

BS\_VolID DD 0 ; 卷序列号

BS\_VolLab DB 'Menguozi’OS'; 卷标, 必须 11 个字节

BS\_FileSysType DB 'FAT12 ' ; 文件系统类型, 必须 8个字节

把这样生成的boot.bin写入软盘引导扇区后，现在的软盘已经能够被DOS以及Linux识别了，我们可以方便地往其中添加loader.bin和kernel.bin了。

#### 加载loader入内存

要加载loader入内存，必须要使用BIOS中断int 13h来读软盘。它的用法如下表所示。



7

#### 将控制权交由loader

loader已被我们成功载入内存，下面的一个跳转将结束boot sector的使命，开始执行loader部分代码。

jmp LOADER\_SEG:LOADER\_OFF ; 这一句正式跳转到已加载到内存中的 LOADER.BIN 的开始处

; 开始执行 LOADER.BIN 的代码

; Boot Sector 的使命到此结束

至此，我们的boot部分已封棺盖土了。

### Loader

#### 加载kernel入内存

#### 将控制权交由kernel

## 内核雏形

上一章我说到，为了加载ELF格式的内核进内存，我们必须研究一下这种格式。

### GCC内联汇编

### 从Loader到内核

### 扩充内核

## 进程管理

## 输入/输出系统

我们刚刚实现了简单的进程，不过现在我不打算把它做得更加完善，比如优化进程调度算法、实现进程间通信机制等。因为这些工作在理论和实践上都没有尽头，而有些也是目前无法实现的，因为进程必须与输入/输出系统、内存管理等其他模块一起运作。正如我所遵循的设计理念——过早的优化是万恶之源，简单的进程更有利于我们的思考和实现。

那么接下来，我们先来实现简单的输入/输出系统。

### 键盘

截止到目前，我们的MOS一旦启动就像推倒了第一块多米诺骨牌，不再为我们所控制，只能等到最后的结果出现。但正如眼下人们看视频的方式，从电视转向互联网一样，我们需要不时的发个弹幕，我们的MOS也需要和用户交互。这里的交互也即输入/输出系统，我们先考虑输入系统，典型的输入设备当然非键盘莫属。

#### AT、PS/2键盘

当前最流行的三种键盘分别是AT键盘、PS/2键盘和USB键盘。在本项目中，我们不讨论USB键盘，只涉及AT和PS/2两种。PS/2键盘只是在于AT键盘的基础之上做了一点扩展，大多数情况下，比如我们MOS最简单的使用情况下，我们可它们认为一样的。

#### 键盘敲击过程分析

在键盘中存在两枚芯片——键盘编码器和键盘控制器，它们通常是Intel 8048、8042以及兼容芯片。键盘编码器8048用来监视键盘的输入，并把数据传送给计算机。键盘控制器8042用来接收和解码来自键盘的数据，并与8259A以及软件等进行通信。

每敲击一次键盘有两种含义：动作和内容。动作分成三类：按下、保持按住状态以及弹开；内容反映的是键盘上具体的某一按键。

敲击键盘产生的编码称为扫描码，分为Make Code和Break Code两种。当按下或者保持按住一个键时，将会产生Make Code；当弹起一个键时，将会产生Break Code。每个键都对应一个Make Code和Break Code，除Pasue键。

扫描码总共有三套，分别是Scan code set 1、Scan code set 2、Scan code set 3。现在的键盘都默认支持Scan code set 2，但是为了XT键盘的程序兼容性问题，最终又转化成了Scan code set 1。

Scan code set 1扫描码

 当8048检测到一个键动作后，会产生相应的扫描码发送给8042，8042将把它转换成相应的Scan code set 1扫描码，并放置在输入缓冲区中，然后由8042通知8259A产生IRQ1键盘中断。若此时键盘又有新键被按下，8042不予接收，直到缓冲区被清空，才重新开始接收新的扫描码。

为了从缓冲区读取扫描码，也即操作8042的输入/输入缓冲区，我们需要了解一下8042的寄存器。

8042的寄存器

 通过in和out指令可以用来进行输入/输出缓冲区的相应读取操作，在keyboard\_handler中添加下面一句就可以读取扫描码了：

in\_byte(0x60);

#### 扫描码

现在我们已经能够获取到扫描码了，接下来我们就要运用Scan code set 1来解析扫描码。Key值和Make Code以及Break Code没用对应的函数关系，既然这样，我们可以以扫描码建立一个数组，对应的元素即为相应的字符。

在keymap.h中建立如下数组，其中每三个值一组，分别是单独按下某键、Shift+某键和有0xE0前缀的扫描码对应的字符。特殊的Esc、Enter等被定义成宏。

扫描码解析数组（include/keyboard.h）

u32 keymap[NR\_SCAN\_CODES \* MAP\_COLS] = {

/\* scan-code !Shift Shift E0 XX \*/

/\* ==================================================================== \*/

/\* 0x00 - none \*/ 0, 0, 0,

/\* 0x01 - ESC \*/ ESC, ESC, 0,

/\* 0x02 - '1' \*/ '1', '!', 0,

/\* 0x03 - '2' \*/ '2', '@', 0,

/\* 0x04 - '3' \*/ '3', '#', 0,

/\* 0x05 - '4' \*/ '4', '$', 0,

/\* 0x06 - '5' \*/ '5', '%', 0,

/\* 0x07 - '6' \*/ '6', '^', 0,

/\* 0x08 - '7' \*/ '7', '&', 0,

/\* 0x09 - '8' \*/ '8', '\*', 0,

/\* 0x0A - '9' \*/ '9', '(', 0,

/\* 0x0B - '0' \*/ '0', ')', 0,

/\* 0x0C - '-' \*/ '-', '\_', 0,

/\* 0x0D - '=' \*/ '=', '+', 0,

/\* 0x0E - BS \*/ BACKSPACE, BACKSPACE, 0,

/\* 0x0F - TAB \*/ TAB, TAB, 0,

/\* 0x10 - 'q' \*/ 'q', 'Q', 0,

/\* 0x11 - 'w' \*/ 'w', 'W', 0,

/\* 0x12 - 'e' \*/ 'e', 'E', 0,

/\* 0x13 - 'r' \*/ 'r', 'R', 0,

/\* 0x14 - 't' \*/ 't', 'T', 0,

/\* 0x15 - 'y' \*/ 'y', 'Y', 0,

/\* 0x16 - 'u' \*/ 'u', 'U', 0,

/\* 0x17 - 'i' \*/ 'i', 'I', 0,

/\* 0x18 - 'o' \*/ 'o', 'O', 0,

/\* 0x19 - 'p' \*/ 'p', 'P', 0,

/\* 0x1A - '[' \*/ '[', '{', 0,

/\* 0x1B - ']' \*/ ']', '}', 0,

/\* 0x1C - CR/LF \*/ ENTER, ENTER, PAD\_ENTER,

/\* 0x1D - l. Ctrl \*/ CTRL\_L, CTRL\_L, CTRL\_R,

/\* 0x1E - 'a' \*/ 'a', 'A', 0,

/\* 0x1F - 's' \*/ 's', 'S', 0,

/\* 0x20 - 'd' \*/ 'd', 'D', 0,

/\* 0x21 - 'f' \*/ 'f', 'F', 0,

/\* 0x22 - 'g' \*/ 'g', 'G', 0,

/\* 0x23 - 'h' \*/ 'h', 'H', 0,

/\* 0x24 - 'j' \*/ 'j', 'J', 0,

/\* 0x25 - 'k' \*/ 'k', 'K', 0,

/\* 0x26 - 'l' \*/ 'l', 'L', 0,

/\* 0x27 - ';' \*/ ';', ':', 0,

/\* 0x28 - '\'' \*/ '\'', '"', 0,

/\* 0x29 - '`' \*/ '`', '~', 0,

/\* 0x2A - l. SHIFT \*/ SHIFT\_L, SHIFT\_L, 0,

/\* 0x2B - '\' \*/ '\\', '|', 0,

/\* 0x2C - 'z' \*/ 'z', 'Z', 0,

/\* 0x2D - 'x' \*/ 'x', 'X', 0,

/\* 0x2E - 'c' \*/ 'c', 'C', 0,

/\* 0x2F - 'v' \*/ 'v', 'V', 0,

/\* 0x30 - 'b' \*/ 'b', 'B', 0,

/\* 0x31 - 'n' \*/ 'n', 'N', 0,

/\* 0x32 - 'm' \*/ 'm', 'M', 0,

/\* 0x33 - ',' \*/ ',', '<', 0,

/\* 0x34 - '.' \*/ '.', '>', 0,

/\* 0x35 - '/' \*/ '/', '?', PAD\_SLASH,

/\* 0x36 - r. SHIFT \*/ SHIFT\_R, SHIFT\_R, 0,

/\* 0x37 - '\*' \*/ '\*', '\*', 0,

/\* 0x38 - ALT \*/ ALT\_L, ALT\_L, ALT\_R,

/\* 0x39 - ' ' \*/ ' ', ' ', 0,

/\* 0x3A - CapsLock \*/ CAPS\_LOCK, CAPS\_LOCK, 0,

/\* 0x3B - F1 \*/ F1, F1, 0,

/\* 0x3C - F2 \*/ F2, F2, 0,

/\* 0x3D - F3 \*/ F3, F3, 0,

/\* 0x3E - F4 \*/ F4, F4, 0,

/\* 0x3F - F5 \*/ F5, F5, 0,

/\* 0x40 - F6 \*/ F6, F6, 0,

/\* 0x41 - F7 \*/ F7, F7, 0,

/\* 0x42 - F8 \*/ F8, F8, 0,

/\* 0x43 - F9 \*/ F9, F9, 0,

/\* 0x44 - F10 \*/ F10, F10, 0,

/\* 0x45 - NumLock \*/ NUM\_LOCK, NUM\_LOCK, 0,

/\* 0x46 - ScrLock \*/ SCROLL\_LOCK, SCROLL\_LOCK, 0,

/\* 0x47 - Home \*/ PAD\_HOME, '7', HOME,

/\* 0x48 - CurUp \*/ PAD\_UP, '8', UP,

/\* 0x49 - PgUp \*/ PAD\_PAGEUP, '9', PAGEUP,

/\* 0x4A - '-' \*/ PAD\_MINUS, '-', 0,

/\* 0x4B - Left \*/ PAD\_LEFT, '4', LEFT,

/\* 0x4C - MID \*/ PAD\_MID, '5', 0,

/\* 0x4D - Right \*/ PAD\_RIGHT, '6', RIGHT,

/\* 0x4E - '+' \*/ PAD\_PLUS, '+', 0,

/\* 0x4F - End \*/ PAD\_END, '1', END,

/\* 0x50 - Down \*/ PAD\_DOWN, '2', DOWN,

/\* 0x51 - PgDown \*/ PAD\_PAGEDOWN, '3', PAGEDOWN,

/\* 0x52 - Insert \*/ PAD\_INS, '0', INSERT,

/\* 0x53 - Delete \*/ PAD\_DOT, '.', DELETE,

/\* 0x54 - Enter \*/ 0, 0, 0,

/\* 0x55 - ??? \*/ 0, 0, 0,

/\* 0x56 - ??? \*/ 0, 0, 0,

/\* 0x57 - F11 \*/ F11, F11, 0,

/\* 0x58 - F12 \*/ F12, F12, 0,

/\* 0x59 - ??? \*/ 0, 0, 0,

/\* 0x5A - ??? \*/ 0, 0, 0,

/\* 0x5B - ??? \*/ 0, 0, GUI\_L,

/\* 0x5C - ??? \*/ 0, 0, GUI\_R,

/\* 0x5D - ??? \*/ 0, 0, APPS,

/\* 0x5E - ??? \*/ 0, 0, 0,

/\* 0x5F - ??? \*/ 0, 0, 0,

/\* 0x60 - ??? \*/ 0, 0, 0,

/\* 0x61 - ??? \*/ 0, 0, 0,

/\* 0x62 - ??? \*/ 0, 0, 0,

/\* 0x63 - ??? \*/ 0, 0, 0,

/\* 0x64 - ??? \*/ 0, 0, 0,

/\* 0x65 - ??? \*/ 0, 0, 0,

/\* 0x66 - ??? \*/ 0, 0, 0,

/\* 0x67 - ??? \*/ 0, 0, 0,

/\* 0x68 - ??? \*/ 0, 0, 0,

/\* 0x69 - ??? \*/ 0, 0, 0,

/\* 0x6A - ??? \*/ 0, 0, 0,

/\* 0x6B - ??? \*/ 0, 0, 0,

/\* 0x6C - ??? \*/ 0, 0, 0,

/\* 0x6D - ??? \*/ 0, 0, 0,

/\* 0x6E - ??? \*/ 0, 0, 0,

/\* 0x6F - ??? \*/ 0, 0, 0,

/\* 0x70 - ??? \*/ 0, 0, 0,

/\* 0x71 - ??? \*/ 0, 0, 0,

/\* 0x72 - ??? \*/ 0, 0, 0,

/\* 0x73 - ??? \*/ 0, 0, 0,

/\* 0x74 - ??? \*/ 0, 0, 0,

/\* 0x75 - ??? \*/ 0, 0, 0,

/\* 0x76 - ??? \*/ 0, 0, 0,

/\* 0x77 - ??? \*/ 0, 0, 0,

/\* 0x78 - ??? \*/ 0, 0, 0,

/\* 0x78 - ??? \*/ 0, 0, 0,

/\* 0x7A - ??? \*/ 0, 0, 0,

/\* 0x7B - ??? \*/ 0, 0, 0,

/\* 0x7C - ??? \*/ 0, 0, 0,

/\* 0x7D - ??? \*/ 0, 0, 0,

/\* 0x7E - ??? \*/ 0, 0, 0,

/\* 0x7F - ??? \*/ 0, 0, 0

};

#### 键盘输入缓冲区

根据我们的扫描码解析数组，利用接收到的扫描码，我们就可以循数组找到对应的字符。但是，其中存在一个问题，之前说过8042的输入缓冲区只有一个字节大小，所以组合按键产生的扫描码产生多次中断。如果当第一次中断接收到0x2A时，我们无法判断是按下Shift又释放，还是组合按下Shift+X。所以当接收到0x2A这类特殊的扫描码时，我们需要先将其保存起来，等待接收到后一个扫描码再行判断。

保存一个字符可以用全局变量来表示，由于组合扫描码的长度不定，借鉴我们的前辈Minix，我们这里建立一个缓冲区，用以放置中断处理程序接收到的扫描码。

键盘缓冲区（include/keyboard.h）

struct kb\_inbuf {

char\* p\_head; /\* 指向缓冲区中下一个空闲位置 \*/

char\* p\_tail; /\* 指向键盘任务应处理的字节 \*/

int count; /\* 缓冲区中共有多少字节 \*/

char buf[KB\_IN\_BYTES]; /\* 缓冲区 \*/

};

#### 解析扫描码

我们完成了从键盘控制器中将扫描码完整地读入了键盘输入缓冲区，接下来我们要进行一些繁琐但是不难的工作——解析扫描码，下面我们来分类分步骤地来实现它。

各种类型的扫描码解析工作，我们都统一放在keyboard\_read()中处理。

##### 处理单个按键

单个按键的扫描码处理十分简单，只要以扫描码为下标，循扫描码解析数组找到对应的字符即可，这里不再赘述。

##### 处理Shift、Alt、Ctrl等组合按键

我的处理方法是：既然按下一个键会产生数个字节的扫描码，我们不妨在一个过程中get\_byte\_from\_kb\_buf()把它们都读取出来。设置一个全局变量key，key值第一次从keymap数组中得到，如果一个组合按键的解析未完成，则key赋值为0，等到下一次keyboard\_read()被执行时再继续处理。总之，我的核心思想是一次组合按键的解析需要多次调用keyboard\_read()完成。

解析扫描码函数（kernel/keyboard.c）

PUBLIC void keyboard\_read(TTY\* tty)

{

u8 scan\_code;

/\*\*

\* 1 : make

\* 0 : break

\*/

int make;

u32 key = 0;

/\*\*

\* 指向keymap[]的某一行

\*/

u32\* keyrow;

while (kb\_in.count > 0) {

code\_with\_E0 = 0;

scan\_code = get\_byte\_from\_kb\_buf();

/\* 开始解析扫描码 \*/

if (scan\_code == 0xE1) {

int i;

u8 pausebreak\_scan\_code[] = {0xE1, 0x1D, 0x45, 0xE1, 0x9D, 0xC5};

int is\_pausebreak = 1;

for (i = 1; i < 6; i++) {

if (get\_byte\_from\_kb\_buf() != pausebreak\_scan\_code[i]) {

is\_pausebreak = 0;

break;

}

}

if (is\_pausebreak) {

key = PAUSEBREAK;

}

}

else if (scan\_code == 0xE0) {

code\_with\_E0 = 1;

scan\_code = get\_byte\_from\_kb\_buf();

/\* PrintScreen被按下\*/

if (scan\_code == 0x2A) {

code\_with\_E0 = 0;

if ((scan\_code = get\_byte\_from\_kb\_buf()) == 0xE0) {

code\_with\_E0 = 1;

if ((scan\_code = get\_byte\_from\_kb\_buf()) == 0x37) {

key = PRINTSCREEN;

make = 1;

}

}

}

/\* PrintScreen被释放\*/

else if (scan\_code == 0xB7) {

code\_with\_E0 = 0;

if ((scan\_code = get\_byte\_from\_kb\_buf()) == 0xE0) {

code\_with\_E0 = 1;

if ((scan\_code = get\_byte\_from\_kb\_buf()) == 0xAA) {

key = PRINTSCREEN;

make = 0;

}

}

}

}

if ((key != PAUSEBREAK) && (key != PRINTSCREEN)) {

int caps;

/\* 首先判断是Make还是Break Code \*/

make = (scan\_code & FLAG\_BREAK ? 0 : 1);

keyrow = &keymap[(scan\_code & 0x7F) \* MAP\_COLS];

column = 0;

caps = shift\_l || shift\_r;

if (caps\_lock &&

keyrow[0] >= 'a' && keyrow[0] <= 'z')

caps = !caps;

if (caps)

column = 1;

if (code\_with\_E0)

column = 2;

key = keyrow[column];

switch(key) {

case SHIFT\_L:

shift\_l = make;

break;

case SHIFT\_R:

shift\_r = make;

break;

case CTRL\_L:

ctrl\_l = make;

break;

case CTRL\_R:

ctrl\_r = make;

break;

case ALT\_L:

alt\_l = make;

break;

case ALT\_R:

alt\_l = make;

break;

case CAPS\_LOCK:

if (make) {

caps\_lock = !caps\_lock;

set\_leds();

}

break;

case NUM\_LOCK:

if (make) {

num\_lock = !num\_lock;

set\_leds();

}

break;

case SCROLL\_LOCK:

if (make) {

scroll\_lock = !scroll\_lock;

set\_leds();

}

break;

default:

break;

}

}

if(make){ /\* 忽略Break Code \*/

int pad = 0;

/\* 首先处理小键盘 \*/

if ((key >= PAD\_SLASH) && (key <= PAD\_9)) {

pad = 1;

switch(key) { /\* '/', '\*', '-', '+',

\* and 'Enter' in num pad

\*/

case PAD\_SLASH:

key = '/';

break;

case PAD\_STAR:

key = '\*';

break;

case PAD\_MINUS:

key = '-';

break;

case PAD\_PLUS:

key = '+';

break;

case PAD\_ENTER:

key = ENTER;

break;

default:

/\* the value of these keys

\* depends on the Numlock

\*/

if (num\_lock) { /\* '0' ~ '9' and '.' in num pad \*/

if (key >= PAD\_0 && key <= PAD\_9)

key = key - PAD\_0 + '0';

else if (key == PAD\_DOT)

key = '.';

}

else{

switch(key) {

case PAD\_HOME:

key = HOME;

break;

case PAD\_END:

key = END;

break;

case PAD\_PAGEUP:

key = PAGEUP;

break;

case PAD\_PAGEDOWN:

key = PAGEDOWN;

break;

case PAD\_INS:

key = INSERT;

break;

case PAD\_UP:

key = UP;

break;

case PAD\_DOWN:

key = DOWN;

break;

case PAD\_LEFT:

key = LEFT;

break;

case PAD\_RIGHT:

key = RIGHT;

break;

case PAD\_DOT:

key = DELETE;

break;

default:

break;

}

}

break;

}

}

key |= shift\_l ? FLAG\_SHIFT\_L : 0;

key |= shift\_r ? FLAG\_SHIFT\_R : 0;

key |= ctrl\_l ? FLAG\_CTRL\_L : 0;

key |= ctrl\_r ? FLAG\_CTRL\_R : 0;

key |= alt\_l ? FLAG\_ALT\_L : 0;

key |= alt\_r ? FLAG\_ALT\_R : 0;

key |= pad ? FLAG\_PAD : 0;

in\_process(tty, key);

}

}

}

#### 键盘中断

之前我提到过，8259A的IRQ1对应的就是键盘中断。那时我们没有为键盘中断指定处理程序，现在我们搞清楚了键盘接收输入的来龙去脉，正是我们施展身手的大好时候。

新建文件keyboard.c，为键盘中断添加一个至简的键盘中断处理程序。

键盘中断处理程序（kernel/keyboard.c）

PUBLIC void keyboard\_handler(int irq)

{

u8 scan\_code = in\_byte(KB\_DATA);

if (kb\_in.count < KB\_IN\_BYTES) {

\*(kb\_in.p\_head) = scan\_code;

kb\_in.p\_head++;

if (kb\_in.p\_head == kb\_in.buf + KB\_IN\_BYTES)

kb\_in.p\_head = kb\_in.buf;

kb\_in.count++;

}

key\_pressed = 1;

}

代码很简洁，这里的keyboard\_handler只负责把接收到的扫描码存入缓冲区；如果缓冲区已满，这里的策略是把新接收到的扫描码直接丢弃。其中kb\_in是我们定义的只在keyboard.c中使用的static变量。

kb\_in这个数据结构使用前需要初始化，我们把初始化的工作放在init\_keyboard()中单独处理。另外，指定键盘中断处理程序和开启键盘中断的工作也一并放在其中。

键盘初始化函数（kernel/keyboard.c）

PUBLIC void init\_keyboard()

{

kb\_in.count = 0;/\*初始化键盘输入缓冲区为空\*/

kb\_in.p\_head = kb\_in.p\_tail = kb\_in.buf;/\*初始化键盘输入缓冲区各指针\*/

shift\_l = shift\_r = 0;

alt\_l = alt\_r = 0;

ctrl\_l = ctrl\_r = 0;

caps\_lock = 0;

num\_lock = 1;

scroll\_lock = 0;

column = 0;

set\_leds();

put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler);/\*设定键盘中断处理程序\*/

enable\_irq(KEYBOARD\_IRQ);/\*开启键盘中断\*/

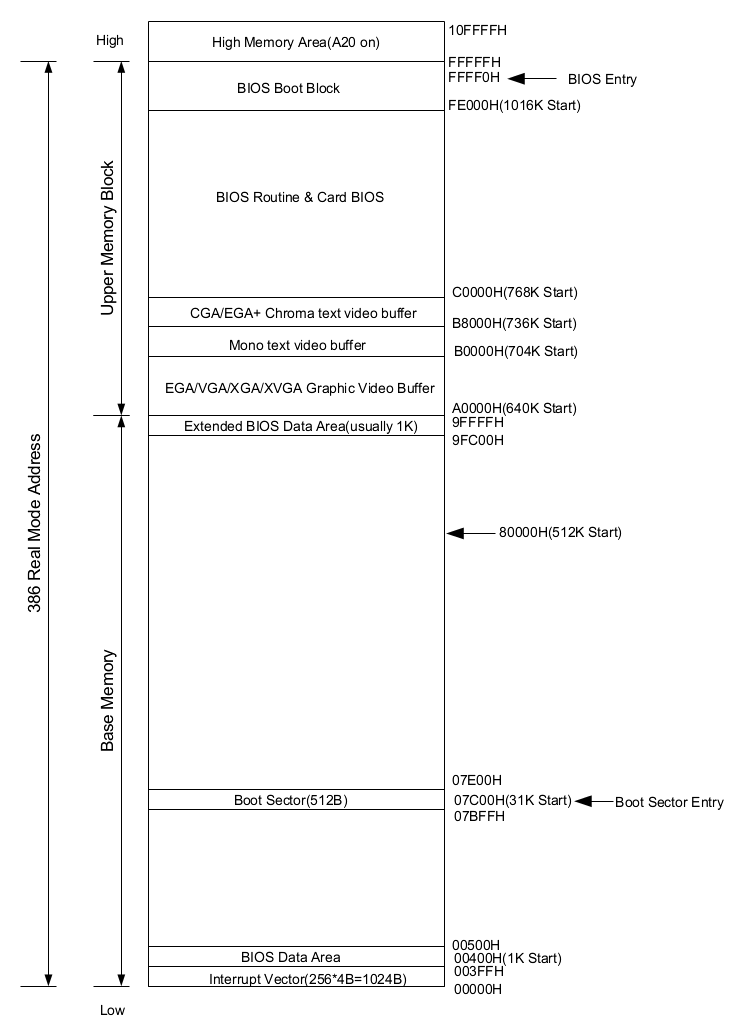
}

### 显示器

这小节里我们描述内核对屏幕输出的控制。

#### 1MB以下的地址空间分布

在之前内核雏形中我们简单的谈过地址空间的概念，并提到4G的地址空间并非全部指向主存储器，而是有部分的地址分给了其他外设。特别地，在地址空间的最低1MB处，有很多地址是属于外部设备的，下图描绘了该处地址映射的分布情况：



在PC上要显示文字，通常需要显示器和显卡这两个硬件设备。一般来说显卡负责提供显示内容，并控制具体的显示模块和状态。显示器的职责是负责将显卡呈递的内容可视化的

显示出来。既然显卡需要控制显示的数据，自然就需要存储这些待显示的内容，所以显卡

就有自己的存储区域。这个存储区域叫做显示存储器（Video RAM，VRAM），简称显存。

当然，访问显存就需要地址。CGA/EGA+ Chroma text video buffer 这个区域映射的就

是工作在文本模式的显存。同时显卡还有另外一个工作模式叫做图形模式，这个模式是目

前最最常用的模式。

#### 显卡在文本模式下的显示规则

我们知道，对于一个字符的编码通常有输入码、内码和字模码三种。其中字模码定义了

一个字符在屏幕上显示的点阵坐标。通常显卡内置一套关于基本英文字符的显示是很容易

做到的，而内置汉字的显示就较为麻烦。在这篇文档中我们只使用显卡的文本模式，不会涉及到图形模式的内容。因为一旦使用了图形模式的内容，我们就需要自行定义字符的字

模码了，这很繁琐而且对我们理解操作系统原理的意义不是很大。所以我们只使用显卡的

文本模式进行屏幕显示控制。所有在PC上工作的显卡，在加电初始化之后都会自动初始化

到80\*25的文本模式。在这个模式下，屏幕被划分为25行，每行可以显示80个字符，所以一屏可以显示2000个字符。上图中的0xB8000～0xBFFFF这个地址段便是映射到文本模式的显存的。当访问这些地址的时候，实际上读写的是显存区域，而显卡会周期性的读取这里的数据，并且把它们按顺序显示在屏幕上。

那么，按照什么规则显示呢？这就要谈到内码了。内码定义了字符在内存中存储的形

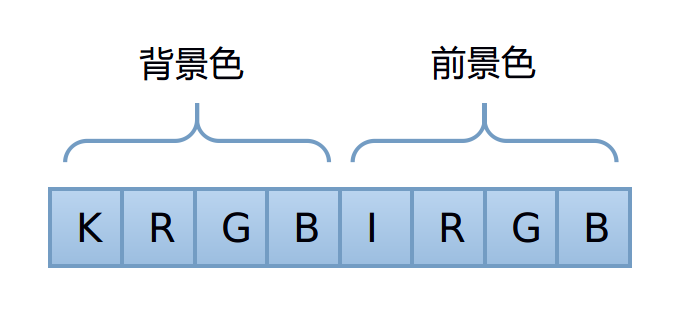
式，而英文编码就是大家所熟知的ASCII（American Standard Code for Information

Interchange，美国信息交换标准代码）码了。对应的关系很简单，从0xB8000这个地址开

始，每2个字节表示屏幕上显示的一个字符。从屏幕的第一行开始对应，一行接着一行的对

应下去。而这两个字节的前一个是显示字符的ASCII码，后一个是控制这个字符颜色和属性

的控制信息，这个字节的8个bit位表示不同的含义。每一位的含义如图所示：



这些位的组合效果如下图所示：

这两张图可以帮助我们在显卡的字符模式显示彩色的文本了，懂得这些原理对于探索性

质的显示也就足够了。

理解了显卡文本模式的原理之后接下来就是对屏幕显示控制编码了。不过显卡除了显

示内容的存储单元之外，还有部分的显示控制单元需要了解。这些显示控制单元被编制在

了独立的I/O空间里，需要用特殊的in/out指令去读写。这里相关的控制寄存器多达300多

个，显然无法一一映射到I/O端口的地址空间。对此工程师们解决方案是，将一个端口作为

内部寄存器的索引：0x3D4，再通过0x3D5端口来设置相应寄存器的值。



#### 颜色的枚举定义和屏幕操作函数的实现

接下来是颜色定义的枚举和一些屏幕控制函数的声明。代码如下：

代码10: include/console.h

#ifndef INCLUDE\_SYS\_CONSOLE\_H\_

#define INCLUDE\_SYS\_CONSOLE\_H\_

#include "types.h"

typedef

enum real\_color {

rc\_black = 0,

rc\_blue = 1,

rc\_green = 2,

rc\_cyan = 3,

rc\_red = 4,

rc\_magenta = 5,

rc\_brown = 6,

rc\_light\_grey = 7,

rc\_dark\_grey = 8,

rc\_light\_blue = 9,

rc\_light\_green = 10,

rc\_light\_cyan = 11,

rc\_light\_red = 12,

rc\_light\_magenta = 13,

rc\_light\_brown = 14, // yellow

rc\_white = 15

} real\_color\_t;

// 清屏操作

void console\_clear();

// 屏幕输出一个字符 带颜色

void console\_putc\_color(char c, real\_color\_t back, real\_color\_t fore);

// 屏幕打印一个以 \0 结尾的字符串 默认黑底白字

void console\_write(char \*cstr);

// 屏幕打印一个以 \0 结尾的字符串 带颜色

void console\_write\_color(char \*cstr, real\_color\_t back, real\_color\_t fore);

// 屏幕输出一个十六进制的整型数

void console\_write\_hex(uint32\_t n, real\_color\_t back, real\_color\_t fore);

// 屏幕输出一个十进制的整型数

void console\_write\_dec(uint32\_t n, real\_color\_t back, real\_color\_t fore);

#endif // INCLUDE\_SYS\_CONSOLE\_H\_

参照着前面的表格，理解颜色的枚举类型并不困难。接下来是显存起始位置和当前输出

的屏幕位置的变量定义。同时，我们将屏幕抽象为一个80\*25的二维数组，每个数组成员都

是2个字节，表示屏幕上显示的一个字符。

// VGA 的显示缓冲的起点是 0xB8000

static uint16\_t \*video\_memory = (uint16\_t \*)0xB8000;

// 屏幕"光标"的坐标

static uint8\_t cursor\_x = 0;

static uint8\_t cursor\_y = 0;

请大家留意这里变量定义时候的static 限定符，当一个全局变量或者函数只在本模

块文件内被使用时，最好限定其作用域。每个模块应当尽可能的向外部暴露较少的接口。

##### 屏幕输入光标的移动

在本模块内，cursor\_x 和cursor\_y 这两个变量指明了逻辑上的当前输出位置，但

是并没有实际上移动硬件的显示"光标"，下面的函数实现了根据这两个变量的值移动光标

的功能。

static void move\_cursor()

{

// 屏幕是 80 字节宽

uint16\_t cursorLocation = cursor\_y \* 80 + cursor\_x;

// 在这里用到的两个内部寄存器的编号为14与15，分别表示光标位置

// 的高8位与低8位。

outb(0x3D4, 14); // 告诉 VGA 我们要设置光标的高字节

outb(0x3D5, cursorLocation >> 8); // 发送高 8 位

outb(0x3D4, 15); // 告诉 VGA 我们要设置光标的低字节

outb(0x3D5, cursorLocation); // 发送低 8 位

}

这里的端口和设置值都是固定的，也没有什么道理可讲。虽然显卡的各项技术发展的很

快，但是这个原始的VGA标准被所有显卡完整的保存了下来。

##### 清屏操作

然后是清屏操作，其实这里的"清屏"很简单，其实就是用白底黑字的"空格符"覆盖整

个屏幕的显示区域罢了。这么做自然就实现了我们想要的"清屏"操作了。代码很简单：

void console\_clear()

{

uint8\_t attribute\_byte = (0 << 4) | (15 & 0x0F);

uint16\_t blank = 0x20 | (attribute\_byte << 8);

int i;

for (i = 0; i < 80 \* 25; i++) {

video\_memory[i] = blank;

}

cursor\_x = 0;

cursor\_y = 0;

move\_cursor();

}

##### 屏幕滚动显示

那么屏幕滚动呢？用C语言来描述实际上就是将后24行的数据全部向上挪动一行，最后一行清空罢了，就是这么简单。

static void scroll()

{

// attribute\_byte 被构造出一个黑底白字的描述格式

uint8\_t attribute\_byte = (0 << 4) | (15 & 0x0F);

uint16\_t blank = 0x20 | (attribute\_byte << 8); // space 是 0x20

// cursor\_y 到 25 的时候，就该换行了

if (cursor\_y >= 25) {

// 将所有行的显示数据复制到上一行，第一行永远消失了...

int i;

for (i = 0 \* 80; i < 24 \* 80; i++) {

video\_memory[i] = video\_memory[i+80];

}

// 最后的一行数据现在填充空格，不显示任何字符

for (i = 24 \* 80; i < 25 \* 80; i++) {

video\_memory[i] = blank;

}

// 向上移动了一行，所以 cursor\_y 现在是 24

cursor\_y = 24;

}

}

##### 显示字符串

那么屏幕显示字符串呢？我们可以先实现屏幕显示一个字符的函数，那么屏幕显示一个

字符串不就可以了么。这几个函数的实现如下：

void console\_putc\_color(char c, real\_color\_t back, real\_color\_t fore)

{

uint8\_t back\_color = (uint8\_t)back;

uint8\_t fore\_color = (uint8\_t)fore;

uint8\_t attribute\_byte = (back\_color << 4) | (fore\_color & 0x0F);

uint16\_t attribute = attribute\_byte << 8;

// 0x08 是退格键的 ASCII 码

// 0x09 是tab 键的 ASCII 码

if (c == 0x08 && cursor\_x) {

cursor\_x--;

} else if (c == 0x09) {

cursor\_x = (cursor\_x+8) & ~(8-1);

} else if (c == '\r') {

cursor\_x = 0;

} else if (c == '\n') {

cursor\_x = 0;

cursor\_y++;

} else if (c >= ' ') {

video\_memory[cursor\_y\*80 + cursor\_x] = c | attribute;

cursor\_x++;

}

// 每 80 个字符一行，满80就必须换行了

if (cursor\_x >= 80) {

cursor\_x = 0;

cursor\_y ++;

}

// 如果需要的话滚动屏幕显示

scroll();

// 移动硬件的输入光标

move\_cursor();

}

void console\_write(char \*cstr)

{

while (\*cstr) {

console\_putc\_color(\*cstr++, rc\_black, rc\_white);

}

}

void console\_write\_color(char \*cstr, real\_color\_t back, real\_color\_t fore)

{

while (\*cstr) {

console\_putc\_color(\*cstr++, back, fore);

}

}

代码里唯一需要注意的便是输出后要检查当前的位置和判断一些特殊的符号表示的操

作，例如换行之类的实现。同时一定要注意修改存储当前位置的两个变量和移动屏幕上的

光标，而且屏幕输出满了以后要上滚。我们暂时不考虑诸如屏幕翻页之类的功能。至于屏

幕输出十六进制数字和十进制数字的函数这并不困难，不再赘述。

### TTY任务

#### TTY任务框架

#### 多控制台

#### 完善键盘处理

## 进程间通信

### 微内核

### IPC

### 实现IPC

## 文件系统

### 硬盘简介

### 硬盘驱动程序

### 硬盘分区表

### 遍历硬盘分区

### FAT16文件系统

### FAT16关键数据结构

### 文件系统接口

### 将TTY纳入文件系统

## 内存管理

### fork

### exit和wait

### exec

## C运行时库与简易Shell

### C运行时库

我们的操作系统肯定是要运行我们自己编写的应用程序的，应用程序中必不可少的要使用系统调用，所以需要我们把应用程序可能用到系统调用和接口函数整理成一个C运行时库。

截至目前，我们可以用来链接成库的系统调用和接口函数及其所在的文件有如下这些：

系统调用：sendrec和printx

- lib/syscall.asm

字符串操作：memcpy、memset、strcpy、strlen

- lib/syscall.asm

文件系统接口：

- lib/open.c

- lib/read.c

- lib/write.c

- lib/close.c

- lib/unlink.c

内存管理接口：

- lib/fork.c

- lib/exit.c

- lib/wait.c

进程接口：

- lib/getpid.c

杂项：

- lib/vprintf.c

- lib/printf.c

我们把这些函数用ar工具链接成一个静态链接库，命名为menguozicrt.a。方法如下：

menguozicrt.a : lib/syscall.o\

lib/printf.o lib/vsprintf.o\

lib/string.o lib/misc.o\

lib/open.o lib/read.o lib/write.o lib/close.o lib/unlink.o\

lib/getpid.o lib/stat.o\

lib/fork.o lib/exit.o lib/wait.o lib/exec.o

ar rcs $@ $^

### 简单用户程序

有了库，我们就可以编写自己的应用程序了。我们写过很多种语言的Hello，world小程序，这一次，我们将在自己的操作系统上实现。除此之外，我们实现两个简单使用的shell命令行程序echo和pwd，分别用于显示字符串和显示当前工作目录。

#### hello

#include "stdio.h"

int main(int argc, char \* argv[])

{

printf("Hello, world\n");

return 0;

}

#### echo

#include "stdio.h"

int main(int argc, char \* argv[])

{

int i;

for (i = 1; i < argc; i++)

printf("%s%s", i == 1 ? "" : " ", argv[i]);

printf("\n");

return 0;

}

#### pwd

#include "type.h"

#include "stdio.h"

int main(int argc, char \* argv[])

{

printf("/\n");

return 0;

}

#### 编译链接应用程序

#### 安装应用程序到硬盘

### 简易Shell

实现了fork()和exec()后，我们可以实现一个简单Shell，不妨称之为MShell。像Linux的bash和Windows的Powershell那样的Shell当然很复杂，但只要满足能够读取命令文件并执行这个基本的需求，我们就可以谓之未一个简单的Shell。

MShell的实现代码如下：

Menguozi\_Shell（kernel/main.c）

void Menguozi\_Shell(const char \* tty\_name)

{

int fd\_stdin = open(tty\_name, O\_RDWR);

assert(fd\_stdin == 0);

int fd\_stdout = open(tty\_name, O\_RDWR);

assert(fd\_stdout == 1);

char rdbuf[128];

while (1) {

write(1, "[menguozi@localhost] $ ", 23);

int r = read(0, rdbuf, 70);

//printf("r :%d\n",r);

rdbuf[r] = 0;

int argc = 0;

char \* argv[PROC\_ORIGIN\_STACK];

char \* p = rdbuf;

char \* s;

int word = 0;

char ch;

do {

ch = \*p;

if (\*p != ' ' && \*p != 0 && !word) {

s = p;

word = 1;

}

if ((\*p == ' ' || \*p == 0) && word) {

word = 0;

argv[argc++] = s;

\*p = 0;

}

p++;

} while(ch);

argv[argc] = 0;

int fd = open(argv[0], O\_RDWR);

if (fd == -1) {

if (rdbuf[0]) {

write(1, "{", 1);

write(1, rdbuf, r);

write(1, "}\n", 2);

}

}

else {

close(fd);

int pid = fork();

if (pid != 0) { /\* parent \*/

int s;

wait(&s);

}

else { /\* child \*/

int ret = execv(argv[0], argv);

printf("execv ret: %d\n",ret);

printf("child ret\n");

}

}

}

close(1);

close(0);

}

我们用Init进程fork两个子进程，分别在终端TTY1和终端TTY2上运行两个Shell。在每个Menguozi\_Shell中，用系统调用read()读取用户输入，然后fork出一个子进程，把exec()交由子进程来执行。遇到不合法的命令时，将用户输入原样输出。