

# 实验三：C与汇编开发独立批处理的内核

## 实验目的

1. 加深理解操作系统内核概念
2. 了解操作系统开发方法
3. 掌握汇编语言与高级语言混合编程的方法
4. 掌握独立内核的设计与加载方法
5. 加强磁盘空间管理工作

## 实验要求

1. 知道独立内核设计的需求
2. 掌握一种x86汇编语言与一种C高级语言混合编程的规定和要求
3. 设计一个程序，以汇编程序为主入口模块，调用一个C语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成COM格式程序，在DOS或虚拟环境运行。
4. 汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个COM格式程序的独立内核。
5. 再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。
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：Windows 10

虚拟机软件：Bochs、qemu

## 实验开发工具

开发环境: Windows Subsystem for 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

磁盘映像文件浏览编辑工具: WinHex

代码编辑器: Visual Studio Code

## 程序设计

这次实验我分为了两个阶段

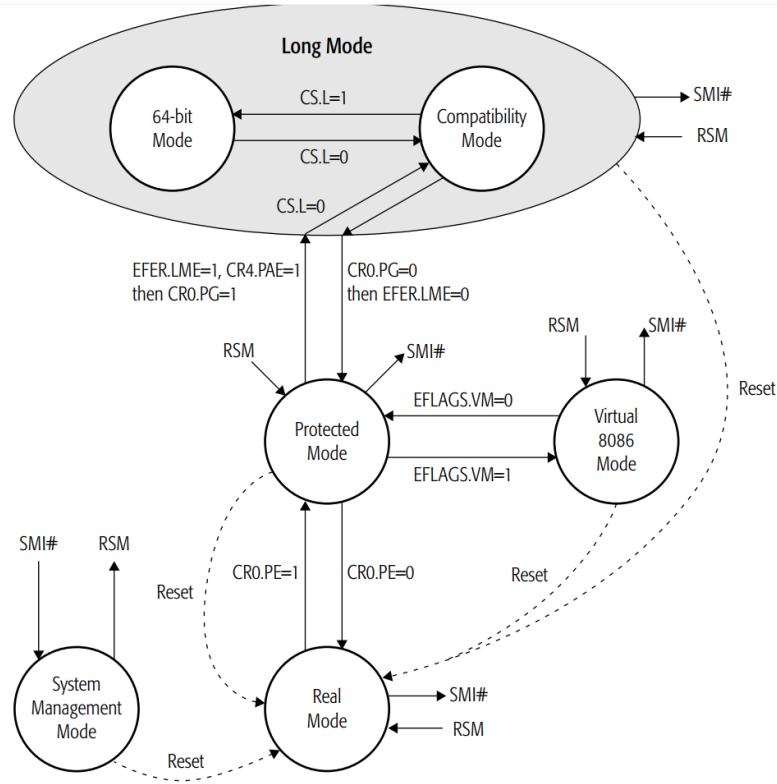
- 进入长模式
- 利用C语言重写系统内核

同时, 由于我使用的是gcc的工具链, 我将会从intel的汇编格式转换为gcc下的默认汇编格式at&t汇编格式。

### 进入长模式

本次实验中, 我利用了AMD官方提供的文档《AMD64 Architecture Programmer's Manual: Volumes 1-5》, 使得引导程序完成了从实模式切换到保护模式, 再从保护模式切换到了长模式的功能。

下图是我在文档里截取的一副各个模式之间的切换图。



**Figure 1-6. Operating Modes of the AMD64 Architecture**

可以看到，要从实模式切换到64-bit的长模式，总共需要以下几个步骤：

- 打开A20地址线控制
- 建立全局描述符表（GDT）、中断描述符表(IDT)
- CR0.PE置为1
- 使用ljmp指令，正确地初始化段寄存器CS
- 利用mov指令，设置好段寄存器DS、ES、FS、SS、GS
- 设置好4级页表的映射（PML-4、PDPT、PDT、PT），并将PML-4的起始地址赋值给寄存器CR3
- 置CR4.PAE为1，启用64位页表项
- 置EFER.LME、CR0.PG为1，打开分页式内存管理机制，进入长模式的兼容模式
- 建立长模式下的全局描述符表（GDT）（与保护模式的不同）、中断描述符表(IDT)
- 继续使用ljmp指令，初始化段寄存器CS，使得CS.L=1，进入长模式里的64-bit模式

### 打开A20地址控制线

在实模式下，CPU只能访问1M内存。在保护模式下，由于使用32位地址线，如果A20恒等于0，那么系统只能访问奇数兆的内存，即只能访问0--1M、2-3M、4-5M.....，这样无法有效访问所有可用内存。但当我们使能了A20地址控制线，我们就可以在保护模式，访问全部的内存。

为了打开A20地址控制线，我们需要通过键盘控制器8042发送一个命令完成。

首先，利用查询式访问I/O端口方式，等待8042的输入缓冲区为空

其次，发送写入8042输出端口请求命令到8042的输入缓冲区

然后，继续采用查询式访问I/O端口方式，等待8042的输入缓冲区为空

最后，写入设置命令，使能A20地址线。

```
# 这部分代码参考的是https://github.com/chyyuu/ucore_os_lab的实现

seta20.1:
    inb $0x64, %al # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.1

    movb $0xd1, %al # 0xd1 -> port 0x64
    outb %al, $0x64 # 0xd1 means: write data to 8042's P2 port

seta20.2:
    inb $0x64, %al # Wait for not busy(8042 input buffer empty).
    testb $0x2, %al
    jnz seta20.2

    movb $0xdf, %al # 0xdf -> port 0x60
    outb %al, $0x60 # 0xdf = 11011111, means set P2's A20 bit(the 1 bit) to 1
```

## 建立全局描述符表（GDT）、中断描述符表(IDT)

在保护模式下，CPU主要采取的是分段存储管理机制和分页式存储管理机制。

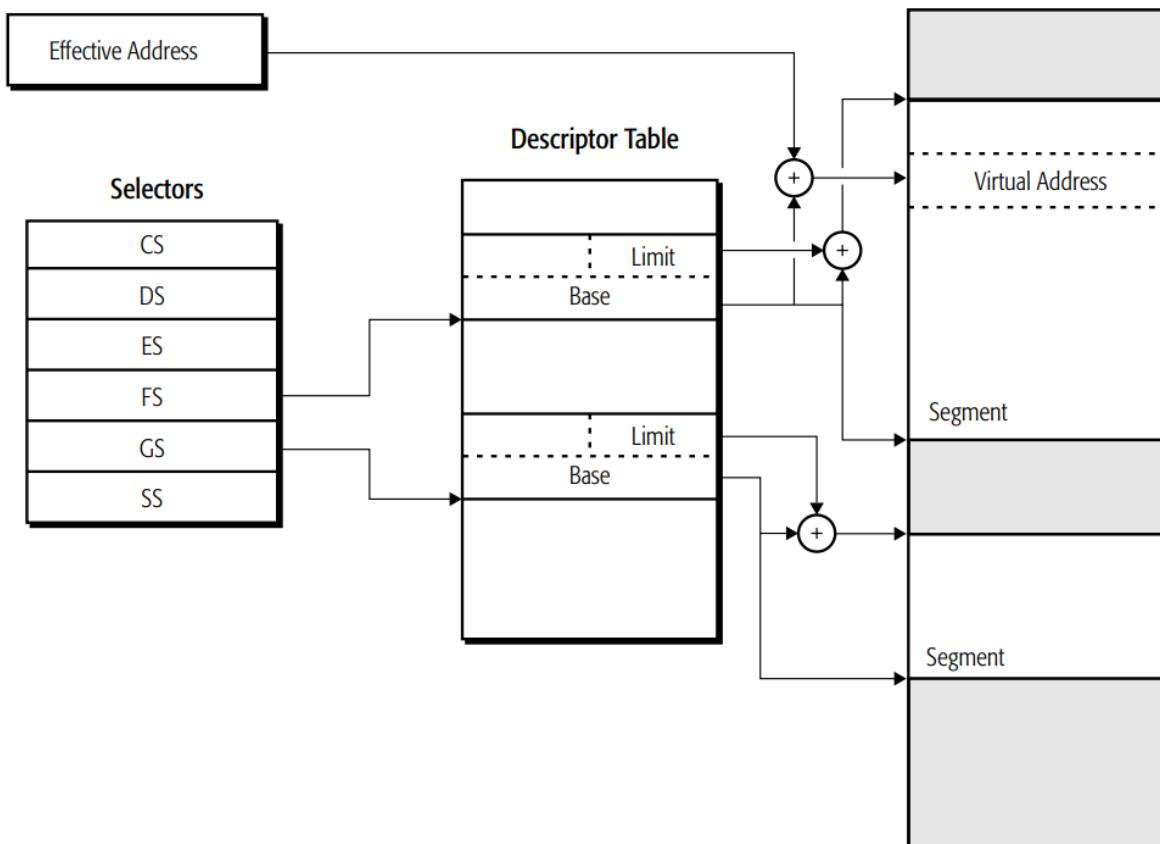


Figure 1-1. Segmented-Memory Model

由于分页（Paging）在保护模式不是必须打开的，而且我们的目的是进入长模式，只需要在保护模式下设置好必要的配置即可，所以只需要建立起保护模式下的全局描述符表（GDT）即可。同时，中断机制也不是必要的，所以我只是设置了IDTR寄存器，而没有建立起完整的中断描述符表。

```
# 这部分代码参考的是https://github.com/chyyuu/ucore_os_lab的实现，我修改了DS的取值，同时增加了GS的设置

.set PROT_MODE_CSEG,          0x8          # 这里设置CS指向GDT里的第二项
lgdt gdtdesc      # 设置GDTR寄存器
lidt idtdesc      # 设置IDTR寄存器

# 设置CR0.PE = 1, 进入保护模式
movl %cr0, %eax
orl $CR0_PE_ON, %eax
movl %eax, %cr0

# 初始化cs的值
ljmp $PROT_MODE_CSEG, $protcseg      # protcseg是保护模式代码的入口地址

gdt:
SEG_NULLASM           # null seg
SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff) # code seg for bootloader and kernel
SEG_ASM(STA_W, 0x0, 0xffffffff)       # data seg for bootloader and kernel
SEG_ASM(STA_W, 0xB8000, 0xffffffff)   # graph seg for bootloader and kernel

gdtdesc:
.word 0x1f            # sizeof(gdt) - 1
.long gdt             # address gdt
idtdesc:
.word 0x0              # set idt empty
.long 0
```

gdt表的设置按照下图：

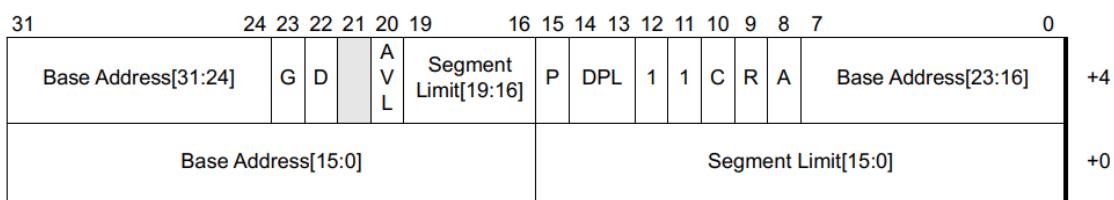
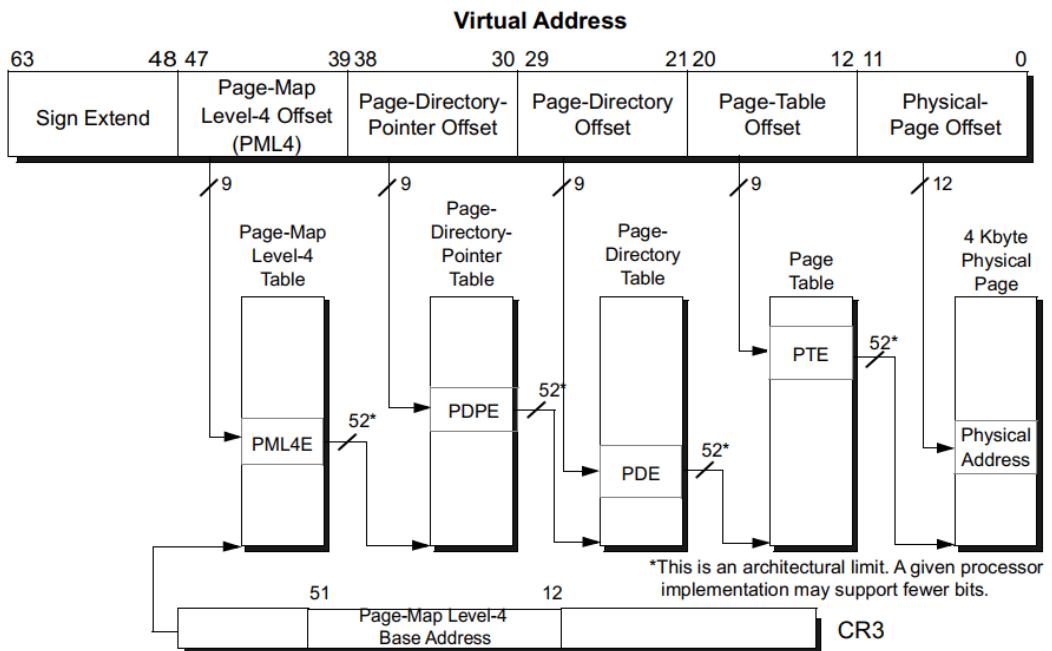


Figure 4-14. Code-Segment Descriptor—Legacy Mode

由于保护模式不是我实验的目标，这里不再继续详细说明了。

## 设置好4级页表的映射

长模式下，CPU虚拟内存的寻址方式（一页大小是4KB）如下图：



**Figure 5-17. 4-Kbyte Page Translation—Long Mode**

从上面可以看到，要建立正确页表，要设置好上面4个页表的映射，同时还要设置CR3的值。

具体到每一个表的表项格式如下图：

63 62		52 51		32															
N	X	Available		Page-Directory-Pointer Base Address (This is an architectural limit. A given implementation may support fewer bits.)															
31		Page-Directory-Pointer Base Address																AVL	12 11 9 8 7 6 5 4 3 2 1 0
																		M B Z M B Z N I A C D P T U S R / W / W P	

**Figure 5-18. 4-Kbyte PML4E—Long Mode**

63 62		52 51		32															
N	X	Available		Page-Directory Base Address (This is an architectural limit. A given implementation may support fewer bits.)															
31		Page-Directory Base Address																AVL I G N 0 I G A P P U R / / P	12 11 9 8 7 6 5 4 3 2 1 0
																		M B Z M B Z N I A C D P T U S R / W / W P	

**Figure 5-19. 4-Kbyte PDPE—Long Mode**

63 62		52 51		32															
N	X	Available		Page-Table Base Address (This is an architectural limit. A given implementation may support fewer bits.)															
31		Page-Table Base Address																AVL I G N 0 I G A P P U R / / P	12 11 9 8 7 6 5 4 3 2 1 0
																		M B Z M B Z N I A C D P T U S R / W / W P	

**Figure 5-20. 4-Kbyte PDE—Long Mode**

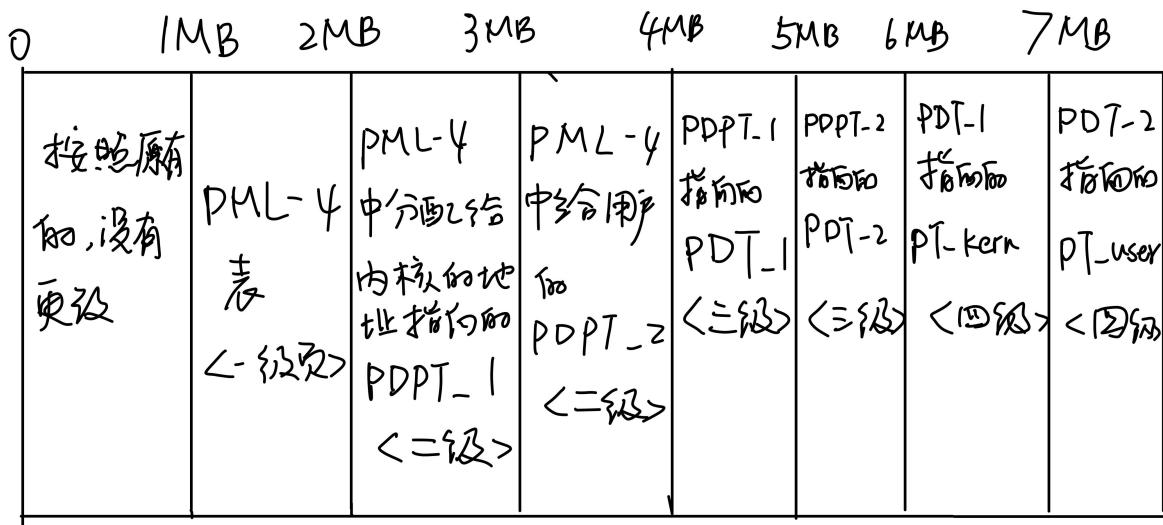
63 62		52 51		Physical-Page Base Address (This is an architectural limit. A given implementation may support fewer bits.)																32								
N	X	Available																										
31																				0								
Physical-Page Base Address																AVL	G	P	A	D	A	P	P	U	R	P		
																AT	C	W	T	S	W	/	/					

Figure 5-21. 4-Kbyte PTE—Long Mode

按照对应项进行填充就可以了。在这里，为了方便，我将所有的页表都设置为可执行、可写入、页表项存在、运行Cache、Write-Back的写回机制。

但是，从后面的实验看出。要及时的、正确的设置屏幕显示，地址0xb8000这一页，需要设置为Not Cacheable、Write Through的机制，确保它能够及时的显示。

我设计的实际地址与虚拟地址的映射关系及虚拟地址的分配，如下图



实际内存映射表一

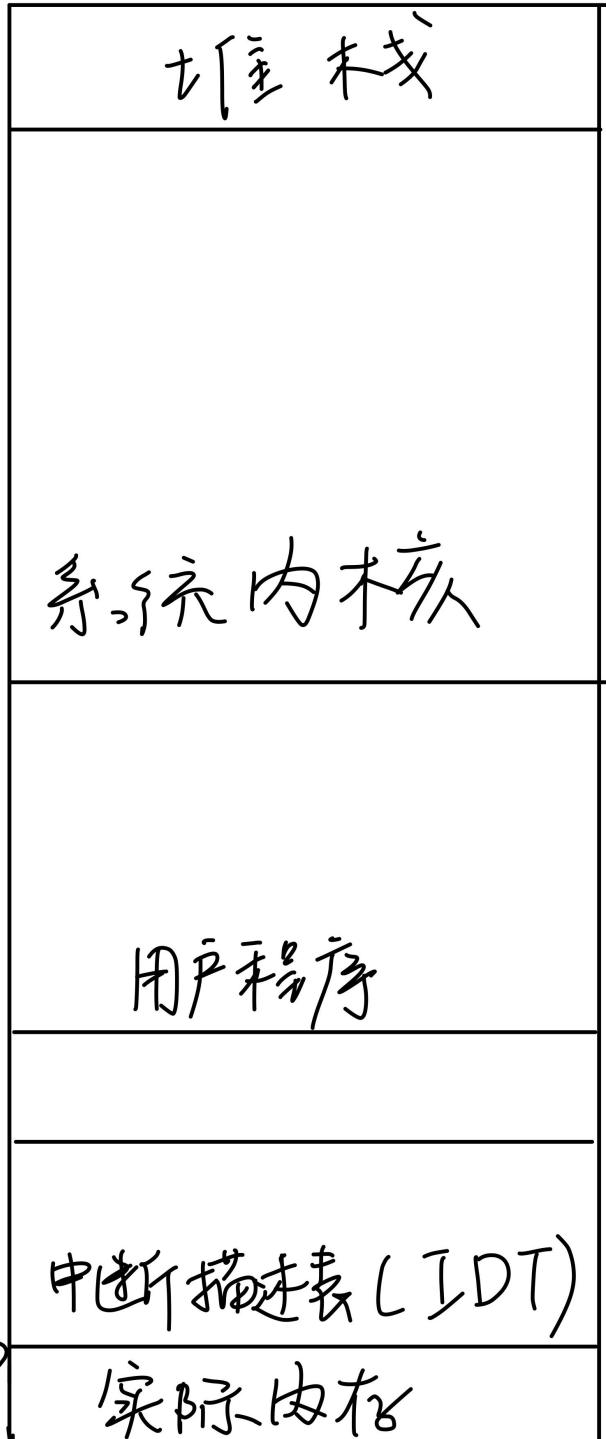
# 接上表

8MB 9MB 10MB 11MB 12MB

虚拟地址	虚拟地址	虚拟地址		虚拟地址	
0xfffff8000 0000 0000	0x0000 0000 0010 0000	0xffff ffff fff 0000 0	PT-User_2 <四级>	0x0000 0000 00C0 0000	.
用于存放 IDT		→ 0xffff ffff ffff ffff 存放栈	用于存放 存放加壳 存放用户 程序		未分配

实际内存地址表 - 2

0xffff ffff  
ffff ffff



虚拟内存空间

由于64位模式需要的内存空间非常多，所以我把原来实验环境的1MB改为了32MB。从上图可以看出，由于我为了尽快进入长模式进行编程，实际上我的内存分配是非常粗糙的（后面的时间需要增加空闲内存空间分配模块和虚拟内存管理模块）。

虚拟空间的分配是参考Linux x64下的虚拟内存空间，但是很多部分设计还不完善，需要继续细致的分段。

```
// 建立页表
// 这部分代码是我用c语言写的，再按32位模式使用gcc进行编译，得到其汇编码后加入到引导程序里
// 详细说明在后面部分

#define PML4_BASE 0x100000

void SetPageTable() {
    unsigned long long *PML4 = PML4_BASE;
    /* Setting Stack virtual memory */
    PML4[0] = 0x300027;
    PML4[511] = 0x200027;
    PML4[256] = 0x200027;

    unsigned long long *pdpl_kern = 0x200000;
    pdpl_kern[0] = 0x400027;
    pdpl_kern[511] = 0x400027;

    unsigned long long *pdpl_user = 0x300000;
    pdpl_user[0] = 0x500027;

    unsigned long long *pdl_kern = 0x400000;
    pdl_kern[0] = 0x600027;
    pdl_kern[511] = 0x600027;

    unsigned long long *pdl_user = 0x500000;
    pdl_user[0] = 0x700027;
    pdl_user[2] = 0xb00027;

    unsigned long long *pt_kern = 0x600000;
    /* Setting kernal stack for 1MB */
    unsigned long long kern_stack = 0xa00027;
    for (int i = 0; i < 1 << 8; ++i, kern_stack += 0x1000) {
        pt_kern[i + 256] = kern_stack;
    }

    /* Setting kernal code memory */
    unsigned long long *kern_code = 0x800027;
    for (int i = 0; i < 1 << 8; ++i, kern_code += 0x1000) {
        pt_kern[i] = kern_code;
    }

    /* 0x0100_00000 */
    unsigned long long *pt_user_1 = 0x700000;
    /* Setting first 1MB to the real memory */
    unsigned long long first_one_mega = 0x000027;
    for (int i = 0; i < 1 << 8; ++i, first_one_mega += 0x1000) {
        pt_user_1[i] = first_one_mega;
    }
    pt_user_1[184] = 0xB802f; /* Setting display memory */
```

```

    unsigned long long idt = 0x900027;
    for (int i = 0; i < 1 << 8; ++i, idt += 0x1000) {
        pt_user_1[i + 256] = idt;
    }

    unsigned long long *pt_user_2 = 0xb00000;
    unsigned long long user_code = 0xc00027;
    for (int i = 0; i < 1 << 9; ++i, user_code += 0x1000) {
        pt_user_2[i] = user_code;
    }

    /* Setting IDT for long mode */
    unsigned long long *idt_addr = 0x900000;
    for (int i = 0; i < 1 << 8; ++i) {
        idt_addr[i * 2] = 0x0000e0000080000;
        idt_addr[i * 2 + 1] = 0;
    }
}

```

以上的赋值操作均按照我在上面提供的3张设计图进行的，经过测试验证，代码所分配的内存空间均正确映射了。

## 进入长模式

在建立好页表以后，实际我离进入长模式已经不远了。接下来的步骤，就是置CR4.PAE、EFER.LME、CR0.PG为1。

```

# 这部分代码我参考的是《AMD64 Architecture Programmer's Manual: Volumes 1-5》卷2第十四章
# Enable the 64-bit page-translation-table entries by
# setting CR4.PAE=1 (this is _required_ before activating
# long mode). Paging is not enabled until after long mode
# is enabled.
#
movl %cr4, %eax
bts $0x5, %eax
movl %eax, %cr4

# Create the long-mode page tables, and initialize the
# 64-bit CR3 (page-table base address) to point to the base
# of the PML4 page table. The PML4 page table must be located
# below 4 Gbytes because only 32 bits of CR3 are loaded when
# the processor is not in 64-bit mode.
#
movl $PML4_BASE, %eax      # Pointer to PML4 table (<4GB).
movl %eax, %cr3            # Initialize CR3 with PML4 base.
# Enable long mode (set EFER.LME=1).
#
mov $0x0c0000080, %ecx      # EFER MSR number.
rdmsr                      # Read EFER.
bts $0x8, %eax              # Set LME=1.
wrmsr                      # Write EFER.

```

```

#
# Enable paging to activate long mode (set CR0.PG=1)
#
movl %cr0, %eax           # Read CR0.
orl $0x80000000, %eax     # Set PG=1.
movl %eax, %cr0            # Write CR0.

```

上面的代码均有AMD官方文档提供的注释，我在这里就不再进行继续的阐释。

在运行完上面的代码以后，实际上，我们还只是进入了64位兼容模式，离我们最终目标——64bit长模式还有一步：CS.L = 1。虽然，在长模式里，程序计数器就只是64位的rip寄存器了，但是段寄存器CS还是不能直接赋值，需要通过设置长模式下的gdt，再通过ljmp指令完成CS在长模式的初始化。

```

.set LONG_MODE_CSEG,          0x8

lgdt gdt64desc               # 设置gdtr
lidt idt64desc                # 设置idtr

start64_linear:
# 长跳转指令设置CS
ljmp $LONG_MODE_CSEG, $longseg      # longseg是长模式程序入口

gdt64:
# NULL
.word 0xffff
.word 0
.byte 0
.byte 0
.byte 1
.byte 0
# Code 设置cs段寄存器，用于进入64bit-mode

gdt64Code:
.word 0
.word 0
.byte 0
.byte 0x9a
.byte 0xaf
.byte 0
# Data 设置数据段寄存器，但实际上作用不大
.word 0
.word 0
.byte 0
.byte 0x92
.byte 0
.byte 0

gdt64desc:
.word 0x1f                  # 长度为64位
.quad gdt64

idt64desc:
.word 0x8000                 # 256个中断向量
.quad 0x100000                # idt起始地址为0x100000

```

上面那段代码里，值得注意的是对于中断描述符表的初始化。因为在进入了64位长模式以后，BIOS中断调用就已经不起作用了，需要自己编写一些中断例程或是硬件驱动程序实现I/O操作。

下面是我所编写的三个在本次实验里所需要用到的三个驱动程序

## 硬件驱动程序

要实现驱动程序，其实主要是利用I/O端口和一些芯片的操作命令。而我现在最需要的是主要有三个驱动：

- 读取扇区
- 屏幕输出
- 键盘读取

由于已经进入了64位长模式了，只要不使用标准C语言库，我们可以直接使用C语言进行编程，再利用gcc和ld链接器进行二进制文件的实现。所以，接下来，我所编写的驱动程序和系统内核，均是使用C语言，或是C语言内联汇编。

### 读扇区

由于读取软盘的操作实在太复杂了（中间还涉及到DMA的数据传送方式），同时，考虑到进入64位后带来的更大存储空间的需求，我决定改用硬盘作为我的磁盘映像文件。

访问硬盘总共有两种模式：CHS模式和LBA模式，在这里我采用的是LBA模式的PIO（Program IO）方式，即所有的IO操作是通过CPU访问硬盘的IO地址寄存器完成。

硬盘控制器的端口号是0x1f0到0x1f7。

读硬盘扇区的流程大致如下：

- 利用轮询方式查询硬盘是否准备好
- 发出读取扇区的命令
- 再次利用轮询方式查询硬盘是否准备好
- 通过字节流的方式，把磁盘扇区数据读到指定内存

```
#define SECTSIZE 512 // 定义一个扇区大小为512字节

// 由于接下来对于端口的读写操作有很多，我利用C语言内联汇编的方式，先把这些函数以C语言函数方式进行编写，方便后续调用

// 给定读取端口，返回端口的输出值
int inp(int port) {
    int res;
    asm(    "mov %1, %%edx\n\t"
            "in %%dx, %%al\n\t"
            "movl %%eax, %0\n\t"
            : "=r"(res)
            : "r"(port)
            : "%rdx", "%rax"
    );
    return res;
}
```

```

// 给定写入端口和写入信息
void outp(int port, int info) {
    asm(    "movl %0, %%edx\n\t"
            "movl %1, %%eax\n\t"
            "out %%al, %%dx\n\n"
            :
            : "r"(port), "r"(info)
            : "%rax", "%rdx"
            );
}

// ins命令，用于连续读取
void insl(short port, void *des, unsigned long long len) {
    asm(    "cld\n\t"
            "movq %2, %%rcx\n\t"
            "movq %1, %%rdi\n\t"
            "movw %0, %%dx\n\t"
            "rep insl\n\t"
            :
            : "r"(port), "r"(des), "r"(len)
            :
            );
}

// 轮询是查询硬盘是否准备好
void WaitDisk() {
    while ((inp(0x1f7) & 0xC0) != 0x40)
        /* 等待硬盘空闲 */;
}

// 给定要加载的内存地址dst，存放扇区起始编号（以0开始），要读取的扇区数
void readsect(void *dst, int secno, short num_of_sec) {
    /* 等待硬盘空闲 */
    WaitDisk();

    /* 一些操作的命令 由于硬盘扇区总共可以有27位，第2、3、4、5条指令都是在给定扇区起始编号 */
    outp(0x1f2, num_of_sec);                      /* 设置读取扇区数 */
    outp(0x1f3, secno & 0xff);
    outp(0x1f4, (secno >> 8) & 0xff);
    outp(0x1f5, (secno >> 16) & 0xff);
    outp(0x1f6, ((secno >> 24) & 0xf) | 0xe0);
    outp(0x1f7, 0x20);                            /* 设置LBA模式 */

    /* 等待硬盘空闲 */
    WaitDisk();

    /* 给定要读取的扇区数，除以4是因为一次读取4个字节，一次读完全部到指定地址dst */
    insl(0x1f0, dst, SECTSIZE * num_of_sec / 4);
}

```

这里的读扇区驱动，是使得引导程序可以加载系统内核到 **0xffff800000000000** 地址。

使用纯C语言进行编写，代码的逻辑会更加清晰，也方便后续的修改。

## 屏幕显示

这部分驱动主要也是分为两部分：

- 读取和设置光标
- 输出字符到显示屏

### 读取和设置光标

事实上，VGA显卡内的寄存器很多，为了不过多占用主机的I/O空间，很多寄存器只能通过索引寄存器访问。

在标准的PC机上。`0x3d4` 和 `0x3d5` 两个端口可以用来读写显卡的内部寄存器。方法是先向 `0x3d4` 端口写入要访问的寄存器编号，再通过 `0x3d5` 端口来读写寄存器数据。存放光标位置的寄存器编号为14和15。两个寄存器合起来组成一个16位整数，这个整数就是光标的位置。比如0表示光标在第0行第0列，81表示第1行第1列（屏幕总共80列）。

```
// 光标位置变量，x代表列号，y代表行号
typedef struct CurPosition {
    int x, y;
} CurPosition;

// 获取光标函数
CurPosition GetCurPos() {
    unsigned short temp = 0;
    CurPosition res;
    outp(0x3d4, 14); /* 向0x3d4端口写入需要读取14号寄存器 */
    temp = (inp(0x3d5) & 0xff) << 8; /* 光标位置高八位 */
    outp(0x3d4, 15); /* 向0x3d4端口写入需要读取15号寄存器 */
    temp += inp(0x3d5); /* 光标位置低八位 */
    res.x = temp % 80;
    res.y = temp / 80;
    return res;
}

// 设置光标
void SetCurPos(CurPosition point_pos) {
    unsigned short pos = point_pos.y * 80 + point_pos.x;
    outp(0x3d4, 14); /* 向0x3d4端口写入需要写入14号寄存器 */
    outp(0x3d5, (pos >> 8) & 0xff); /* 设置光标高八位 */

    outp(0x3d4, 15); /* 向0x3d4端口写入需要写入15号寄存器 */
    outp(0x3d5, pos & 0xff); /* 设置光标低八位 */
}
```

## 输出字符到显示屏

考虑到平时编写C语言代码的习惯，这部分有两个函数 `putc()` 和 `puts()`，分别用于输出字符和字符串。

输出原理就是直接读写VGA显存地址，在当前光标位置输出

```
// 输出字符
void putc(char c) {
    unsigned char *vga_addr = 0xb8000;
    CurPosition cur_pos = GetCurPos(); /* 获取当前光标 */
    if (c == '\n') {
        cur_pos.x = 0; /* 回车 */
        ++cur_pos.y; /* 换行 */
    }
    else {
        /* 这里乘以2是因为每个字符由两个字节控制 */
        unsigned int ch_pos = (cur_pos.y * 80 + cur_pos.x) << 1;
        vga_addr[ch_pos] = c;
        vga_addr[ch_pos + 1] = 0x07; /* 设置字符前景色和背景色 */
        ++cur_pos.x;
    }
    SetCurPos(cur_pos); /* 更新光标位置 */
}

// 调用上面的putc()函数，输出字符串
void puts(char *str) {
    int index = 0;
    while (str[index] != '\0') {
        putc(str[index++]);
    }
}
```

正如之前提到的，`0xb8000` 地址被设置了不缓存、Write Through的策略，字符会立刻写入显存的。

## 键盘输入

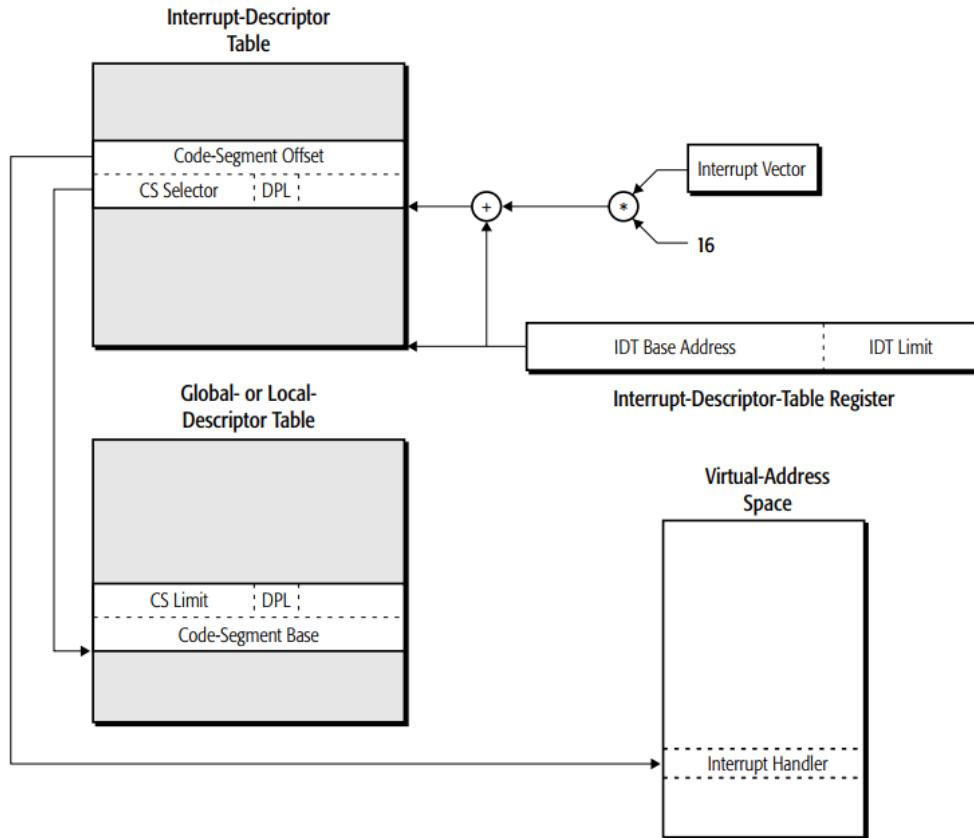
这部分是整个驱动程序里最麻烦的一部分。因为键盘输入实际上是一个异步操作。

这里利用计组里学到的8259芯片和i8042键盘控制芯片。工作原理是，当用户按下键盘时，会给8259中断控制的IR1发出信号，然后8259再对CPU发出中断控制信号，CPU通过查询中断向量表切换到IRQ1对应中断处理例程（也就是我们的键盘输入处理程序）。

要实现上面的操作，也是需要三部分工作：

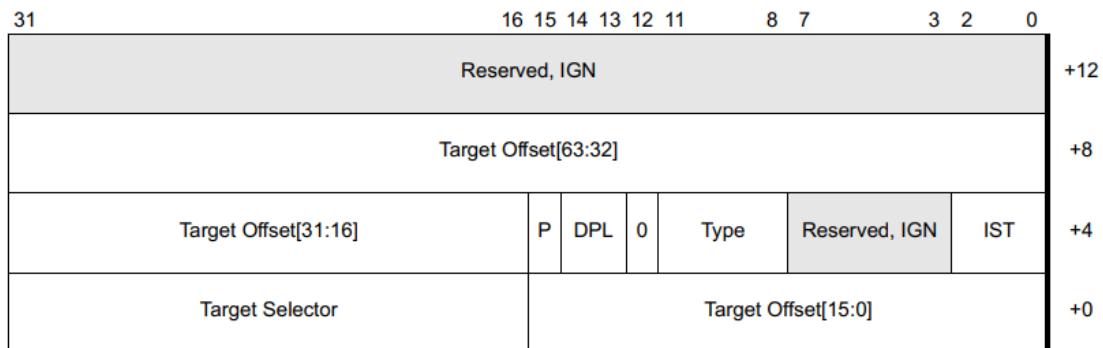
- 初始化编程8259芯片
- 编写键盘中断例程
- 装载到中断描述符表

64位中断描述符表的寻址和表项格式如下：



**Figure 8-12. Long-Mode Interrupt Control Transfer**

上面的CS Selector唯一的作用就是检查一下CPL，判断一下是否出现了特权级切换。为了简化起见，我都设为了00（内核态）。



**Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode**

这个表项里总长度是16字节，Target Offset就是中断例程的地址，P是该例程是否已经在内存里，IST即为（Interrupt Stack Table）涉及到堆栈地址切换。为了简便起见，我都没有使用这种复杂的功能（后续可能会补上）。

```
# 8259初始化程序，基本上就是对主8259芯片和从8259芯片进行编程
setint_8259:
    movb $0x11, %al      # ICW1
    out %al, $0x20
    out %al, $0xa0

    movb $0x20, %al      # master ICW2
```

```

out %al, $0x21

movb $0x28, %al      # slave ICW2
out %al, $0xa1

movb $0x04, %al      # master ICW3
out %al, $0x21

movb $0x02, %al      # slave ICW3
out %al, $0xa1

movb $0x01, %al      # ICW4
out %al, $0x21
out %al, $0xa1

movb $0xfd, %al
out %al, $0x21      # 屏蔽中断

movb $0xff, %al      # 屏蔽从8259的所有中断
out %al, $0xa1

```

由于中断不是本次实验的目标，我这里不做过多陈述。

```

// 键盘中断设置

/* 输入设置的中断向量号int_num及对应的中断向量例程地址func */
void SetInterrupt(unsigned short int_num, void *func) {
    __asm__ ("cli\n\t");                                /* 关闭中断 */
    unsigned long long *idt_addr = 0x100000;
    unsigned long long func_addr = (unsigned long long)func;
    idt_addr[int_num * 2] = (func_addr & 0xffff) | (((func_addr >> 16) & 0xffff) << 48)
| 0x8e0000080000;
    idt_addr[int_num * 2 + 1] = ((func_addr >> 32) & 0xffffffff);
// __asm__ ("sti\n\t");
}

// Scan Code Set 1
static char normalmap[256] = {
    NO, 0x1B, '1', '2', '3', '4', '5', '6', // 0x00
    '7', '8', '9', '0', '-', '=', '\b', '\t',
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', // 0x10
    'o', 'p', '[', ']', '\n', NO, 'a', 's',
    'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', // 0x20
    '`', '^', NO, '\\\\', 'z', 'x', 'c', 'v',
    'b', 'n', 'm', ',', '.', '/', NO, '*', // 0x30
    NO, ' ', NO, NO, NO, NO, NO, NO,
    NO, NO, NO, NO, NO, NO, '7', // 0x40
    '8', '9', '-', '4', '5', '6', '+', '1',
    '2', '3', '0', '.', NO, NO, NO, NO // 0x50
};

// 按了shift键后的键盘映射，在这里我其实还没有实现

```

```

static char shiftmap[256] = {
    NO,    033,   '!',   '@',   '#',   '$',   '%',   '^',   // 0x00
    '&',   '*',   '(',   ')',   '_',   '+',   '\b',   '\t',
    'Q',   'W',   'E',   'R',   'T',   'Y',   'U',   'I',   // 0x10
    'O',   'P',   '{',   '}',   '\n',   NO,   'A',   'S',
    'D',   'F',   'G',   'H',   'J',   'K',   'L',   ':',   // 0x20
    '"',   '~',   NO,   '|',   'Z',   'X',   'C',   'V',
    'B',   'N',   'M',   '<',   '>',   '?',   NO,   '*',   // 0x30
    NO,   ' ',   NO,   NO,   NO,   NO,   NO,   NO,
    NO,   NO,   NO,   NO,   NO,   NO,   NO,   '7',   // 0x40
    '8',   '9',   '-',   '4',   '5',   '6',   '+',   '1',
    '2',   '3',   '0',   '.',   NO,   NO,   NO,   NO   // 0x50
};

/* 设置键盘输入缓冲区 */
char output = 0;

void KbHandler() {
    unsigned char scan_code = inp(0x60);
    /* 暂时不支持大写输入 */
    if (scan_code < 0x80) {
        char ch = normalmap[scan_code];
        output = ch;
    }
    __asm__("movb $0x20, %al\n\t");
    __asm__("out %al, $0x20\n\t");           // 给CPU发EOI命令, 表示中断结束
    __asm__("nop\n\tleaveq\n\tiretq\n\t");
}

// 设置IRQ1的对应例程
void InitKeyboard() {
    SetInterrupt(KEYBOARD_IRQ, KbHandler);
}

```

事实上，键盘输入比较诡异的地方，是敲击键盘和手抬起以后，键盘控制器都会生成一个Scan Code，而且同一个键的Scan Code还不一样。这里，我按照个人习惯，计算敲击作为一次输入，而忽略了抬手。（敲击的和抬手的Scan Code相差0x80，所以有了上面的判断条件）。

在这部分，我还使用了一点点小技巧。事实上，C语言里的函数返回汇编指令是 `retq`，而中断例程的返回指令应该是 `iretq`。（注意：两者这里对于弹出栈的操作不一样，如果不这样写，会有重大bug）因此，我使用内联汇编，用我自己写的返回使得C语言编译出来的 `retq` 失效。

为了方便内核使用，我按照屏幕输出，也写了两个输入的函数。

```

// 功能和C语言的getchar()完全相同
char getchar() {
    output = 0;
    __asm__("sti\n\t");                  /* 打开中断, 允许键盘输入中断 */
    while (output == 0) {
        int a;
        for (int i = 0; i < 1000000; ++i) {
            a++;
    }
}
```

```

    }
    /* do nothing */;
}
__asm__("cli\n\t");           /* 关闭中断，禁用键盘输入中断 */
return output;
}

// 功能和c语言的gets(char *)基本上相同
void gets(char *buf) {
    int buf_index = 0;
    for (char ch = getchar(); ch != '\n'; ch = getchar()) {
        buf[buf_index++] = ch;
        putc(ch);
    }
    putc('\n');
    buf[buf_index] = '\0';
}

```

这部分比较奇怪的是 `getchar()` 的循环。本来我是写成下面这种行式

```

while (output == 0) {
    /* Do nothing */;
}

```

因为要等待键盘输入完改变 `output` 的值。但是很诡异的是，这种方式并不能得到我想要的效果。所以，我仿造老师在弹球实验写的延迟，人为地给予延迟等待键盘的输入，最终效果还不错，但是可能还需要修改。

## 系统内核

终于，在完成了模式切换，部分基础设施的构建，我可以利用自己写的接口在64位长模式下进行操作系统的编写。

为了修改代码的方便，同时因为64位和16位的程序执行方式非常不同，我用C语言重新编写了我的系统内核。

### 设置终端背景

同样的，第一件事情当然是设置个性化的终端背景

```

unsigned char *vga_addr = 0xb8000;
for (int i = 0; i < 25 * 80; ++i) {
    vga_addr[2 * i] = 0x20;           /* 设置青色背景 */
    vga_addr[2 * i + 1] = 0x30;
}

```

在C语言下这个逻辑就非常清晰了，一个循环语句，循环给显存赋值即可。

## 欢迎语句

这里用作交互功能，主要是介绍本监控程序可以调用的用户程序有哪些。

```
char *program_name[ ] = {"1.com", "2.com", "3.com", "4.com"};  
  
puts("Hello this is ouyhlans monitor program! The following programs which you can  
use: ");  
putc('\n');  
  
for (int i = 0; i < NUM_OF_PROGRAM; ++i) {  
    puts(program_name[i]);  
    putc('\n');  
}
```

直接使用上面写的驱动就可以实现了，完美地实现了驱动与系统代码编写的分离。

## 等待用户输入

依赖于C语言，我增强了这部分的功能：

- 可以识别用户输入的所有信息
- 读取到非法输入时提示重新输入

```
// 一个简单的字符串比较函数，主要用于识别用户输入  
int StrCmp(char *str1, char *str2, unsigned int n) {  
    for (int i = 0; i < n; ++i) {  
        if (str1[i] != str2[i])  
            return -1;  
    }  
    return 0;  
}  
void Welcoming() {  
    char str[32];  
    while (1) {  
        puts("Os1> "); /* 终端特色标识符 */  
        gets(str); /* 读取用户输入 */  
  
        /* 识别用户输入 */  
        int program_id = 0;  
        if (StrCmp(str, "1.com", 4) == 0) {  
            program_id = 1;  
        }  
        else if (StrCmp(str, "2.com", 4) == 0) {  
            program_id = 2;  
        }  
        else if (StrCmp(str, "3.com", 4) == 0) {  
            program_id = 3;  
        }  
        else if (StrCmp(str, "4.com", 4) == 0) {  
            program_id = 4;  
        }  
    }  
}
```

```
    }

    else if (StrCmp(str, "exit", 4) == 0) {
        puts("\n[System exits normally!]\n");
    }
    else {
        /* 提示非法输入 要求重新输入 */
        puts("Wrong input! Please enter again!\n");
        continue;
    }

    /* 加载用户程序选择 */
    switch (program_id) {
    case 1:
        LoadUserProgram(USER_1, 3);
        break;
    case 2:
        LoadUserProgram(USER_2, 3);
        break;
    case 3:
        LoadUserProgram(USER_3, 3);
        break;
    case 4:
        LoadUserProgram(USER_4, 3);
        break;
    default:
        /* Do nothing */;
    }
}
}
```

很明显的，在使用高级语言以后，程序的表现力提高了很多。

## 加载用户程序

这个模块实现的关键是

- 加载用户程序到内存 `0x400000`
- 移交控制权

加载部分其实就是调用读扇区驱动。

移交控制权，由于我没有想到什么比较好的方法，只能先利用内联汇编的 `call` 指令。

```

/* 指明要加载的用户程序扇区号和所需扇区数 */
void LoadUserProgram(unsigned int sec_no, unsigned int num_of_sec) {
    void *user_code_addr = 0x400000;
    readsect(user_code_addr, sec_no, num_of_sec); /* 加载用户程序到内存里 */
    __asm__("call %0\n\t"
           :
           : "r"(user_code_addr)
           :
    );
}

```

对比之前的长段加载程序，整个程序简洁了非常多！整个逻辑读起来也是清晰易懂。

## 用户程序

既然重写了系统内核，自然用户程序也需要重写。

由于前面的代码量已经太高了，我这部分没做太多修改，只是按照原来汇编代码的逻辑，按C语言重新写一遍而已。

程序流程：

- 显示小球当前所在位置
- 为了显示效果，延迟一段时间
- 决策算法，决定小球下一个位置

```

#define Lmax 0
#define Rmax 39
#define Umax 1
#define Dmax 12
#define Dn_Rt 1
#define Up_Rt 2
#define Up_Lt 3
#define Dn_Lt 4
#define delay 5000
#define ddelay 580

char *vga_addr = 0xb8000;
char ch_disp = 'A'; /* 要显示的字符 */
short x = 3; /* 行号 */
short y = 0; /* 列号 */
unsigned int rdul = Dn_Rt;
char bg_color = 0x30;
char ft_color = 0x0;

void ShowChar();
void DownRight();
void DownLeft();
void UpRight();
void UpLeft();
void SetReturnPos();

```

```

int _start() {
    /* 清屏 */
    for (int i = 0; i < 25 * 80; ++i) {
        vga_addr[2 * i] = 0x20;           /* 设置青色背景 */
        vga_addr[2 * i + 1] = 0x30;
    }

    for (int i = 0; i < 40; ++i) {
        ShowChar();

        /* 设置延迟，实现动态弹球 */
        for (int j = 0; j < delay; ++j) {
            for (int k = 0; k < ddelay; ++k) {
                /* Do nothing */;
            }
        }
    }

    switch (rdul) {
        case 1:
            DownRight();
            break;
        case 2:
            UpRight();
            break;
        case 3:
            UpLeft();
            break;
        case 4:
            DownLeft();
            break;
    }
    SetReturnPos();
    return 0;
}

/* 显示运动中的字符 */
void ShowChar() {
    unsigned cur_pos = x * 80 + y;
    ft_color = (ft_color + 1) % 0x10;

    vga_addr[2 * cur_pos] = ch_disp;      /* 设置显示的字符 */
    vga_addr[2 * cur_pos + 1] = ft_color | bg_color; /* 设置文字颜色 */

    ch_disp = (ch_disp + 1 > 'Z' ? 'A' : ch_disp + 1);
}

void DownRight() {
    x++, y++;
    if (x > Dmax) {
        /* 右下 -> 右上 */
    }
}

```

```

        x = Dmax - 1;
        rdul = Up_Rt;
    }
    else if (y > Rmax) {
        /* 右下 -> 左下 */
        y = Rmax - 1;
        rdul = Dn_Lt;
    }
}

void UpRight() {
    x--, y++;
    if (y > Rmax) {
        /* 右上 -> 左上 */
        y = Rmax - 1;
        rdul = Up廖;
    }
    else if (x < Umax) {
        /* 右上 -> 右下 */
        x = Umax + 1;
        rdul = Dn_Rt;
    }
}

void UpLeft() {
    x--, y--;
    if (x < Umax) {
        /* 左上 -> 左下 */
        x = Umax + 1;
        rdul = Dn廖;
    }
    else if (y < Lmax) {
        /* 左上 -> 右上 */
        y = Lmax + 1;
        rdul = Up_Rt;
    }
}

void DownLeft() {
    x++, y--;
    if (y < Lmax) {
        /* 左下 -> 右下 */
        y = Lmax + 1;
        rdul = Dn_Rt;
    }
    else if (x > Dmax) {
        /* 左下 -> 左上 */
        x = Dmax - 1;
        rdul = Up廖;
    }
}

```

```

void outp(int port, int info) {
    asm(
        "movl %0, %%edx\n\t"
        "movl %1, %%eax\n\t"
        "out %%al, %%dx\n\n"
        :
        : "r"(port), "r"(info)
        : "%rax", "%rdx"
    );
}

/* 设置回归光标在最后一行，便于显示， y是行， x是列 */
void SetReturnPos() {
    unsigned short pos = 24 * 80 + 0;
    outp(0x3d4, 14);
    outp(0x3d5, (pos >> 8) & 0xff); /* 设置光标高八位 */

    outp(0x3d4, 15);
    outp(0x3d5, pos & 0xff); /* 设置光标低八位 */

    /* 上滚一页 */
    for (int i = 0; i < 24 * 80; ++i) {
        vga_addr[i * 2] = vga_addr[i * 2 + 80 * 2];
        vga_addr[i * 2 + 1] = vga_addr[i * 2 + 1 + 80 * 2];
    }
}

```

## 实验过程和结果

### (1) 寻找一套匹配的汇编与c编译器组合。

由于Linux里有很多优秀的开源工具，我选择使用WSL的Ubuntu版本进行完整的实验。

#### 模板C语言函数

```

char Message[10] = "AaBbCcDdEe";
/* 变量_Message, 初值为AaBbCcDdEe */
char *upper(int a, int b, int c, int d, int e, int f){
    a++;
    b++;
    c++;
    d++;
    e++;
    f++;
    int i=0;
    while(Message[i]) {
        if (Message[i]>='a' && Message[i]<='z')
            Message[i]=Message[i]+ 'A' - 'a';
        i++;
    }
}

```

```

    }
    return Message;
}

```

这个程序主要是为了确定gcc编译时全局变量、函数调用、参数传递的情况，使用了如下的编译和链接命令：

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master ✘ [23:18:13]
→ gcc -N -ffreestanding -c 1.c -o 1.o

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master ✘ [23:18:43]
→ ld 1.o -o 1.elf
ld: warning: cannot find entry symbol _start; defaulting to 0000000004000e8

```

事实上，gcc里有汇编器as（可以用gcc命令直接使用），[-c]这个选项是纯编译的意思，我们再把生成的1.o，用链接器ld进行链接操作。

要观察汇编器as生成的汇编代码，直接用 objdump 反汇编链接后的代码就好了

## 分析生成文件

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master ✘ [23:23:24]
→ objdump -d 1.elf > 1

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master ✘ [23:23:38]
→ objdump -t 1.elf > 2

```

[-d]命令是查看代码，[-t]命令是查看符号列表文档，我们先来看代码部分。

```

1.elf:      file format elf64-x86-64

Disassembly of section .text:

0000000004000e8 <upper>:
4000e8: 55          push    %rbp
4000e9: 48 89 e5    mov     %rsp,%rbp
4000ec: 89 7d ec    mov     %edi,-0x14(%rbp)
4000ef: 89 75 e8    mov     %esi,-0x18(%rbp)
4000f2: 89 55 e4    mov     %edx,-0x1c(%rbp)
4000f5: 89 4d e0    mov     %ecx,-0x20(%rbp)
4000f8: 44 89 45 dc  mov     %r8d,-0x24(%rbp)
4000fc: 44 89 4d d8  mov     %r9d,-0x28(%rbp)
400100: 83 45 ec 01 addl   $0x1,-0x14(%rbp)
400104: 83 45 e8 01 addl   $0x1,-0x18(%rbp)
400108: 83 45 e4 01 addl   $0x1,-0x1c(%rbp)
40010c: 83 45 e0 01 addl   $0x1,-0x20(%rbp)
400110: 83 45 dc 01 addl   $0x1,-0x24(%rbp)
400114: 83 45 d8 01 addl   $0x1,-0x28(%rbp)
400118: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
40011f: eb 50        jmp    400171 <upper+0x89>
400121: 8b 45 fc    mov    -0x4(%rbp),%eax
400124: 48 98        cltq

```

```

400126: 48 8d 15 d3 0e 20 00    lea    0x200ed3(%rip),%rdx      # 601000
<Message>
40012d: 0f b6 04 10              movzbl (%rax,%rdx,1),%eax
400131: 3c 60                  cmp    $0x60,%al
400133: 7e 38                  jle    40016d <upper+0x85>
400135: 8b 45 fc              mov    -0x4(%rbp),%eax
400138: 48 98                  cltq
40013a: 48 8d 15 bf 0e 20 00    lea    0x200ebf(%rip),%rdx      # 601000
<Message>
400141: 0f b6 04 10              movzbl (%rax,%rdx,1),%eax
400145: 3c 7a                  cmp    $0x7a,%al
400147: 7f 24                  jg    40016d <upper+0x85>
400149: 8b 45 fc              mov    -0x4(%rbp),%eax
40014c: 48 98                  cltq
40014e: 48 8d 15 ab 0e 20 00    lea    0x200eab(%rip),%rdx      # 601000
<Message>
400155: 0f b6 04 10              movzbl (%rax,%rdx,1),%eax
400159: 83 e8 20              sub    $0x20,%eax
40015c: 89 c1                  mov    %eax,%ecx
40015e: 8b 45 fc              mov    -0x4(%rbp),%eax
400161: 48 98                  cltq
400163: 48 8d 15 96 0e 20 00    lea    0x200e96(%rip),%rdx      # 601000
<Message>
40016a: 88 0c 10              mov    %cl,(%rax,%rdx,1)
40016d: 83 45 fc 01              addl   $0x1,-0x4(%rbp)
400171: 8b 45 fc              mov    -0x4(%rbp),%eax
400174: 48 98                  cltq
400176: 48 8d 15 83 0e 20 00    lea    0x200e83(%rip),%rdx      # 601000
<Message>
40017d: 0f b6 04 10              movzbl (%rax,%rdx,1),%eax
400181: 84 c0                  test   %al,%al
400183: 75 9c                  jne    400121 <upper+0x39>
400185: 48 8d 05 74 0e 20 00    lea    0x200e74(%rip),%rax      # 601000
<Message>
40018c: 5d                  pop    %rbp
40018d: c3                  retq

```

从 `4000ec` 到 `4000fc` 分别是6个参数的输入，使用的寄存器按先后顺序分别是（对应的64位寄存器） `rdi`、`rsi`、`rdx`、`rcx`、`r8`、`r9`。

继续观察发现，C语言使用的局部变量，事实上就是在使用对应的堆栈区域（使用`rbp`寄存器 + 偏移量）。所以变量初始化，就是给堆栈相应区域赋值。

与局部变量不同的是，C语言使用全局变量，用的是 `rip` + 偏移地址的方式，使用的是 `601000` 这部分内存地址。

函数的返回值部分，看 `400185` 的位置，直接使用寄存器 `rax` 存储返回值。

```

1.elf:      file format elf64-x86-64

SYMBOL TABLE:
000000000004000e8 l      d  .text  0000000000000000 .text

```

```

00000000000400190 l    d .eh_frame 0000000000000000 .eh_frame
00000000000601000 l    d .data 0000000000000000 .data
0000000000000000000 l    d .comment 0000000000000000 .comment
0000000000000000000 l    df *ABS* 0000000000000000 1.c
00000000000601000 g    O .data 0000000000000000a Message
0000000000000000000 *UND* 0000000000000000 _start
0000000000060100a g    .data 0000000000000000 __bss_start
000000000004000e8 g    F .text 0000000000000000a6 upper
0000000000060100a g    .data 0000000000000000 __edata
00000000000601010 g    .data 0000000000000000 __end

```

从这个符号列表，就可以看清楚C语言对于全局变量（即数据区存放位置）的详情。第三行的 `.data` 部分就是数据区位置——`601000`，因为虚拟内存的缘故，他们与代码区不在同一页，链接的时候需要非常注意这一点。

综上所述，最终确定的汇编与c编译器组合是：64位下的gcc和ld。（因为我的目标平台是长模式，使用64位编译正好与我的目标平台吻合）。

同时，gcc默认是AT&T风格的汇编代码，为了一致性。我将在这个实验和接下来的实验里均使用AT&T风格的汇编。

## (2) 写一个汇编程和c程序混合编程实例

由于我这套组合环境是64位长模式，而dos是16位实模式，所以我选择用Linux代替Dos进行演示。

对应的字符串：`I am hungry now! Come and bring me some eat`

### 对应代码编写

```

.section .data
str: .asciz "I am hungry now! Come and bring me some eat"
output: .ascii " "

.section .text
.extern Count
.global _start
_start:
# Count(a, str)
mov $'a', %rdi
mov $str, %rsi
call Count

lea 0x30(%rax), %rax      # +0x30是为了转换为ascii
mov %al, output(%rip)     # 使用相对于rip的寻址方式

# write(fd, str, len) 写入stdout文件就是输出
mov $1, %rax      # system call 1 write
mov $1, %rdi      # fd = 1 stdout
lea output(%rip), %rsi
mov $1, %rdx      # len = 1

```

```

syscall

# exit(0)
mov $60, %rax    # system call 60 exit 返回Linux系统
xor %rdi, %rdi  # return 0;
syscall

```

上面用到的C语言函数定义如下：

```

/* 统计字符ch在字符串str出现的次数 */
unsigned long long Count(char ch, char *str) {
    int i = 0, res = 0;
    while (str[i] != '\0') {
        if (str[i++] == ch) {
            res++;
        }
    }
    return res;
}

```

## 实验结果

编译与链接过程：

```

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master x [0:21:11]
→ gcc -c AsCount.S -o AsCount.o

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master x [0:23:59]
→ gcc -c count.c -o count.o

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master x [0:24:05]
→ ld AsCount.o count.o -o count

```

结果：

对应的字符串： I am hungry now! Come and bring me some eat

统计'a'的输出次数：

```

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master x [0:24:08]
→ ./count
3%

```

统计'e'的输出次数：

```

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/test on git:master x [0:26:00]
→ ./count
4%

```

统计't'的输出次数：

```
# ouyhan @ DESKTOP-3TMC71 in ~/Code Workplace/os/Lab3/test on git:master ✘ [0:26:58]
└─ ./count
1%
```

以上结果均是正确的。

### (3) 混合编程重写系统内核

#### 代码的编写

这里的代码主要由三部分

- boot 引导程序
- kernal 系统内核
- user\_program 用户程序

代码的编写还是采用的VS Code，从下面这张图可以看到，我对每部分都有一个专门的文件夹，便于管理源码

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure under "LAB3 [WSL: UBUNTU]". It includes:
  - boot: boot.S
  - kernel: user1.c, user2.c, user3.c, user4.c
  - user program: user1.c, user2.c, user3.c, user4.c
- Code Editor:** The main editor window displays assembly code for boot.S. The code initializes memory, sets up segment registers (CS, DS, ES, SS), and performs disk reads. It also handles interrupt management and enables A20.
- Output Panel:** Shows the command line output from the terminal, indicating successful compilation and linking.
- Status Bar:** Displays the file number (141), line number (11), character count (4), encoding (UTF-8), and file type (x86 and x64 Assembly).

#### 代码的编译和链接

编译和链接的指令都按照（1）中的来。

```
gcc -N -ffreestanding -c [input file] -o [output file]
ld --oformat binary -Ttext [.text target location] -Tdata [.data target location]
[input_1 input_2 ..] -o [output file]
```

在编译SetPageTable的时候，我先把它生成汇编文件SetPageTable.S [warning是因为我直接给指针赋值地址]

```

# ouyhlun @ DESKTOP-3T1MC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:54:47]
- gcc -N -ffreestanding -S SetPageTable.c -o SetPageTable.S
SetPageTable.c: In function 'SetPageTable':
SetPageTable.c:1:19: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  1 | #define PML4_BASE 0x100000
   |
SetPageTable.c:4:32: note: in expansion of macro 'PML4_BASE'
  4 |     unsigned long long *PML4 = PML4_BASE;
   |
SetPageTable.c:10:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 10 |     unsigned long long *pdpl_kern = 0x200000;
   |
SetPageTable.c:14:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 14 |     unsigned long long *pdpl_user = 0x300000;
   |
SetPageTable.c:17:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 17 |     unsigned long long *pdpl_kern = 0x400000;
   |
SetPageTable.c:21:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 21 |     unsigned long long *pdpl_user = 0x500000;
   |
SetPageTable.c:25:35: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 25 |     unsigned long long *pt_kern = 0x600000;
   |
SetPageTable.c:33:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 33 |     unsigned long long *kern_code = 0x800027;
   |
SetPageTable.c:35:20: warning: assignment to 'long long unsigned int' from 'long long unsigned int *' makes integer from pointer without a cast [-Wint-conversion]
 35 |         pt_kern[i] = kern_code;
   |
SetPageTable.c:39:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 39 |     unsigned long long *pt_user_1 = 0x700000;
   |
SetPageTable.c:52:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 52 |     unsigned long long *pt_user_2 = 0xb00000;
   |
SetPageTable.c:59:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 59 |     unsigned long long *idt_addr = 0x900000;
   |

```

打开生成的SetPageTable.S，可以看到，其实他和我们正常编写的汇编语言差不多，只是多了一些链接语句.cfi\_def\_cfa\_offset这些。删去后，就是完整的汇编文件。

```

1  .file  "SetPageTable.c"
2  .text
3  .globl SetPageTable
4  .type  SetPageTable, @function
5  SetPageTable:
6  .LFB0:
7      .cfi_startproc
8      pushq %rbp
9      .cfi_offset %rbp, 16
10     .cfi_offset 6, -16
11     movq %rsp, %rbp
12     .cfi_offset %rbp, 6
13     subq $40, %rsp
14     movq $1048576, -88(%rbp)
15     movq -88(%rbp), %rax
16     movq $3145767, (%rax)
17     movq -88(%rbp), %rax
18     addq $4088, %rax
19     movq $2097191, (%rax)
20     movq -88(%rbp), %rax
21     addq $2048, %rax
22     movq $2097191, (%rax)
23     movq $2097152, -96(%rbp)
24     movq -96(%rbp), %rax
25     movq $4194343, (%rax)
26     movq -96(%rbp), %rax
27     addq $4088, %rax
28     movq $4194343, (%rax)
29     movq $3145728, -104(%rbp)
30     movq -104(%rbp), %rax
31     movq $5242919, (%rax)
32     movq $4194304, -112(%rbp)
33     movq -112(%rbp), %rax
34     movq $6291495, (%rax)
35     movq -112(%rbp), %rax
36     addq $4088, %rax
37     movq $6291495, (%rax)
38     movq $5242880, -120(%rbp)
39     movq -120(%rbp), %rax
40     movq $7340071, (%rax)
41     movq -120(%rbp), %rax
42     addq $16, %rax
43     movq $11534375, (%rax)
44     movq $6291456, -128(%rbp)
45     movq $10485799, -8(%rbp)
46     movl $0, -12(%rbp)
47     jmp .L2

```

把删去后的代码贴到引导代码里即可（为什么这么做会在后面排错过程的时候解释）。

## 编译引导文件boot

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:55:41]
+ gcc -N -ffreestanding -I -c ./ boot.S -o boot.o
boot.S:1:10: fatal error: asm.h: No such file or directory
  1 | #include <asm.h>
   |           ^
compilation terminated.

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:56:49] C:1
+ gcc -N -ffreestanding -I ./ -c boot.S -o boot.o
boot.S: Assembler messages:
boot.S:152: Warning: indirect jmp without `*'

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:56:57]
+ gcc -N -ffreestanding -c puts.c -o puts.o
puts.c: In function 'putc':
puts.c:63:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  63 |     unsigned char *vga_addr = 0xb8000;
               ^~~~~~

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:57:16]
+ ld --oformat binary -Ttext 0x7c00 boot.o puts.o -o boot2.bin

```

在链接的时候，我还生成了一份没有加 `--oformat binary` 命令的文件，这样生成的文件是ELF格式的，可以利用objdump进行反汇编，方便后面调试的时候看。

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master ✘ [10:57:54]
+ ld -Ttext 0x7c00 boot.o puts.o -o boot3.bin

```

## 编译用户程序文件user\_program

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [10:58:44]
+ gcc -N -ffreestanding -c user2.c -o user2.o
user2.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  12 | char *vga_addr = 0xb8000;
               ^~~~~~

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [10:59:19]
+ gcc -N -ffreestanding -c user3.c -o user3.o
user3.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  12 | char *vga_addr = 0xb8000;
               ^~~~~~

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [10:59:25]
+ gcc -N -ffreestanding -c user4.c -o user4.o
user4.c:12:18: warning: initialization of 'char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  12 | char *vga_addr = 0xb8000;
               ^~~~~~

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [10:59:30]
+ ld --oformat binary -Ttext 0x400000 -Tdata 0x400580 user2.o -o user2.bin

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [11:00:15]
+ ld --oformat binary -Ttext 0x400000 -Tdata 0x400580 user3.o -o user3.bin

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/user_program on git:master ✘ [11:00:20]
+ ld --oformat binary -Ttext 0x400000 -Tdata 0x400580 user4.o -o user4.bin

```

利用winhex软件，把生成的二进制文件放入我们的磁盘映像文件。

## 复制到磁盘映像文件后的结果

## 编译系统内核文件

```
# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [11:06:46]
→ gcc -N -ffreestanding -c driver.c -o driver.o
driver.c: In function 'SetInterrupt':
driver.c:42:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  42 |     unsigned long long *idt_addr = 0x100000;
                 ^~~~~~
driver.c: In function 'ScrollUp':
driver.c:145:32: warning: initialization of 'short unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 145 |     unsigned short *vga_addr = 0xb8000;      /* 两个字节为基本单位 */
                 ^~~~~~
driver.c: In function 'putc':
driver.c:155:31: warning: initialization of 'unsigned char *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 155 |     unsigned char *vga_addr = 0xb8000;
                 ^~~~~~

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [11:06:51]
→ gcc -N -ffreestanding -c main.S -o main.o

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [11:06:54]
→ ld --oformat binary -Ttext 0xfffff800000000000 -Tdata 0xfffff800000000C00 main.o 1.o driver.o -o main1.bin

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [11:06:56]
→ ld -Ttext 0xfffff800000000000 -Tdata 0xfffff800000000C00 main.o 1.o driver.o -o main2.bin
```

最终把所用代码都打包到硬盘映像文件后：

## 使用虚拟机Bochs运行

为了验证我按照实验方案成功进入长模式，我将利用bochsdbg的单步调试，通过展示每个阶段寄存器的值，按照下面这个图来确定现在CPU所处的模式

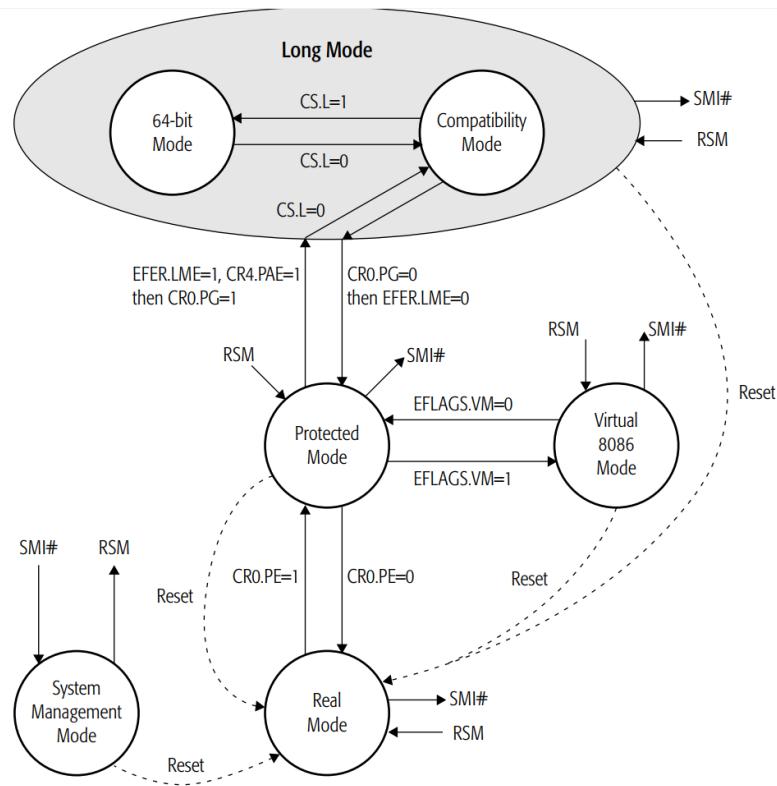


Figure 1-6. Operating Modes of the AMD64 Architecture

## 配置bochssrc

这里面有个注意的地方，需要修改

```
boot: a
```

为

```
boot: c
```

这样才能正常使用硬盘作为引导扇区。

## 引导程序进入保护模式

这里我直接使用objdump反汇编引导程序（右边），作为调试（左边）的依据。

The screenshot shows the Bochs for Windows interface. On the left, the BIOS setup screen displays various system configuration options. On the right, a terminal window titled 'boot' shows the assembly code of the boot loader. The code is in Intel hex format, showing instructions for setting up memory tables and entering protected mode.

```

0010543201[BIOS] ] BAR #4: I/o base address = 0xc020
0010549361[BIOS] ] region 4: 0x0000<0x20
0010550701[BIOS] ] new IRQ line = 0
0010550702[BIOS] ] IRQ line 0 mapped to 0x0b: vendor_id=0x0088 device_id=0x7113 class=0x0600
0010572361[ACPI] ] new IRQ line = 11
0010572501[ACPI] ] new IRQ line = 9
0010572771[ACPI] ] new PM base address: 0x0000
0010573191[PCI] ] new PCI base address: 0xb000
0010573191[PCI] ] Setting SWRAN control register to 0x4a
00102214121[CPU0] ] Enter to System Management Mode
00102214121[CPU0] ] enter_system_management_mode: temporary disable VMX while in SMM mode
00102214211[CPU0] ] KSM: Release from System Management Mode
00102214211[CPU0] ] Disabling SWRAN control register to 0x0a
0014121591[BIOS] ] MP table addr=0x000fe9e0 MPC table add=0x000f9d0 size=0xc8
0014140301[BIOS] ] SMBIOS table add=0x000f9ea0
0014162161[BIOS] ] ACPI table: RSDP add=0x000ff4d0 ACPI DATA add=0x01ff0000 size=0xffff
0014219431[BIOS] ] 1440WX FMC write to PAM register 59 (TLB Flush)
0014226661[BIOS] ] bios_table_cus add: 0x000ff9ff
0010551371[VBIOS] ] VGA Bios $Id: vgbios.c 226 2020-01-02 21:36:23Z vruppert $
0010551401[EXTRA] ] VBE known display Interface Version 0.5
0010551401[EXTRA] ] VBE known display Interface Version 0.6
0010551401[VBIOS] ] VBE Bios $Id: vbe.c 228 2020-01-02 23:09:02Z vruppert $
00109019701[BIOS] ] ata0-0: PCHS-20/16/63 translation:none LCHS-20/16/63
0082491000[BIOS] ] Boot time on: 00:00:00
001047871[BIOS] ] Booting from 0000:7c00
D: Breakpoint 1, 0x0000000000007c00 in ?? ()  

rxt at t+17404242  

D: [0x0000000000000000]:0000:7c00 (unkn. ctxt): mov ax, cs ; Bcc8  

rcxh:3  

代码的编译和链接

```

```

ouyilan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [11:23:49]
$ objdump -d boot3.bin

boot3.bin:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000<_start>:
    7c00:   b8 c8          mov    %cs,%eax
    7c02:   8e c0          mov    %eax,%es
    7c04:   bb 00 7e b8 04  mov    $0x4b87e00,%ebx
    7c09:   02 b0 80 00 b9 02  add    0x2b90080(%rdx),%bx
    7c0f:   00 cd          add    %cl,%ch
    7c11:   13 fa          adc    %edx,%edi
    7c13:   fc              cld
    7c14:   31 c0          xor    %eax,%eax
    7c16:   8e d8          mov    %eax,%ds
    7c18:   8e c0          mov    %eax,%es
    7c1a:   8e d0          mov    %eax,%ss
0000000000000000<cseta20.1>:
    7c1c:   e4 64          in    $0x64,%al
    7c1e:   a8 02          test   $0x2,%al
    7c20:   75 fa          jne   7c1c <cseta20.1>
    7c22:   b0 d1          mov    $0xd1,%al
    7c24:   e6 64          out   %al,$0x64

```

看右边的反汇编代码可知，**0x7c41** 是进入保护模式的最后一步。

```

    7c14: 31 c0 xor %eax,%eax
    7c16: 8e d8 mov %eax,%ds
    7c18: 8e c0 mov %eax,%es
    7c1a: 8e d0 mov %eax,%ss

00000000000000007c1c <seta20.1>:
    7c1c: e4 64 in $0x64,%al
    7c1e: a8 02 test $0x2,%al
    7c20: 75 fa jne 7c1c <seta20.1>
    7c22: b0 d1 mov $0xd1,%al
    7c24: e6 64 out %al,$0x64

00000000000000007c26 <seta20.2>:
    7c26: e4 64 in $0x64,%al
    7c28: a8 02 test $0x2,%al
    7c2a: 75 fa jne 7c26 <seta20.2>
    7c2c: b0 df mov $0xdf,%al
    7c2e: e6 60 out %al,$0x60
    7c30: 0f 01 16 lgdt (%rsi)
    7c31: 1c 7d sbb $0x7d,%al
    7c35: 0f 01 1e lidt (%rsi)
    7c38: 22 7d 0f and $0xf(%rbp),%bh
    7c3b: 20 c0 and %al,%al
    7c3d: 66 83 c8 01 or $0x1,%ax
    7c41: 0f 22 c0 mov %rax,%cr0
    7c44: ea (bad)
    7c45: 49 7c 08 rex.WB jl 7c50 <protcseg+0x7>

...
00000000000000007c49 <protcseg>:

```

执行那段代码的前后，CR0的pe位发生了变化（从0->1），可以证明，我们现在已经进入了保护模式

```

(0) [0x00000000007c41] 0000:7c41 (unk. ctxt): mov cr0, eax ; 0f22c0
<bochs:5> creg
CR0=0x60000010: pg CD NW ac wp ne ET ts em mp pe
CR2=page fault laddr=0x0000000000000000
CR3=0x000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: cet pke smap smep osxsavc pcid fsgsbase smx vmx osxmmexcpt umip osfxsr pce pge mce pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffbsr nxe lma lme sce
<bochs:6> n
Next at t=17406559
(0) [0x00000000007c44] 0000:00000000000000007c44 (unk. ctxt): jmpf 0x0008:7c49 ; ea497c0800
<bochs:7> creg
CR0=0x60000011: pg CD NW ac wp ne ET ts em mp PE
CR2=page fault laddr=0x0000000000000000
CR3=0x000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: cet pke smap smep osxsavc pcid fsgsbase smx vmx osxmmexcpt umip osfxsr pce pge mce pae pse de tsd pvi vme
CR8: 0x0
EFER=0x00000000: ffbsr nxe lma lme sce

```

正确设置好GDT后，CS段寄存器的值。下面的32-bit也证明了我们已经进入保护模式。

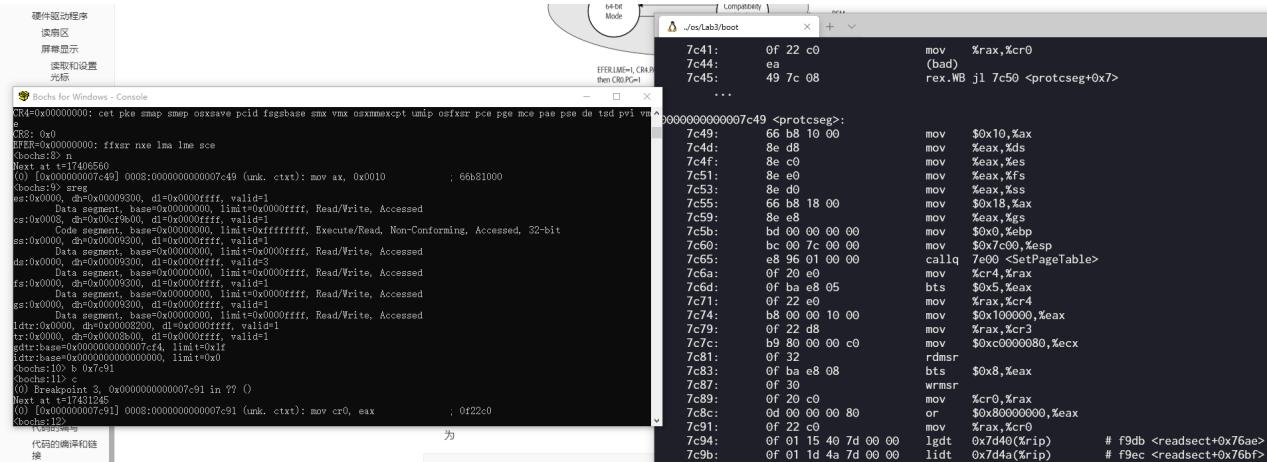
```

next at t=17406559
(0) [0x00000000007c49] 0008:00000000000000007c49 (unk. ctxt): mov ax, 0x0010 ; 66b81000
<bochs:9> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0008, dh=0x00cf9b00, dl=0x0000ffff, valid=1
    Code segment, base=0x00000000, limit=0xffffffff, Execute/Read, Non-Conforming, Accessed, 32-bit
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=3
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
gdtr:base=0x00000000000000007cf4, limit=0x1f
idtr:base=0x0000000000000000, limit=0x0

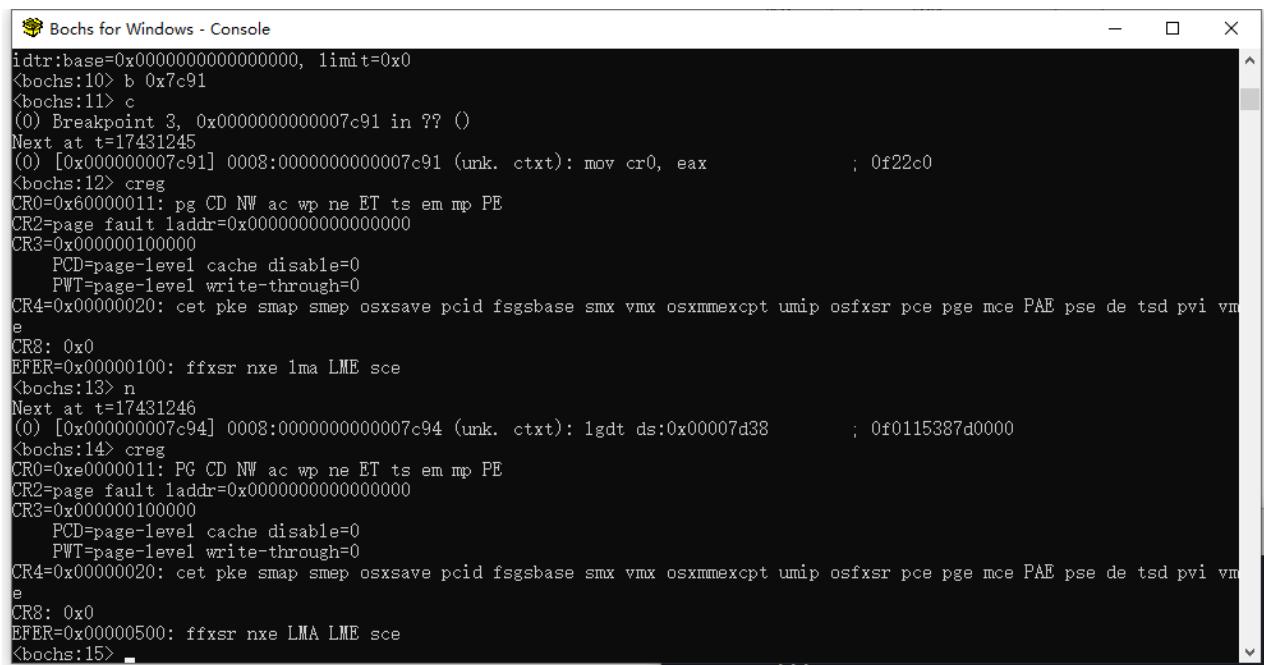
```

## 从保护模式走向长模式

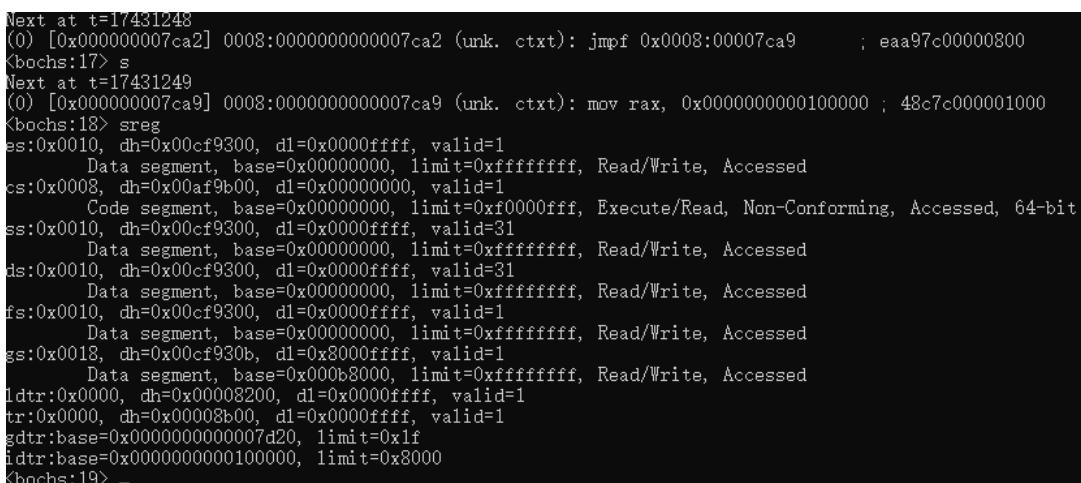
继续观察可看到 **0x7c91** 是进入长模式的最后一步



进入前后控制寄存器的前后对比图，按照上面的流程图，**CR4.PAE = 1**、**EFER.LME = 1**、**CR0.PG = 1**，这代表我们已经进入了长模式的兼容模式



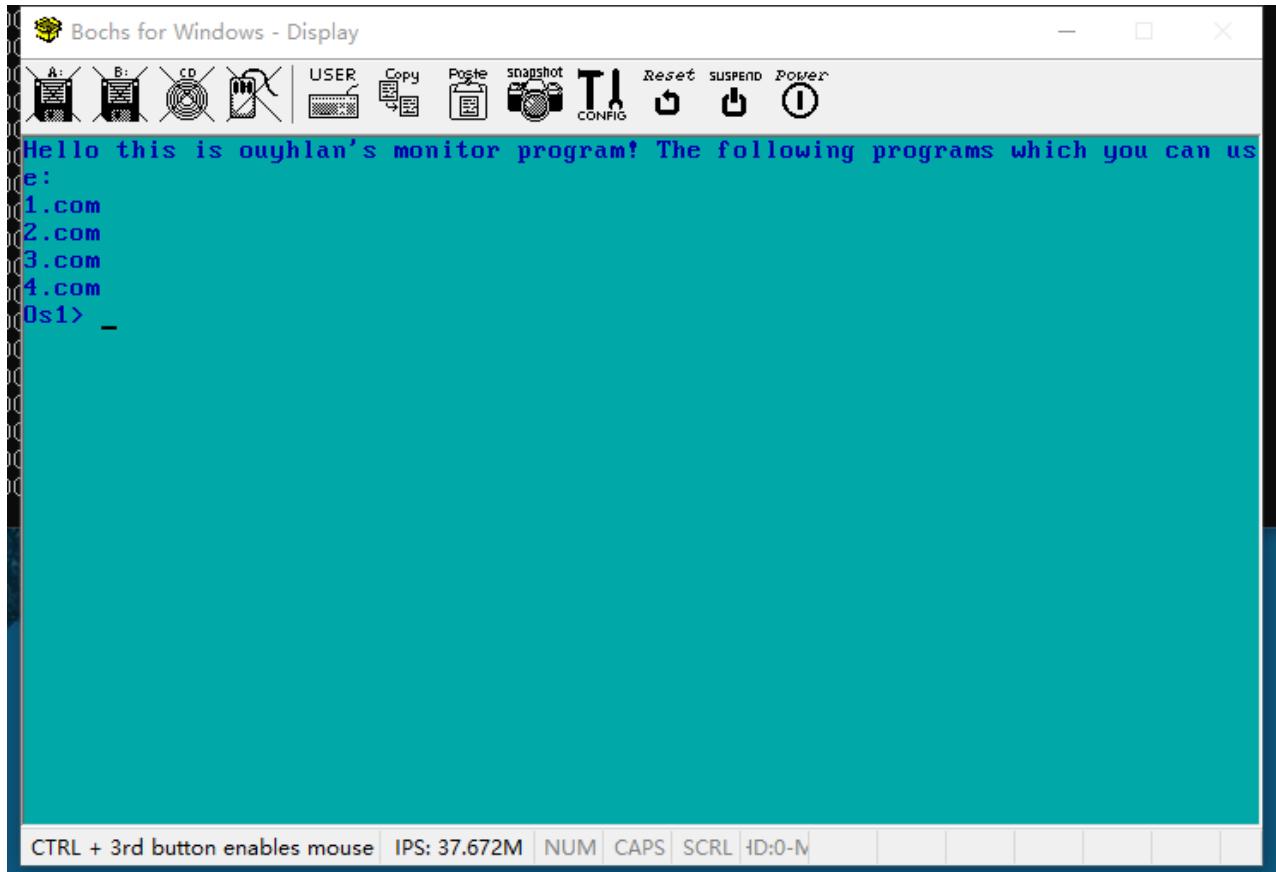
使用 **1jmp** 指令初始化段寄存器CS后，下方CS的值可以证明，我们已经进入了长模式的64-bit模式。



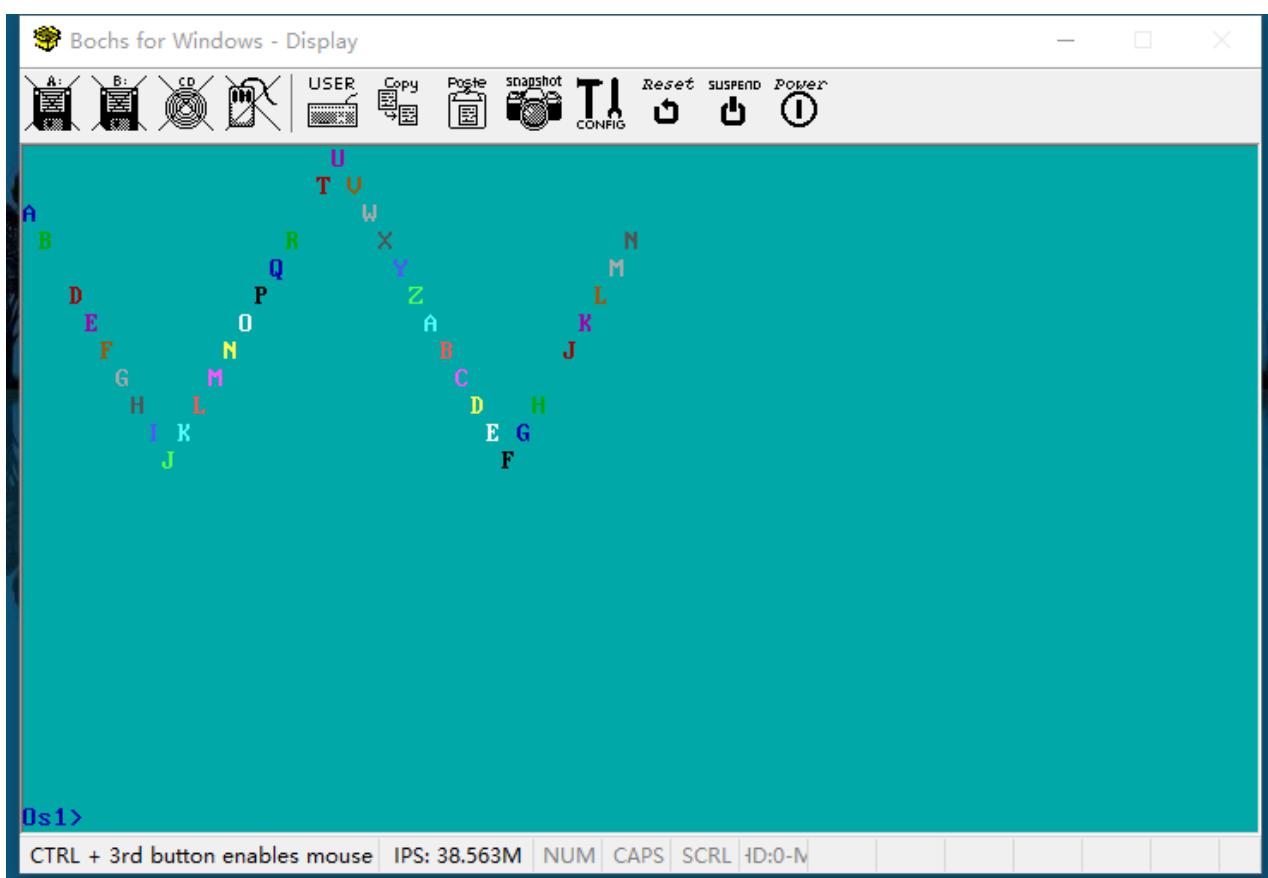
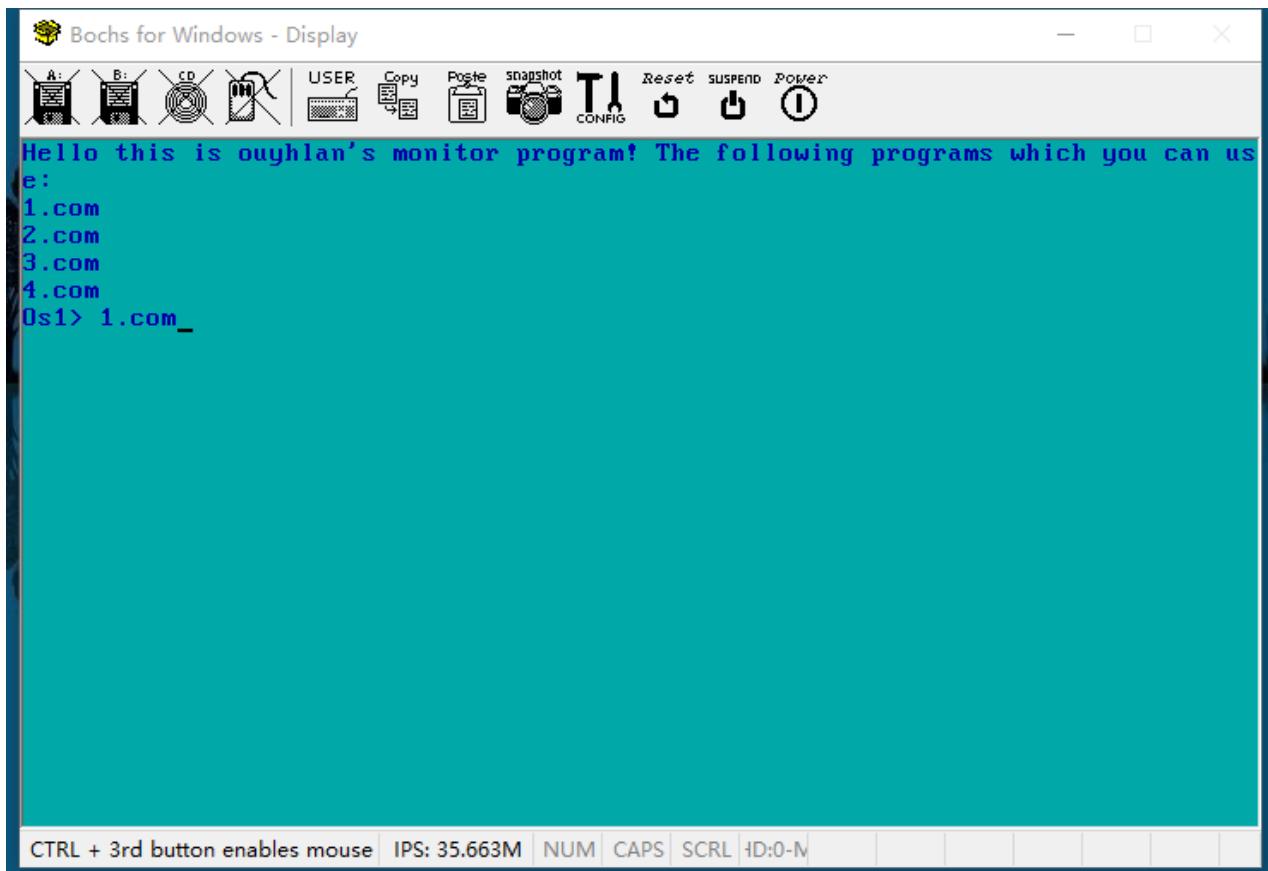
至此，整个引导程序已经完成了他的使命，接下来就是进入系统内核部分了。

## 进入系统内核

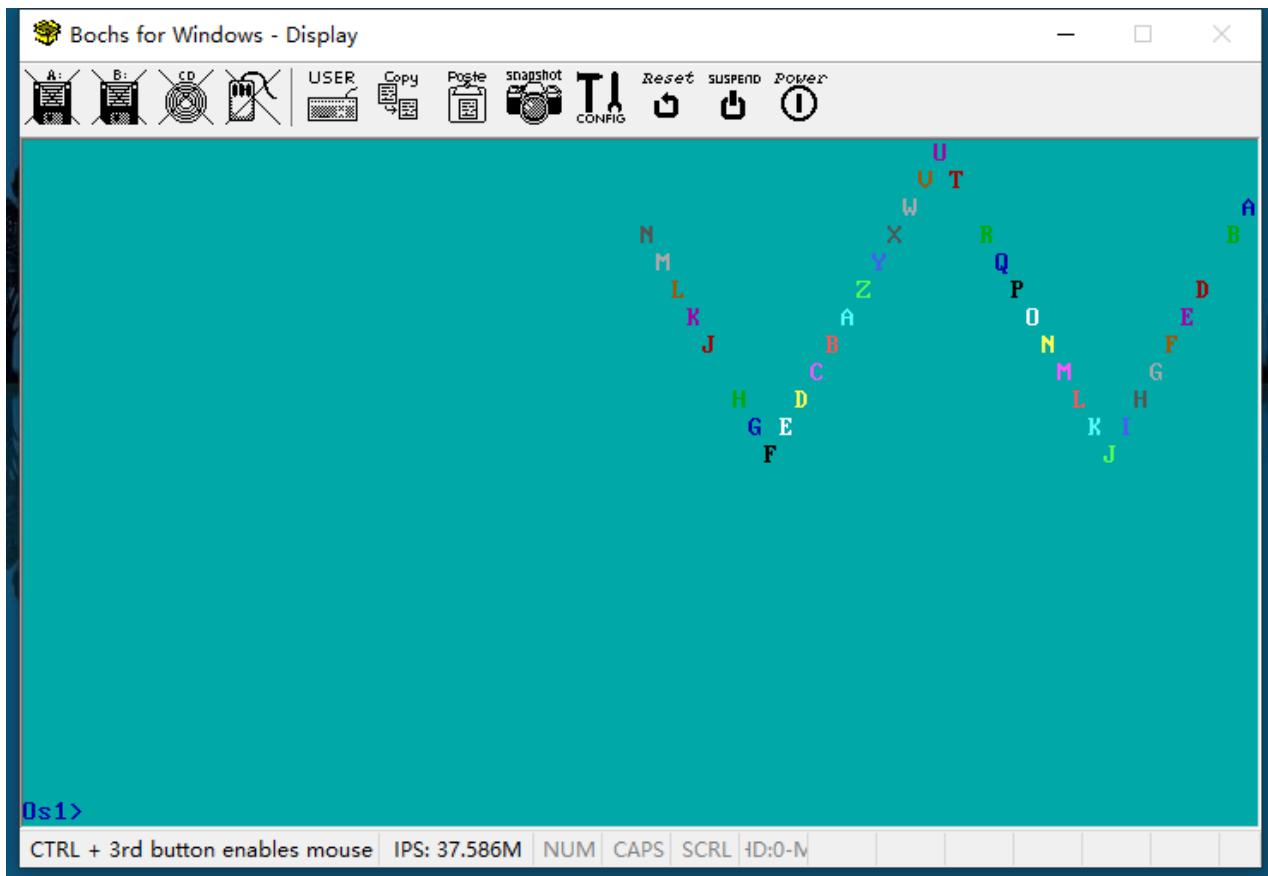
开始是一些提示信息，接下来有个输入提示符



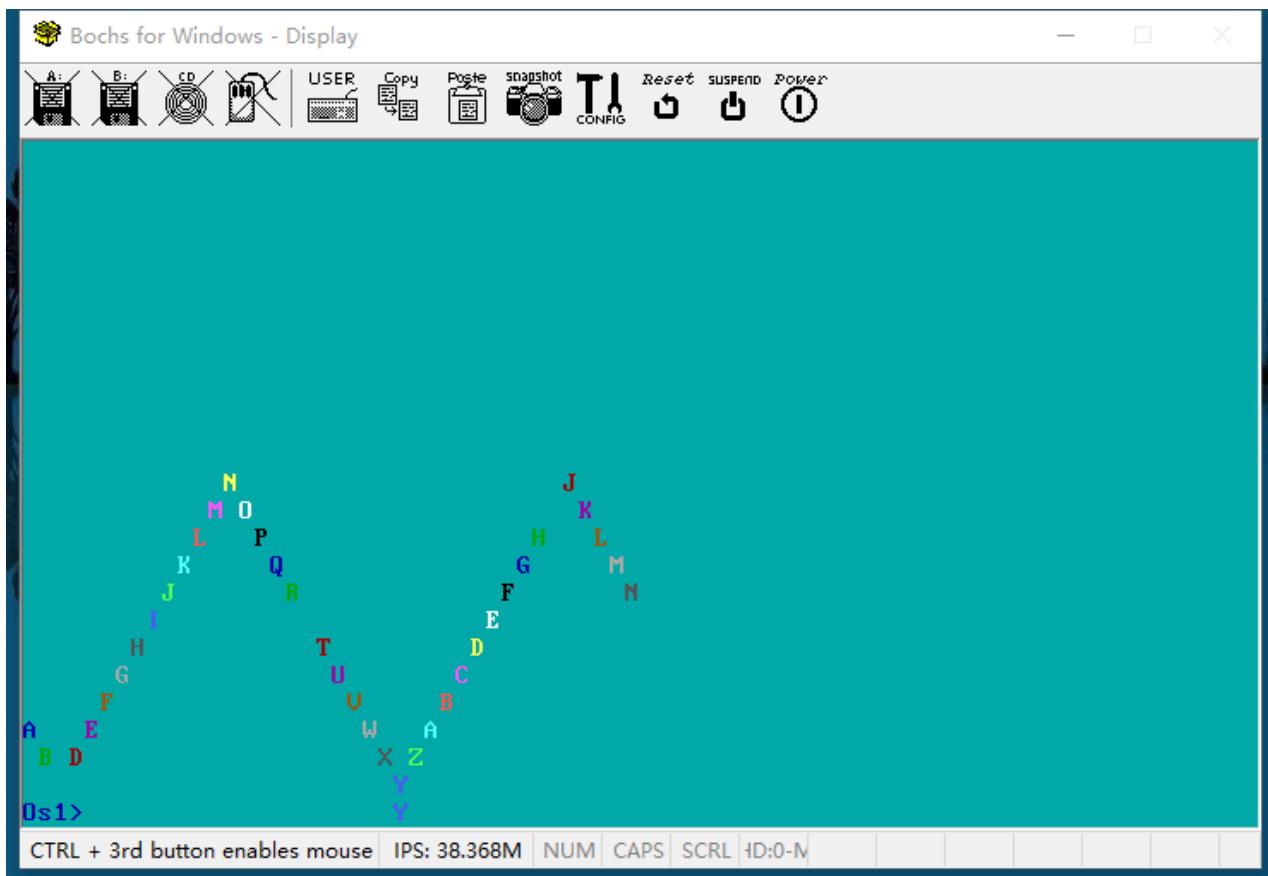
输入1.com执行第一个用户程序



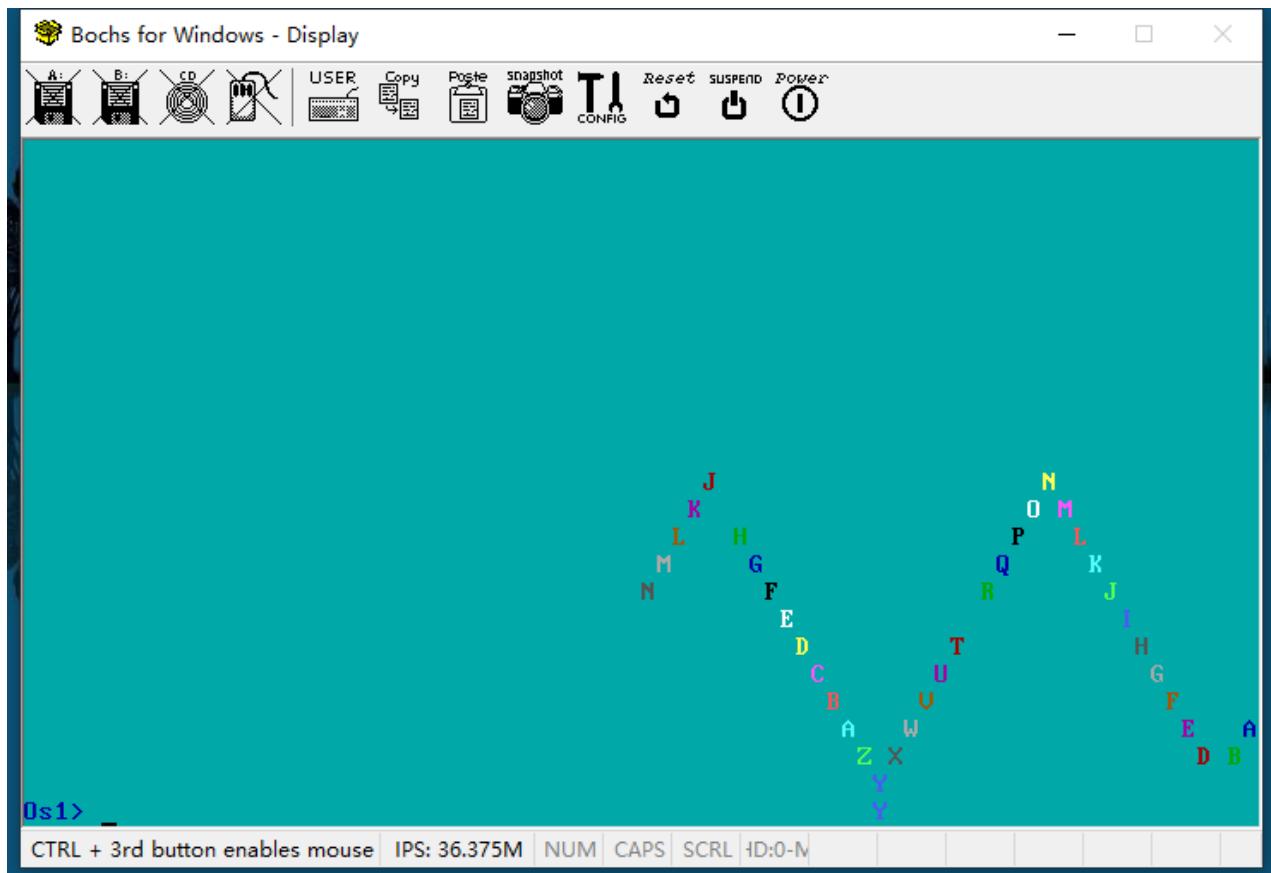
第二个用户程序



第三个用户程序

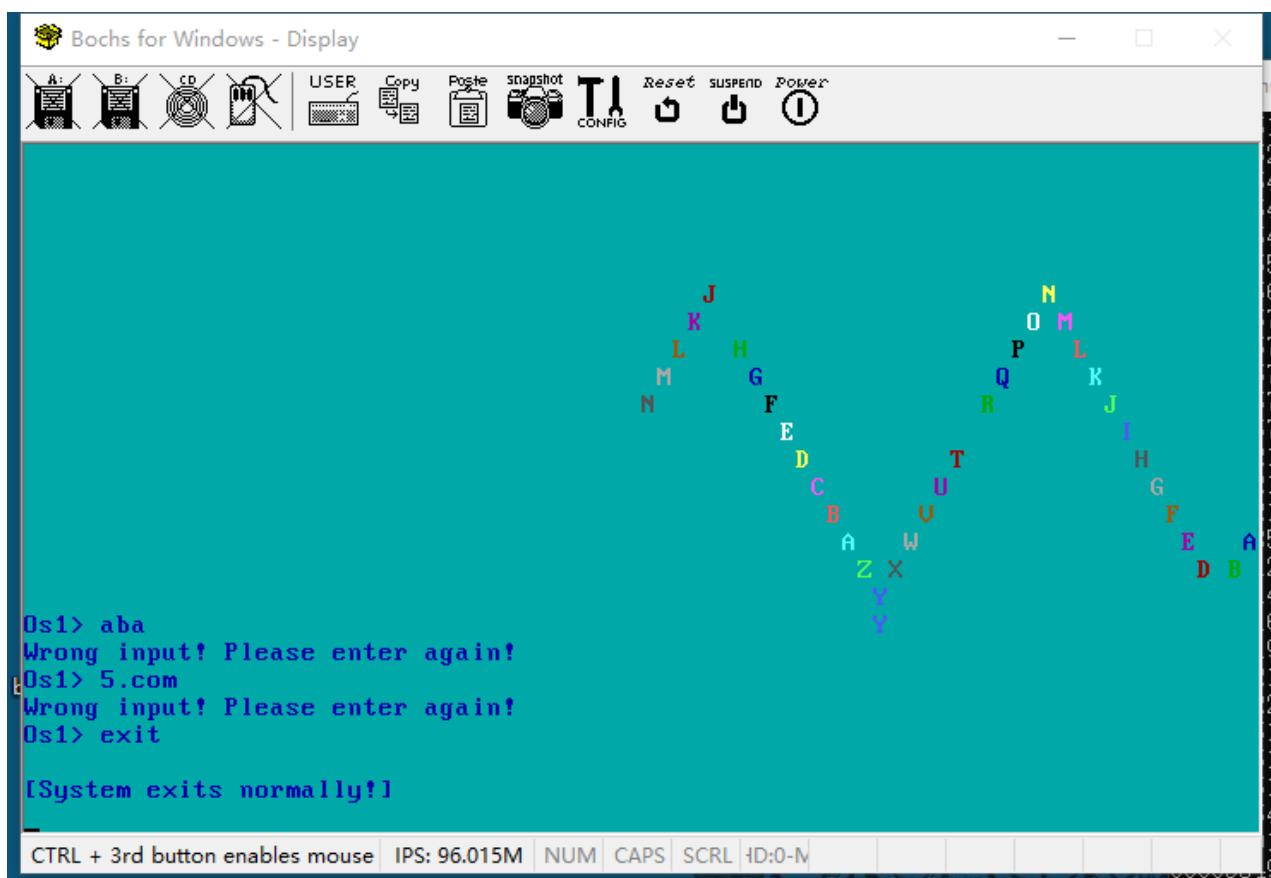


第四个用户程序



## (4) 扩展监控程序命令处理能力

这部分主要扩展的功能对于非法输入的处理



上面的aba和5.com都是目前系统里没有的功能，所以会提示用户重新输入。

最后，敲击exit退出本系统

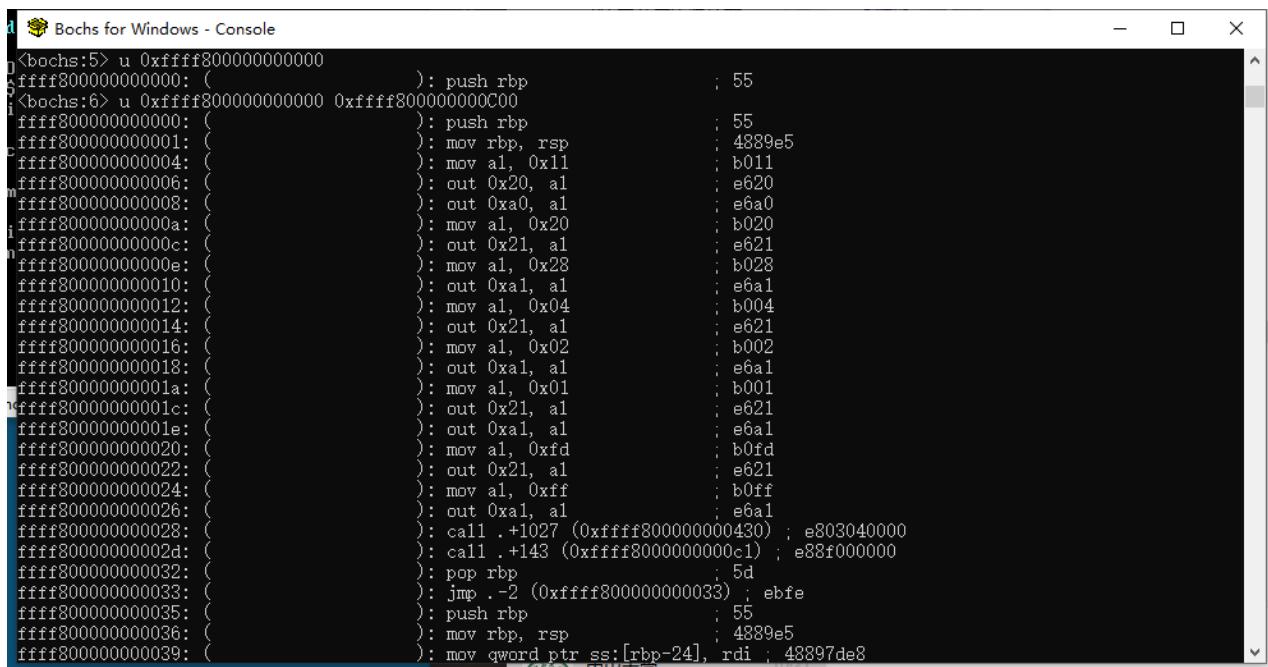
## (5) 重写引导程序，加载程序的独立内核。

这部分实际上已经在前面完成了，下面是boot.S里加载的代码：

```
.set LONG_MODE_SYSM,      0xfffff8000000000000          # 系统内核加载的内存地址
.set KERNEL_SEC,          0x5
.set NUM_KERNEL_SEC,      0x0a                         # 为了简便，先预读取16个扇区

# Loading Kernel Code in
movq $LONG_MODE_SYSM, %rdi
movq $KERNEL_SEC, %rsi
movq $NUM_KERNEL_SEC, %rdx
call readsect           # 调用读扇区功能
```

用bochs调试就可以看到加载后的情况了。



The screenshot shows the Bochs debugger console window. The command `<bochs:5> u 0xfffff8000000000000` has been entered, and the assembly code for loading the kernel is displayed. The code includes instructions for pushing rbp, mov rbp, rsp, mov al, out al, mov al, and calls to readsect. The assembly output is color-coded with orange for comments and brown for labels and numbers.

```
<bochs:5> u 0xfffff8000000000000
fffff8000000000000: (          ): push rbp      ; 55
<bochs:6> u 0xfffff8000000000000 0xfffff800000000C00
fffff8000000000000: (          ): push rbp      ; 55
fffff8000000000001: (          ): mov rbp, rsp   ; 4889e5
fffff8000000000004: (          ): mov al, 0x11    ; b011
fffff8000000000006: (          ): out 0x20, al   ; e620
fffff8000000000008: (          ): out 0xa0, al   ; e6a0
fffff800000000000a: (          ): mov al, 0x20    ; b020
fffff800000000000c: (          ): out 0x21, al   ; e621
fffff800000000000e: (          ): mov al, 0x28    ; b028
fffff8000000000010: (          ): out 0xa1, al   ; e6a1
fffff8000000000012: (          ): mov al, 0x04    ; b004
fffff8000000000014: (          ): out 0x21, al   ; e621
fffff8000000000016: (          ): mov al, 0x02    ; b002
fffff8000000000018: (          ): out 0xa1, al   ; e6a1
fffff800000000001a: (          ): mov al, 0x01    ; b001
fffff800000000001c: (          ): out 0x21, al   ; e621
fffff800000000001e: (          ): out 0xa1, al   ; e6a1
fffff8000000000020: (          ): mov al, 0xfd    ; b0fd
fffff8000000000022: (          ): out 0x21, al   ; e621
fffff8000000000024: (          ): mov al, 0xff    ; b0ff
fffff8000000000026: (          ): out 0xa1, al   ; e6a1
fffff8000000000028: (          ): call .+1027 (0xfffff800000000430) ; e803040000
fffff800000000002d: (          ): call .+143 (0xfffff80000000001) ; e88f000000
fffff8000000000032: (          ): pop rbp       ; 5d
fffff8000000000033: (          ): jmp .-2 (0xfffff800000000033) ; ebfe
fffff8000000000035: (          ): push rbp       ; 55
fffff8000000000036: (          ): mov rbp, rsp   ; 4889e5
fffff8000000000039: (          ): mov qword ptr ss:[rbp-24], rdi ; 48897de8
```

## (6) 拓展自己的软件项目管理目录

```
PS D:\Files\Os\Lab3\18340133_欧阳洁岗_实验三_v0\src> tree
卷 新加卷 的文件夹 PATH 列表
卷序列号为 2461-717C
D: .
├── boot
├── kern
└── test
└── user_program
```

现在看到，我的整个软件项目管理目录已经非常系统化了。

- boot 引导程序
- kern 操作系统内核
- test 一些测试小程序
- user\_program 用户程序

## 实验中遇到的问题

### (1) 编译和链接的命令不正确

```
# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [16:45:40]
→ gcc -c asm.h boot.S -o boot.o
gcc: fatal error: cannot specify '-o' with '-c', '-S' or '-E' with multiple files
compilation terminated.

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [16:45:58]
→ gcc -c boot.S -o boot.o
boot.S:1:10: fatal error: asm.h: No such file or directory
  1 | #include <asm.h>
    |
compilation terminated.

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [16:46:11] C:1
→ gcc boot.S -I ./ -o boot.o
/usr/bin/ld: /tmp/cc1hrbNP.o: relocation R_X86_64_16 against '.text' can not be used when making a PIE object; recompile with -fPIC
/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/Scrt1.o: In function '_start':
(.text+0x20): undefined reference to 'main'
/usr/bin/ld: final link failed: Invalid operation
collect2: error: ld returned 1 exit status

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [16:47:19] C:1
→ gcc -c boot.S -I ./ -o boot.o

# ouyhlant @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [16:53:07] C:1
→ ld -Ttext 0x7c00 -O binary boot.o -o boot.bin
ld: warning: cannot find entry symbol _start; defaulting to 00000000000000007c00
```

经过多次尝试，最终得出了我在试验中使用的命令。

```
gcc -N -ffreestanding -c [input file] -o [output file]
ld --oformat binary -Ttext [.text target location] -Tdata [.data target location]
[input_1 input_2 ..] -o [output file]
```

### (2) 链接中的问题

前面提到的，设置64位长模式的页表的时候，我们是还处于保护模式下，所以代码需要以32位的形式编译，所以我在下面加了[-m32]这个参数：

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [12:34:46]
→ gcc -m32 -c -N -ffreestanding SetPageTable.c -o SetPageTable.o
SetPageTable.c: In function 'SetPageTable':
SetPageTable.c:1:19: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
  1 | #define PML4_BASE 0x100000
   |
SetPageTable.c:4:32: note: in expansion of macro 'PML4_BASE'
  4 |     unsigned long long *PML4 = PML4_BASE;
   |
SetPageTable.c:10:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 10 |     unsigned long long *pdpl_kern = 0x200000;
   |
SetPageTable.c:14:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 14 |     unsigned long long *pdpl_user = 0x300000;
   |
SetPageTable.c:17:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 17 |     unsigned long long *ndl_kern = 0x400000;
   |
SetPageTable.c:21:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 21 |     unsigned long long *ndl_user = 0x500000;
   |
SetPageTable.c:25:35: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 25 |     unsigned long long *pt_kern = 0x600000;
   |
SetPageTable.c:33:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 33 |     unsigned long long *kern_code = 0x800027;
   |
SetPageTable.c:35:20: warning: assignment to 'long long unsigned int' from 'long long unsigned int *' makes integer from pointer without a cast [-Wint-conversion]
 35 |         pt_kern[i] = kern_code;
   |
SetPageTable.c:39:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 39 |     unsigned long long *pt_user_1 = 0x700000;
   |
SetPageTable.c:52:37: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 52 |     unsigned long long *pt_user_2 = 0xb00000;
   |
SetPageTable.c:59:36: warning: initialization of 'long long unsigned int *' from 'int' makes pointer from integer without a cast [-Wint-conversion]
 59 |     unsigned long long *idt_addr = 0x900000;
   |

```

但是，引导程序里有64位汇编的代码，所以我需要使用64位环境进行编译，这就引发了两者在链接时候的问题

```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/boot on git:master x [12:36:08]
→ ld boot.o SetPageTable.o puts.o -o boot4.bin
SetPageTable.o: In function 'SetPageTable':
SetPageTable.c:(.text+0x0): multiple definition of 'SetPageTable'
boot.o:(.text+0x200): first defined here
ld: i386 architecture of input file 'SetPageTable.o' is incompatible with i386:x86-64 output
boot.o: In function 'seta20.2':
(.text+0x33): relocation truncated to fit: R_X86_64_16 against '.text'
(.text+0x38): relocation truncated to fit: R_X86_64_16 against '.text'
(.text+0x45): relocation truncated to fit: R_X86_64_16 against '.text'
SetPageTable.o:(.text+0xc): undefined reference to '_GLOBAL_OFFSET_TABLE_'

```

上面显示i386架构与x86-64不兼容。

我尝试了许多办法，都没有解决这个问题，索性就直接编译成汇编文件，贴在引导程序代码里。

### (3) 链接时没指明数据段位置

一开始我的链接命令是这样的：

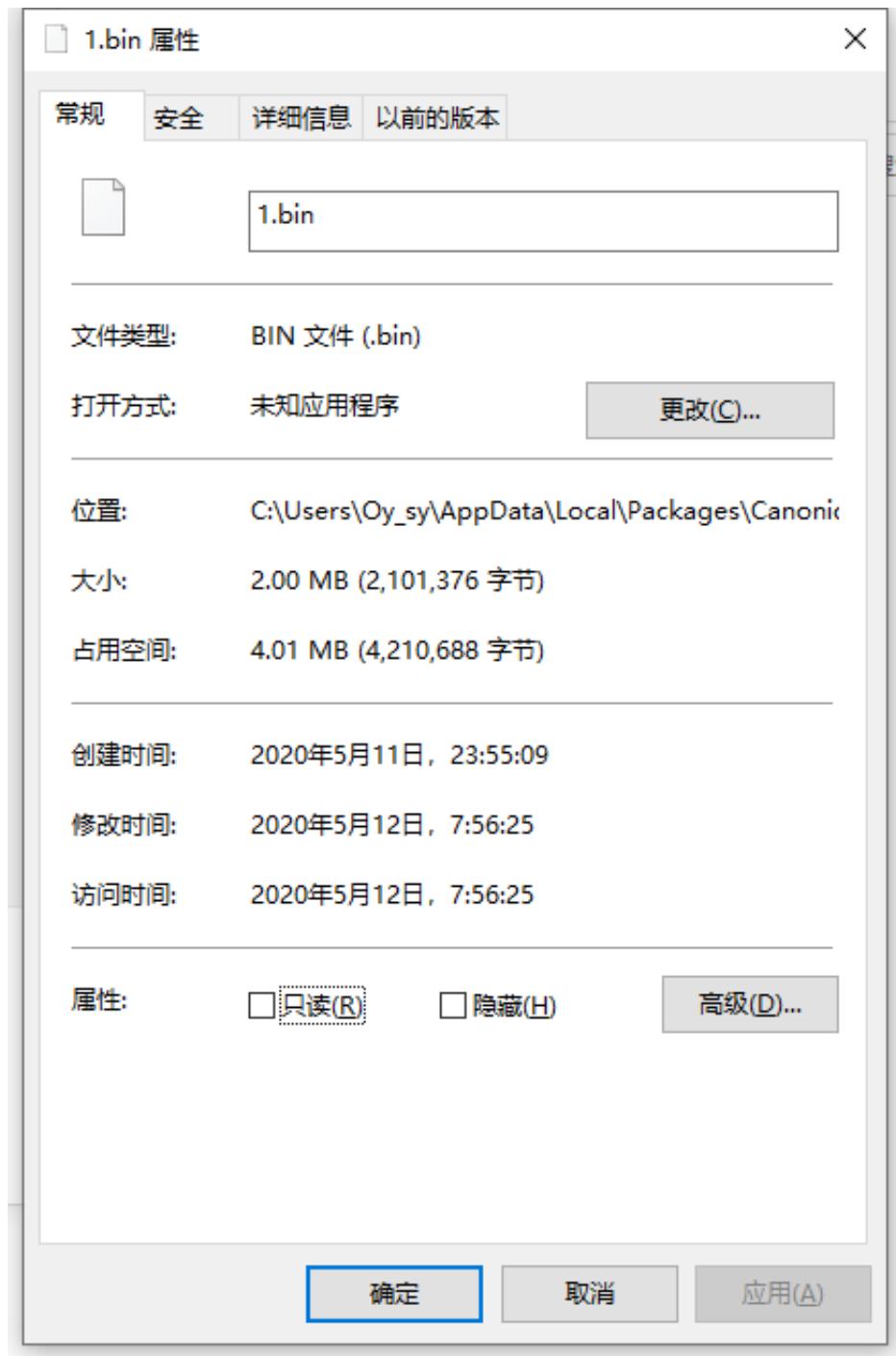
```

# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [7:56:22]
→ ld --oformat binary -Ttext 0xffffffff8000000000 main.o 1.o driver.o -o 1.bin

```

但是在调试过程中，发现程序会跳转到或者去读取到非法内存地址，这让我很诧异。

经过一番观察，我发现生成的bin文件异常的大：

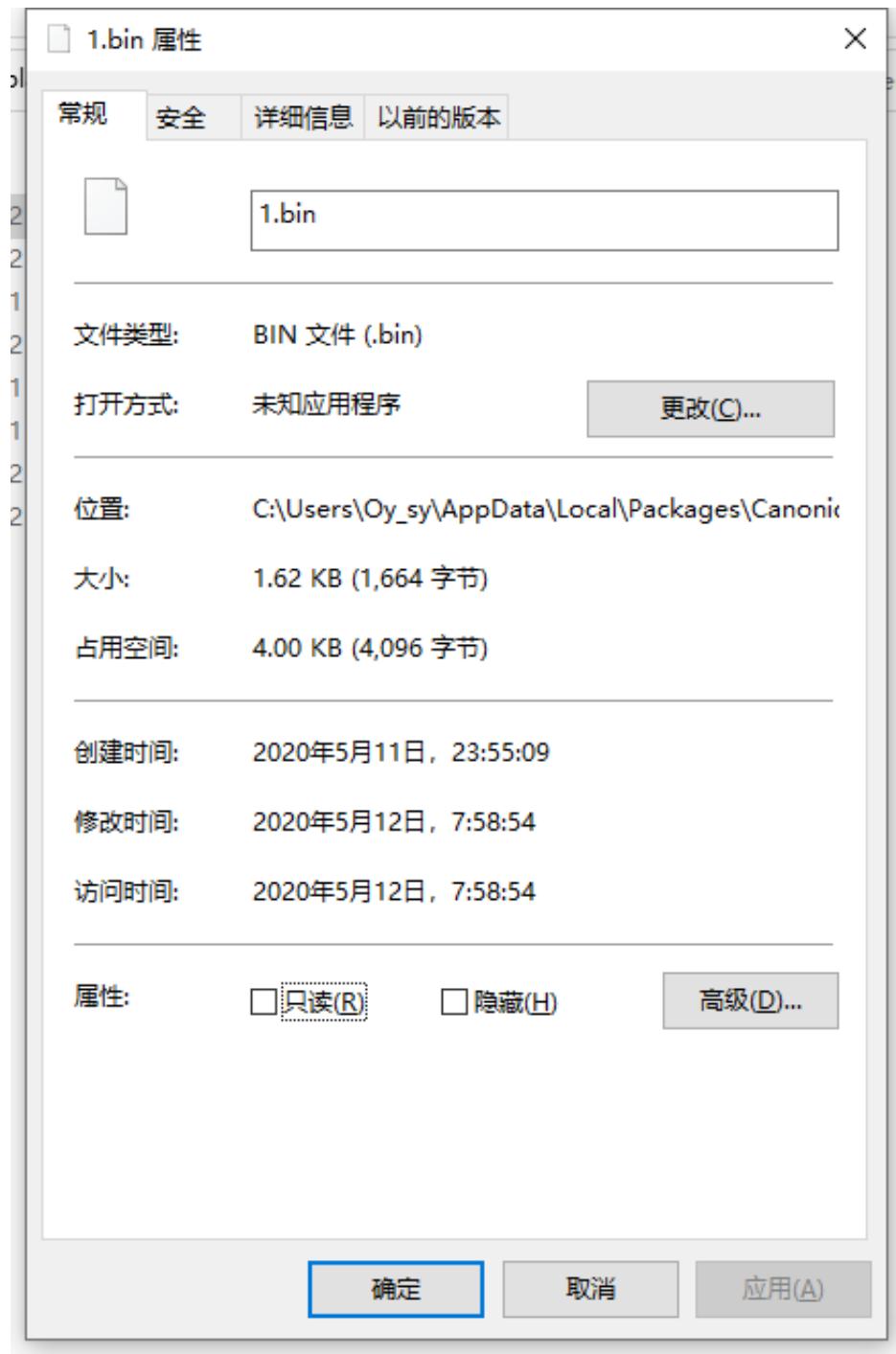


理论上，我的代码应该是以字节为单位，再大，也是KB为单位，但这个生成的确实MB为单位。

我再利用winhex和bochs调试发现，原来是ld在分配数据区空间的时候，默认是分配到另外一页。这样，经过查询ld官方文档，我修改了我的链接命令：

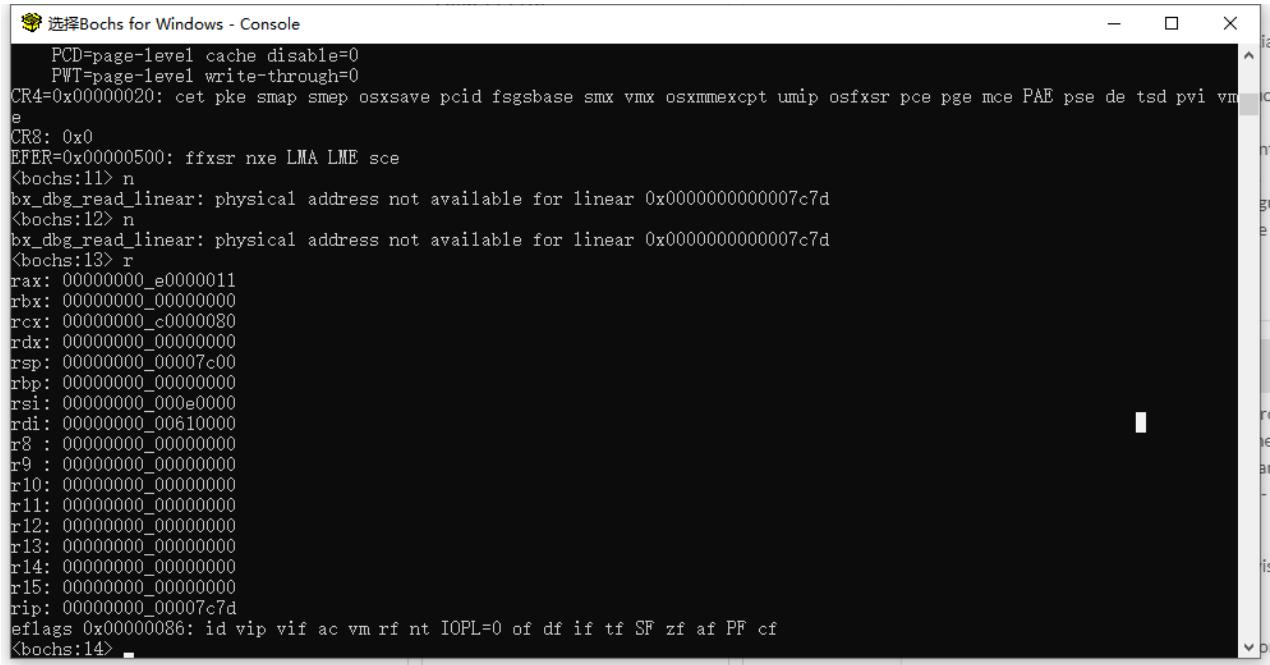
```
# ouyhan @ DESKTOP-3TIMC71 in ~/Code Workplace/os/Lab3/kern on git:master x [7:58:21] C:1
$ ld --oformat binary -Ttext 0xffffffff8000000000 -Tdata 0xffffffff8000000600 main.o 1.o driver.o -o 1.bin
```

生成文件大小变成了1.62KB，足足小了1000倍：



#### (4) 页表初始化不正确

根据AMD官方的文档，进入长模式一定要启用页式存储结构。而我在开始的时候，对于页表项还不够熟悉，填错了几项，导致进入长模式后立刻出错：



选择Bochs for Windows - Console

```
PCD=page-level cache disable=0
PWT=page-level write-through=0
CR4=0x00000020: cet pke smap smep osxsav e pcid fsgsbase smx vmx osxmmexcpt umip osfxsr pce pge mce PAE pse de tsd pvi vme
CR8: 0x0
EFER=0x00000500: ffxsr nxe LMA LME sce
<bochs:11> n
bx_dbg_read_linear: physical address not available for linear 0x0000000000007c7d
<bochs:12> n
bx_dbg_read_linear: physical address not available for linear 0x0000000000007c7d
<bochs:13> r
rax: 00000000_e0000011
rbx: 00000000_00000000
rcx: 00000000_c0000080
rdx: 00000000_00000000
rsp: 00000000_00007c00
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_00610000
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
r12: 00000000_00000000
r13: 00000000_00000000
r14: 00000000_00000000
r15: 00000000_00000000
rip: 00000000_00007c7d
eflags 0x00000086: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af PF cf
<bochs:14>
```

这个错误是致命的，这直接导致我的代码段也无法执行，因为当前rip的值对应的内存空间都没有映射。

## (5) 未完待续...

在进入长模式的时候，还有很多很多问题。但限于篇幅原因，这里就不再做进一步的阐述了。

# 实验总结

这次是我在操作系统实验课的第三个实验，这次实验对比上两次实验，有了飞跃性的跳转，可以说是质变。

分享一下为什么想要做长模式而不是在实模式或者保护模式下编写操作系统。事实上，保护模式是在 intel 80386 的时候已经提出了，而 AMD64 的长模式，也早在十几年前提出了。我觉得，现在我写的操作系统，应该紧跟时代的发展，了解更多更新的技术才行。同时，在 64-bit 长模式里，AMD 实际上基本抛弃了原来在实模式和保护模式下的分段式管理机制，而采用现代操作系统更加广泛使用的分页式管理机制。不仅如此，进入长模式，我可以更好地通过实验课，把理论课上学到的知识（例如，特权级、分页式内存管理、虚拟内存等等）这些都真正用代码实现一遍。这个我觉得是非常有意义的一件事情，学到的东西也会更多。

虽然网上有很多关于保护模式的文档，但是对于长模式的文档，却很少，大部分相关的博客都是建议查阅 AMD64 的官方手册《AMD64 Architecture Programmer's Manual: Volumes 1-5》。所以我通过查询手册，同时利用网上的资源 [https://wiki.osdev.org/Main\\_Page](https://wiki.osdev.org/Main_Page)，一步步进入长模式。不仅如此，在长模式里，我们原来的 BIOS 中断调用都没有了，需要自己动手操作 I/O 端口来实现一些硬件驱动。这部分对于现在的我来说，也是一件非常困难的事情。

限于资料的匮乏以及目前我理解能力的薄弱，我实在无法理解软盘加载的整个流程，我决定使用更加简单一点的硬盘作为磁盘映像文件（其实硬盘和软盘之间差别不大）。

实际上，这次实验我只用了一周的时间去做，由于时间的匮乏而要做的事情太多，我目前实现的功能都是不完善的，下面是我后面会去拓展的功能：

- 完善虚拟内存分配
- 动态内存资源的分配
- 用户态和内核态的实现
- 完善中断机制
- 实现用户程序和内核栈的切换
- 编写更多的C语言库函数
- 建立起系统调用system call
- ...

最后，本次实验里我还是没有通过Makefile完成整个过程的自动化编译、链接和运行。不过，我通过X server已经实现了WSL的GUI显示，我将在下一次实验里，使用工具链：gcc + ld + make + qemu + bochs，完成在WSL环境下的编程。