

# **ShenleOS 微内核实现**

# 目录

一. 实验目的及实验环境.....	3
1. 实验目的.....	3
2. 实验环境.....	3
二. 实验内容.....	4
1. 实验名称.....	4
2. 实验模块划分.....	4
三. 方案设计.....	4
1. 配置开发环境.....	4
2. 熟悉计算机启动过程并实现显示“hello! ”.....	8
3. 添加全局段描述符表.....	17
4. 添加中断描述符表.....	21
5. 完成中断请求和定时器中断.....	26
6. 物理内存管理的实现.....	28
7. 虚拟内存管理的实现.....	30
8. 内核堆管理的实现.....	33
四. 测试数据及运行结果.....	35
1. 测试屏幕操作函数打印”hello ShenleOS!”.....	35
2. 测试中断处理函数.....	36
3. 测试时钟中断.....	36
4. 测试内存管理.....	37
五. 总结.....	37

1. 实验过程中遇到的问题及解决办法； .....	37
2. 对设计及调试过程的心得体会。 .....	37
六. 附录.....	38
1. 源代码.....	38
2. 参考资料.....	38

## 一. 实验目的及实验环境

### 1. 实验目的

通过本次实验学会如何在基于 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。学习 x86CPU 的保护模式下操作系统内核的编写方法，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。

### 2. 实验环境

(1) 工作环境：Linux（我选用的 linux 版本是 ubuntu12.04）

（选择 linux 理由：可以自由使用的一系列的开源软件能很好的协助我们开发和调试工作。）

(2) 开发语言：主要为 C 语言，部分由 32 位汇编语言实现。

(3) 开发工具

C 语言编译器使用 gcc,链接器使用 ld。

汇编编译器 nasm。（方便习惯于 Intel 风格的汇编语法。由于 gcc 编译器的原因，因此某些 C 语言代码中有内联汇编指令使用的是 AT&T 风格的汇编语法。）

虚拟机选用 bochs：需要下载原码编译后才可开启 gdb 的联合调试功能。

(4) 代码管理

使用 git 作为版本管理工具，并托管在 github 上。

（github 地址：<https://github.com/vmezhang>）

## 二. 实验内容

### 1. 实验名称

ShenleOS：一个运行在 x86-IA32 架构下的小内核，要求能驱动 x86 简单的硬件。具体实现参照《jamesm-tutorials Documentation Release 2.0.0》的文档。

### 2. 实验模块划分

实验入门部分：配置开发环境，熟悉计算机启动过程等。

中断部分：添加全局段描述符表，添加中断描述符表，完成中断请求和定时器中断。

内存管理部分：物理内存管理的实现，虚拟内存管理的实现，内核堆管理的实现。

## 三. 方案设计

微内核实现说明：

关于部分数据类型以及库函数的实现：由于现有的用户态的 C 语言标准库无法使用在内核中,但是内核开发中难免要用到一些数据类型还有字符串操作的函数等等,所以我们需要自己实现这些字相关的数据定义以及函数。因此具体的定义及实现可以参见 Linux 内核原码。（之后报告中将直接使用这些函数以及数据类型，不再赘述，具体实现见电子版原码。）

### 1. 配置开发环境

(1) bochs 虚拟机的配置文件：bochsrc.txt

```
# -----
```

```
# Bochs 配置文件
```

```

#
# -----

# 开始 gdb 联合调试,这很重要
gdbstub: enabled=1,port=1234,text_base=0,data_base=0,bss_base=0
# 内存
megs: 32
# ROM 文件
romimage: file="$BXSHARE/BIOS-bochs-latest"
vgaromimage: file="$BXSHARE/VGABIOS-lgpl-latest"

# 软盘
floppya: 1_44=floppy.img, status=inserted
boot: a
# 启动设备为软盘
boot: floppy
# 鼠标不启用
mouse: enabled=0
# 键盘启用 US 键盘映射
keyboard_mapping: enabled=1, map="$BXSHARE/keymaps/x11-pc-us.map"

# CPU 配置
clock: sync=realtime
cpu: ips=1000000

```

## (2) Makefile 文件

```

#!/Makefile
#
# -----
#
#   ShenleOS 这个小内核的 Makefile
#
#   默认使用的 C 语言编译器是 GCC、汇编语言编译器是 nasm

```

```

#
# -----
#

# patsubst 处理所有在 C_SOURCES 字列中的字（一系列文件名），如果它的 结
尾是 '.c'，就用 '.o' 把 '.c' 取代
C_SOURCES = $(shell find . -name "*.c")
C_OBJECTS = $(patsubst %.c, %.o, $(C_SOURCES))
S_SOURCES = $(shell find . -name "*.s")
S_OBJECTS = $(patsubst %.s, %.o, $(S_SOURCES))

CC = gcc
LD = ld
ASM = nasm

C_FLAGS = -c -Wall -m32 -ggdb -gstabs+ -nostdinc -fno-builtin -fno-stack-protector
-I include
LD_FLAGS = -T link.ld -m elf_i386 -nostdlib
ASM_FLAGS = -f elf -g

all: $(S_OBJECTS) $(C_OBJECTS) link update_image

# The automatic variable `$$' is just the first prerequisite
.c.o:
    @echo 编译代码文件 $$< ...
    $(CC) $(C_FLAGS) $$< -o $$@

.s.o:
    @echo 编译汇编文件 $$< ...
    $(ASM) $(ASM_FLAGS) $$<

link:
    @echo 链接内核文件...
    $(LD) $(LD_FLAGS) $(S_OBJECTS) $(C_OBJECTS) -o kernel

.PHONY:clean
clean:
    $(RM) $(S_OBJECTS) $(C_OBJECTS) kernel

```

```

.PHONY:update_image
update_image:
    sudo mount floppy.img /mnt/kernel
    sudo cp kernel /mnt/kernel/kernel
    sleep 1
    sudo umount /mnt/kernel

.PHONY:mount_image
mount_image:
    sudo mount floppy.img /mnt/kernel

.PHONY:umount_image
umount_image:
    sudo umount /mnt/kernel

.PHONY:qemu
qemu:
    qemu -fda floppy.img -boot a

.PHONY:bochs
bochs:
    bochs -f bochsrc.txt

.PHONY:debug
debug:
    gnome-terminal -e "bochs -f bochsrc.txt"
    sleep 1
    cgdb -x gdbinit
.PHONY:run
run:
    bochs -f bochsrc2.txt

```

### (3) 链接器脚本 link.ld

```

ENTRY(start)
SECTIONS
{
    .text 0x100000 :
    {
        code = .; _code = .; __code = .;
        *(.text)
        . = ALIGN(4096);
    }
}

```



```

.data :
{
    data = .; _data = .; __data = .;
    *(.data)
    *(.rodata)
    . = ALIGN(4096);
}

.bss :
{
    bss = .; _bss = .; __bss = .;
    *(.bss)
    . = ALIGN(4096);
}

.stab :
{
    stab = .; _stab = .; __stab = .;
    *(.stab)
    . = ALIGN(4096);
}

.stabstr :
{
    stabstr = .; _stabstr = .; __stabstr = .;
    *(.stabstr)
    . = ALIGN(4096);
}

end = .; _end = .; __end = .;

/DISCARD/ : { *(.comment) *(.eh_frame) }
}

```

这个脚本告诉 ld 程序如何构造我们所需的内核映像文件。

## 2. 熟悉计算机启动过程并实现显示“hello! ”

### (1) 计算机启动过程理论知识:

首先,实验中的内核使用 32 位的地址总线来寻址,所以能编址出 2 的 32 次方,

也就是 4G 的地址空间。在主板上除了内存还有 BIOS、显卡、声卡、网卡 15 等外部设备,CPU 需要和这些外设进行通信。在 x86 下,当需要访问这些存储单元的时候,就需要给予不同的访问地址来区分每一个读写单元。端口统一编址就是把所有和外设存储单元对应的端口直接编址在这 4G 的地址空间里,当我们对某一个地址进行访问的时候实际上是在访问某个外设的存储单元。而端口独立编址就是说这些端口没有编址在地址空间里,而是另行独立编址。x86 架构一部分采用了端口独立编址,还有部分采用了端口统一编址。部分外设的存储单元直接可以通过某个内存地址访问,而其他部分在一个独立的端口地址空间中,需要使用 in/out 指令去访问。

CPU 在加电后的启动过程:从按下电源开始,首先是 CPU 重置。主板加电之后在电压尚未稳定之前,主板上的北桥控制芯片会向 CPU 发出重置信号(Reset),此时 CPU 进行初始化。当电压稳定后,控制芯片会撤销 Reset 信号,CPU 便开始了模式化的工作。此时形成的第一条指令的地址是 0xFFFFFFFF018 ,从这里开始,CPU 就进入了一个"取指令-翻译指令-执行"的循环了。所以我们需要做的就是各个阶段提供给 CPU 相关的数据,以完成这个"接力赛"。这个接力过程中任何一个环节如果出现致命问题,其导致的直接后果就是宕机。死机是最好的结果,最坏的结果是程序在"默默的"破坏我们的数据,所以一定要谨慎对待。在 4G 的地址空间里,有一些地址是分给外设的,这个地址便是映射到 BIOS 的。计算机刚加电的时候内存等芯片尚未初始化,所以也只能是指向 BIOS 芯片里已经被"固化"的指令了。

紧接着就是 BIOS 的 POST(Power On Self Test,上电自检)过程了,BIOS 对计算机各个部件开始初始化,如果有错误会给出报警音。当 BIOS 完成这些工作之后,

它的任务就是在外部存储设备中寻找操作系统。BIOS 里面就有一张启动设备表 19, BIOS 会按照这个表里面列出的顺序查找可启动设备。检测设备是否可以启动的规则: 如果这个存储设备的第一个扇区中 512 个字节 20 的最后两个字节是 0x55 和 0xAA, 那么该存储设备就是可启动的。所以 BIOS 会对这个列表中的设备逐一检测, 只要有一个设备满足要求, 后续的设备将不再测试。当 BIOS 找到可启动的设备后, 便将该设备的第一个扇区加载到内存的 0x7C00 地址处, 并且跳转过去执行。而我们要做的事情, 便是从构造这个可启动的扇区开始。因为一个扇区只有 512 字节, 放不下太多的代码, 所以常规的做法便是在这里写下载入操作系统内核的代码——bootloader 程序。一般意义上的 bootloader 负责将软硬件的环境设置到一个合适的状态, 然后加载操作系统内核并且移交执行权限。

GRUB 是一个来自 GNU 项目的多操作系统启动程序。它是多启动规范的实现, 允许用户可以在计算机内同时拥有多个操作系统, 并在计算机启动时选择希望运行的操作系统。这次实验决定直接使用 GRUB 来加载内核。以后就能让它很简单的安装在物理机器上并与其他操作系统共存。

## (2) 内核的启动

启动镜像的制作: 将小内核放在 1.44MB 的软盘里, 并运行在虚拟机中。

内核的入口和初始化:

从 ld 链接脚本中可以看到:

ENTRY(start)

ld 链接器入口函数是 start, 所以代码从 start 函数开始。

入口部分的代码 boot/boot.s:

```
;
; boot.s -- Kernel start location. Also define multiboot header.
;
```

`MBOOT_HEADER_MAGIC` `equ` `0x1BADB002` ; Multiboot 魔数,  
由规范决定的

`MBOOT_PAGE_ALIGN` `equ` `1 << 0` ; 0 号位表示所有的引导模  
块将按页(4KB)边界对齐

`MBOOT_MEM_INFO` `equ` `1 << 1` ; 1 号位通过 Multiboot 信  
息结构的 `mem_*` 域包括可用内存的信息(一般的 ELF 映像这样就可以)

`MBOOT_AOUT_KLUDGE` `equ` `1 << 16`

; 注意, 我们并没有加上第 16 位(`MBOOT_AOUT_KLUDGE`), 这意味着  
GRUB 不会使用我们的符号表

`MBOOT_HEADER_FLAGS` `equ` `MBOOT_PAGE_ALIGN` |  
`MBOOT_MEM_INFO`

; 域 `checksum` 是一个 32 位的无符号值, 当与其他的 `magic` 域(也就是 `magic`  
和 `flags`)相加时,

; 结果必须是 32 位的无符号值 0(即 `magic + flags + checksum = 0`)

`MBOOT_CHECKSUM` `equ` `-(MBOOT_HEADER_MAGIC +`  
`MBOOT_HEADER_FLAGS)`

; 符合 Multiboot 规范的 OS 映像需要这样一个 `magic Multiboot` 头

[BITS 32] ; 所有代码以 32-bit 的方式编译

`section .text` ; 代码段从这里开始

; 在代码段的起始位置设置符合 `multiboot` 规范的标记

dd MBOOT\_HEADER\_MAGIC ; GRUB 会通过这个魔数判断该映像是否支持

dd MBOOT\_HEADER\_FLAGS ; GRUB 的一些加载时选项，其详细注释在定义处

dd MBOOT\_CHECKSUM ; 检测数值，其含义在定义处

[GLOBAL start] ; 内核代码入口，此处提供该声明给 ld 链接器

[EXTERN sl\_main] ; 声明内核 C 代码的入口函数

start:

cli ; 此时还没有设置好保护模式的中断处理，所以必须关闭中断

mov esp, stack ; 设置内核栈地址，按照 multiboot 规范，当需要使用堆栈时，OS 映像必须自己创建一个

push ebx ; 调用内核 main 函数的参数，struct multiboot \*mboot\_ptr

mov ebp, 0 ; 帧指针修改为 0

call sl\_main ; 调用内核入口函数

stop:

hlt ; 停机指令，什么也不做，可以降低 CPU 功耗

jmp stop ; 到这里结束，关机什么的后面再说

.end:

[GLOBAL stack]

section .bss ; 未初始化的数据段从这里开始

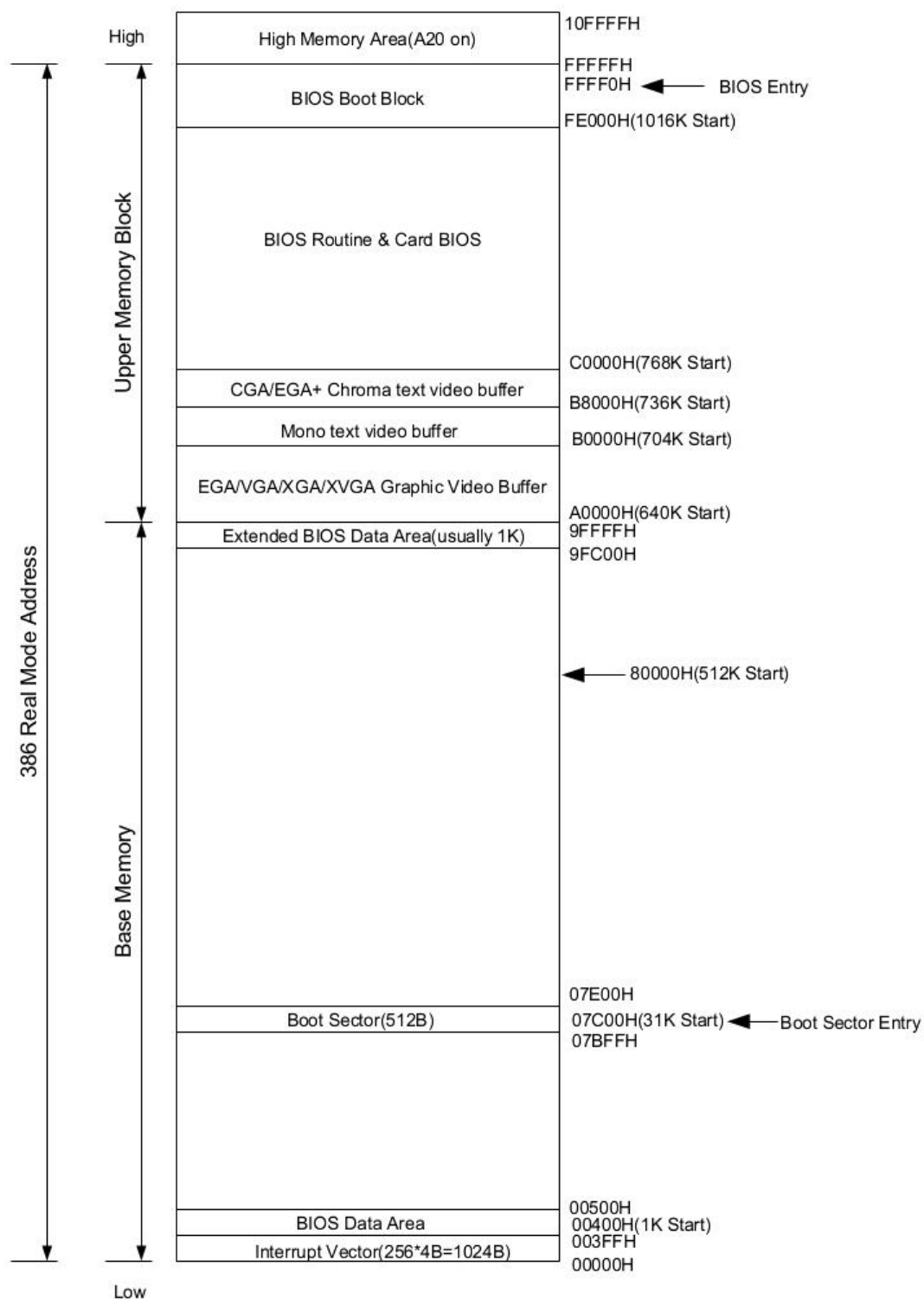
```
resb 32768    ; 这里 32KB 作为内核栈  
stack:
```

用 C 语言来实现内核的入口函数：

```
int sl_main()  
{  
    return 0;  
}
```

(3) 字符模式下的显卡驱动：

1MB 以下的地址空间分布：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。

#### (4) 测试屏幕操作函数打印”hello ShenleOS!”

直接向 sl\_main 函数中添加以下内容:

```
#include "#include "monitor.h"

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.
    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");

    return 0;
}
```

编译运行后就会打印红色的”hello ShenleOS!”。

### 3. 添加全局段描述符表

#### (1) 保护模式下的内存分段

对 CPU 来讲,系统中的所有储存器中的储存单元都处于一个统一的逻辑储存器中,它的容量受 CPU 寻址能力的限制。这个逻辑储存器就是我们所说的线性地址空间。8086 有 20 位地址线,拥有 1MB 的线性地址空间。而 80386 有 32 位地址线,拥有 4GB 的线性地址空间。但是 80386 依旧保留了 8086 采用的地址分段的方式,只是增加了一个折中的方案,即只分一个段,段基址  $0 \times 00000000$ ,段长  $0 \times \text{FFFFFFFF}$ (4GB),这样的话整个线性空间可以看作就一个段,这就是所谓的平坦模型(Flat Mode)。首先是对内存分段中每一个段的描述,内模式对于内存段并没有访问控制,任意的程序可以修改任意地址的变量,而保护模式需要对内存段的性质和允许的操作给出定义,以实现特定内存段的访问检测和数据保护。考虑到各种属性和需要设置的操作,32 位保护模式下对一个内存段的描述需要 8 个字节,其称之为段描述符(Segment Descriptor)。段描述符分为数据段描述符、指令段描述符和系统段描述符三种。

段描述符的 8 个字节的分解图:

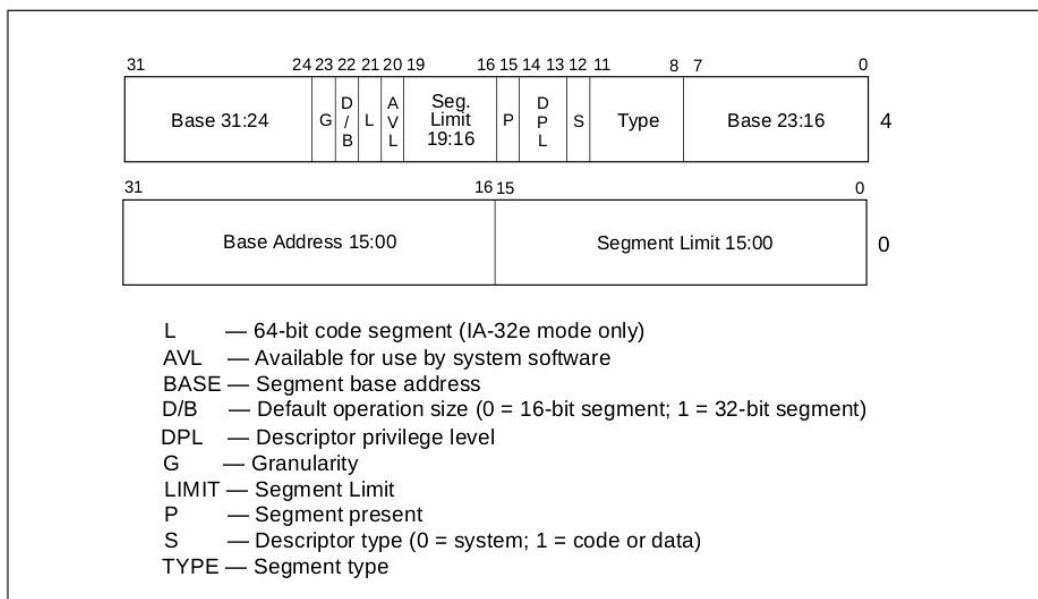
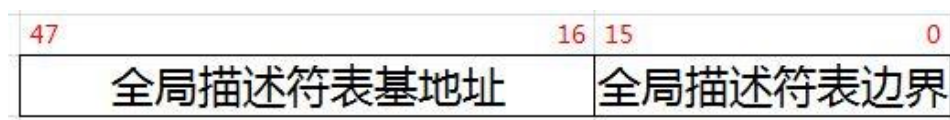


Figure 3-8. Segment Descriptor

显然,寄存器不足以存放 N 多个内存段的描述符集合,所以这些描述符的集合(称之为描述符表)被放置在内存里了。在很多描述符表中,最重要的就是所谓的全局描述符表(Global Descriptor Table,GDT),它为整个软硬件系统服务。

这些描述符表放置在内存中没有固定的位置,可以任由程序员安排在任意合适的位置。Intel 设置了一个 48 位的专用的全局描述符表寄存器(GDTR)来保存全局描述符表的信息。如图所示,0-15 位表示 GDT 的边界位置(数值为表的长度-1,因为从 0 计算),16-47 位,这 32 位存放的就是 GDT 的基地址(恰似数组的首地址)。

GDTR 的结构图:



既然用 16 位来表示表的长度,那么 2 的 16 次方就是 65536 字节,除以每一个描述符的 8 字节,那么最多能创建 8192 个描述符。

80386CPU 加电的时候自动进入实模式 55 既然 CPU 加电后就一直工作在实模式下了。80386CPU 内部有 5 个 32 位的控制寄存器(Control Register,CR),分别

是 CR0 到 CR3,以及 CR8。用来表示 CPU 的一些状态,其中的 CR0 寄存器的 PE 位(Protection Enable,保护模式允许位),0 号位,就表示了 CPU 的运行状态,0 为实模式,1 为保护模式。通过修改这个位就可以立即改变 CPU 的工作模式。

CR0 寄存器的结构图:

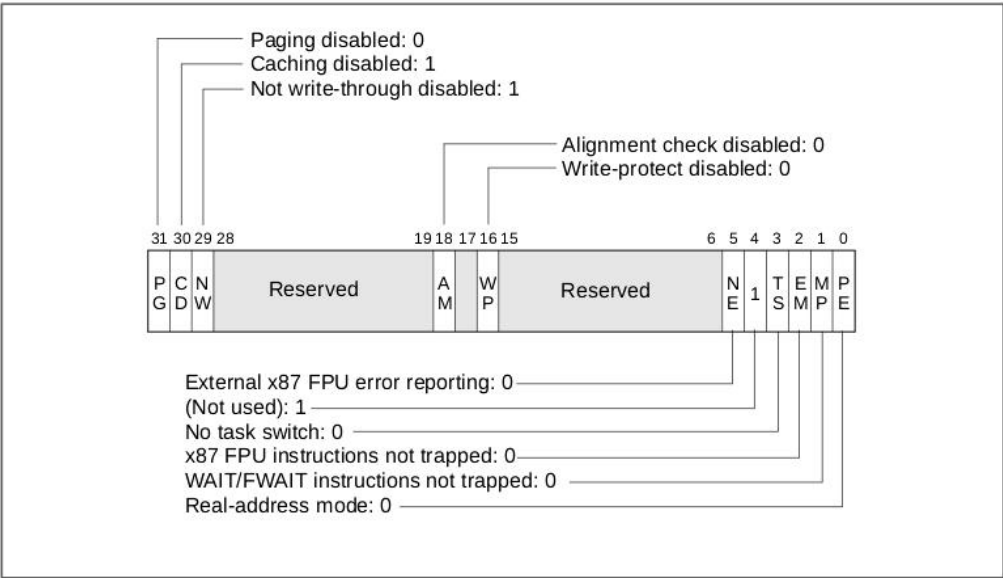


Figure 9-1. Contents of CR0 Register after Reset

需要注意的是，一旦 CR0 寄存器的 PE 位被修改,CPU 就立即按照保护模式去寻址了,所以这就要求我们必须在进入保护模式之前就在内存里放置好 GDT,然后设置好 GDTR 寄存器。实模式下只有 1MB 的寻址空间,所以 GDT 就等于被限制在了这里。不过进入保护模式之后我们就可以在 4G 的空间里设置并修改原来的 GDTR 了。

地址合成的过程如下图所示：

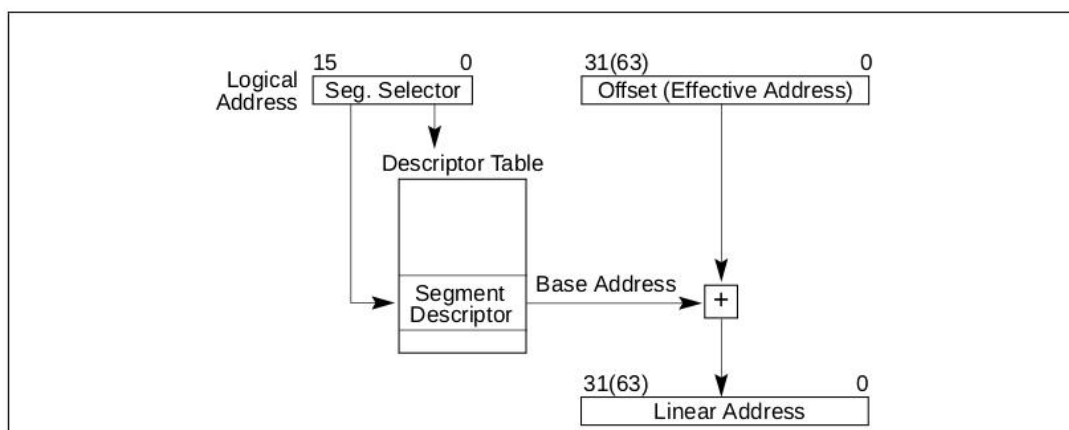


Figure 3-5. Logical Address to Linear Address Translation

## (2) 具体采用的分段策略

下面开始就是我们的内核具体的设计方案。分段是 Intel 的 CPU 一直保持着的一种机制,而分页只是保护模式下的一种内存管理策略。不过想开启分页机制,CPU 就必须工作在保护模式,而工作在保护模式时候可以不开启分页。所以事实上分段是必须的,而分页是可选的。

我们将通过平坦模式“绕过”分段机制：当整个虚拟地址空间是一个起始地址为 0,限长为 4G 的“段”时,我们给出的偏移地址就在数值上等于是段机制处理之后的地址了。我们不只是简单的对所有的段使用同样的描述符,而是给代码段和数据段分配不同的描述符。

下面的示意图描述了这个抽象:

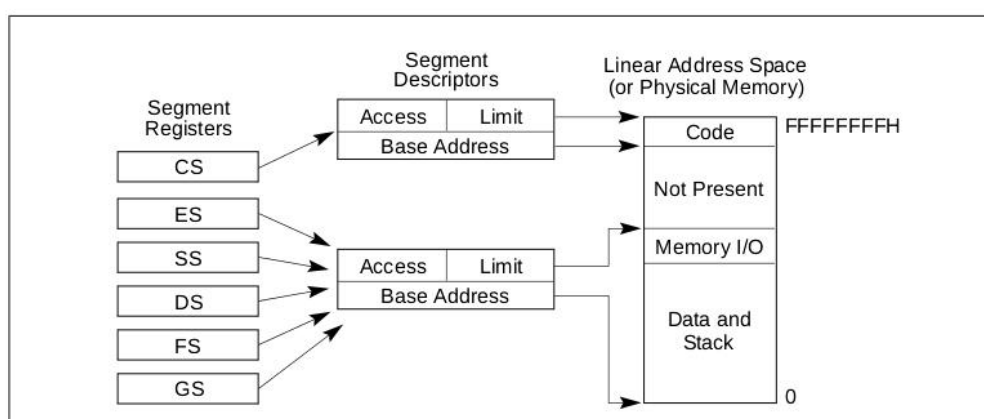


Figure 3-3. Protected Flat Model

接下来开始添加全局描述符表。(全局描述符表的具体实现代码详见代码清

单 include/gdt.h, gdt/gdt.c, gdt/gdt.s.s。)

完成添加后,再次修改入口函数 sl\_main 如下:

```
#include "gdt.h"
#include "monitor.h"
#include "debug.h"

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    init_debug();
    // 初始化全局段描述符表
    init_gdt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
        study.\nYou can copy it freely!\n\n");

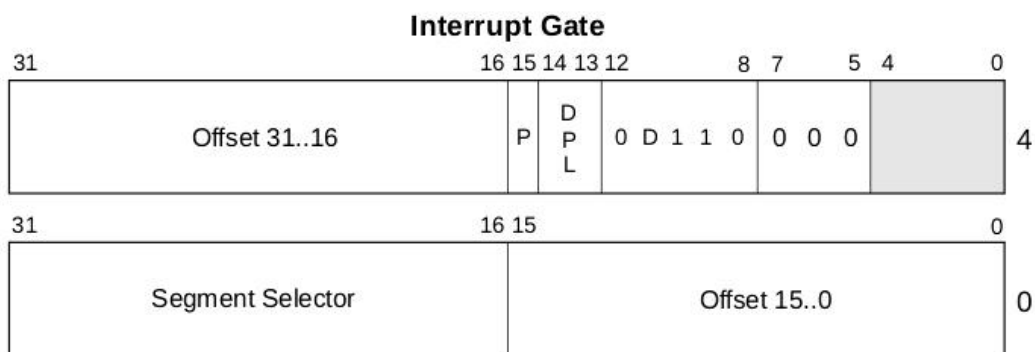
    return 0;
}
```

#### 4. 添加中断描述符表

##### (1) 中断的实现

我们的重点是保护模式下的中断处理。中断处理程序是运行在 ring0 层的,这就意味着中断处理程序拥有着系统的全部权限,仿照内存段描述符表的思路,Intel 设置了一个叫做中断描述符表(IDT, Interrupt Descriptor Table)的东西,和段描述符表一样放置在主存中,类似地,也有一个中断描述符表寄存器(IDTR)记录这个表的起始地址。

中断描述符表的结构:



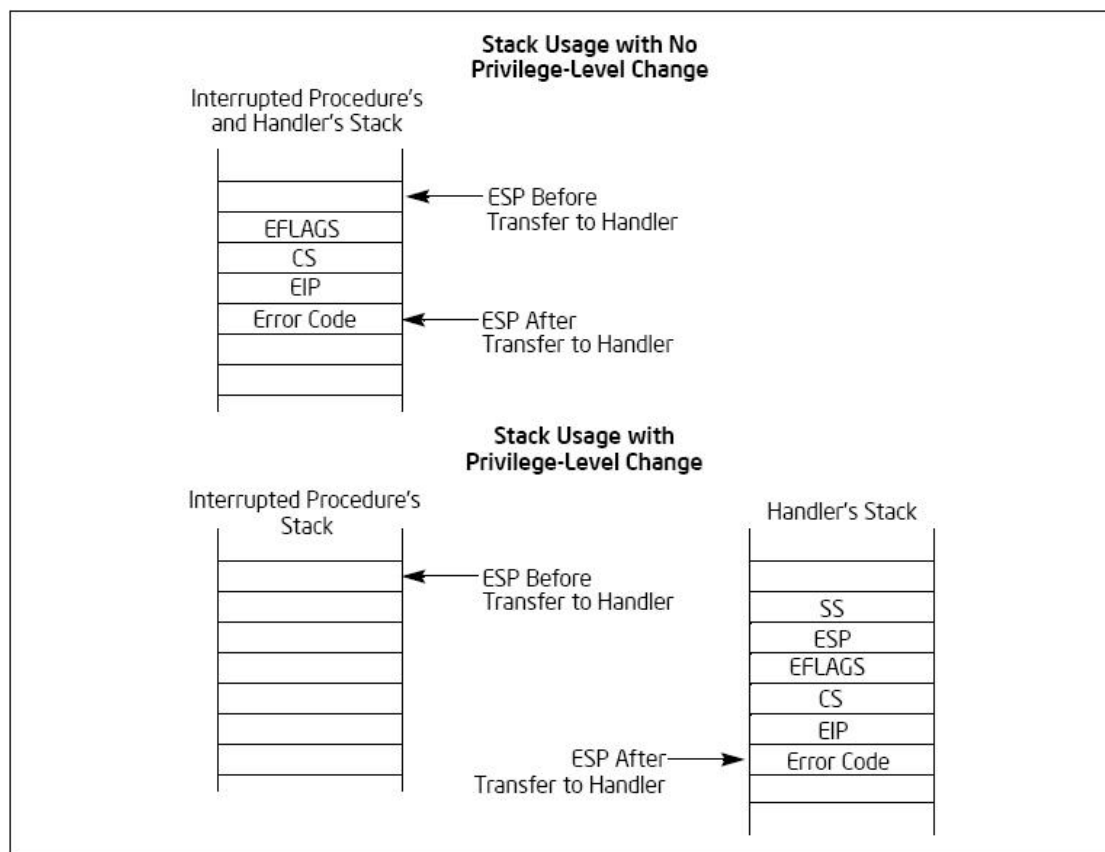
根据这个描述信息相关的 C 语言结构体定义详见 include/idt.h。

CPU 处理中断的过程,首先是起始过程,也就是从 CPU 发现中断事件后,打断当前程序或任务的执行,根据某种机制跳转到中断处理函数去执行的过程:

- a. CPU 在执行完当前程序的每一条指令后,都会去确认在执行刚才的指令过程中是否发送中断请求过来,如果有那么 CPU 就会在相应的时钟脉冲到来时从总线上读取中断请求对应的中断向量。然后根据得到的中断向量为索引到 IDT 中找到该向量对应的中断描述符,中断描述符里保存着中断处理函数的段选择子;
- b. CPU 使用 IDT 查到的中断处理函数段选择子从 GDT 中取得相应的段描述符,段描述符里保存了中断处理函数的段基址和属性信息。此时 CPU 要进行一个很关键的特权检验的过程,这个涉及到 CPL、RPL 和 DPL 的数值检验以及判断是否发生用户态到内核态的切换。如果发生了切换,还要涉及到 TSS 段和用户栈和内核栈的切换;
- c. 确认无误后 CPU 开始保存当前被打断的程序的现场(即一些寄存器的值),以便于将来恢复被打断的程序继续执行。这需要利用内核栈来保存相关现场信息,即依次压入当前被打断程序使用的 eflags、cs、eip、以及错误代码号(如果当前中断有错误代码);
- d. 最后 CPU 会从中断描述符中取出中断处理函数的起始地址并跳转过去执行。

以上是起始过程,中断处理函数执行完成之后需要通过 `iret` 或 `iretd` 指令恢复被打断的程序的执行。这时候比较简单,首先 CPU 会从内核栈里弹出先前保存的被打断的程序的现场信息,即之前的 `eflags`,`cs`,`eip` 重新开始被打断前的任务。61 需要注意的是:如果此次处理的是带有错误码的中断,CPU 在恢复先前程序的现场时,并不会弹出错误代码。这一步需要通过软件完成,即要求相关的中断处理函数在使用 `iret` 指令返回之前添加出栈代码主动弹出错误代码。

下图描述了 CPU 自动保护和恢复的寄存器的栈结构:



CPU 在发生中断的时候是按照上问所描述的过程执行的.操作系统首先是实现中断处理函数。按照 Intel 的规定,0~19 号中断属于 CPU 所有 62 ,而且第 20-31 号中断也被 Intel 保留,所以从 32~255 号才属于用户自定义中断。虽说是"用户自定义",其实在 x86 上有些中断按照习惯还是给予了固定的设备。比如 32 号是 timer 中断,33 号是键盘中断等等。



中断处理函数的实现：CPU 在中断产生时自动保存了部分的执行现场,但是依旧有很多寄存器需要我们自己去保护和恢复。CPU 保护的寄存器和剩余需要保护的寄存器一起定义的结构体（详见 include/idt.h）。

所有的中断处理函数中,除了 CPU 本身保护的现场外,其它寄存器的保存和恢复过程都是一样的。所以,如果在每个中断处理函数中都实现一次显然冗余而且易错。所以我们很自然把原本的中断处理函数逻辑上拆解为三部分,第一部分是统一的现场保护操作;第二部分是每个中断特有的处理逻辑;第三部分又是一致的现场恢复。实际上我们把每个中断处理函数拆解为四段,在四个函数里实现。具体的实现详见 idt/idt\_s.s。

函数执行的开始位置我们自行向栈里压入中断的号码,这是为了识别出中断号码。我们可以用宏汇编来生成代码。有的中断处理函数会自动压入错误号,而有的不会,这样给我们的清栈造成了麻烦。所以我们在不会压入错误号的中断的处理函数里手动压入 0 作为占位,这样方便我们在清理的时候不用分类处理。

所有中断处理函数共有的保护现场操作参见 idt/idt\_s.s。

idt/idt\_s.s 中声明了一个外部函数 idt\_handler()它的实现添加在 idt/idt.c 中：

```
// 调用中断处理函数
void isr_handler(pt_regs *regs)
{
    if (interrupt_handlers[regs->int_no]) {

        interrupt_handlers[regs->int_no](regs);
    } else {
        printk_color(rc_black , rc_blue , "Unhandled interrupt: %d\n", regs
            ->int_no);
    }
}
```

从上面的代码中我们看到,事实上具体的中断处理函数的原型都是 `void (pt_regs*)`,它们被统一组织放置在了全局的函数指针数组 `interrupt_handlers` 里面。`idt_handler` 函数如判断这个中断函数是否注册,如果注册了会执行该函数,否则打印出 `Unhandled in-terrput` 和中断号码。

最后是中断描述符表的创建和和加载函数参见 `idt/idt.c`。

## (2) 测试中断处理函数

添加完中断描述符表后,入口代码 `sl_main()`修改如下:

```
#include "gdt.h"
#include "idt.h"
#include "monitor.h"
#include "debug.h"

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    init_debug();
    // 初始化全局段描述符表

    init_gdt();
    init_idt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
        study.\nYou can copy it freely!\n\n");

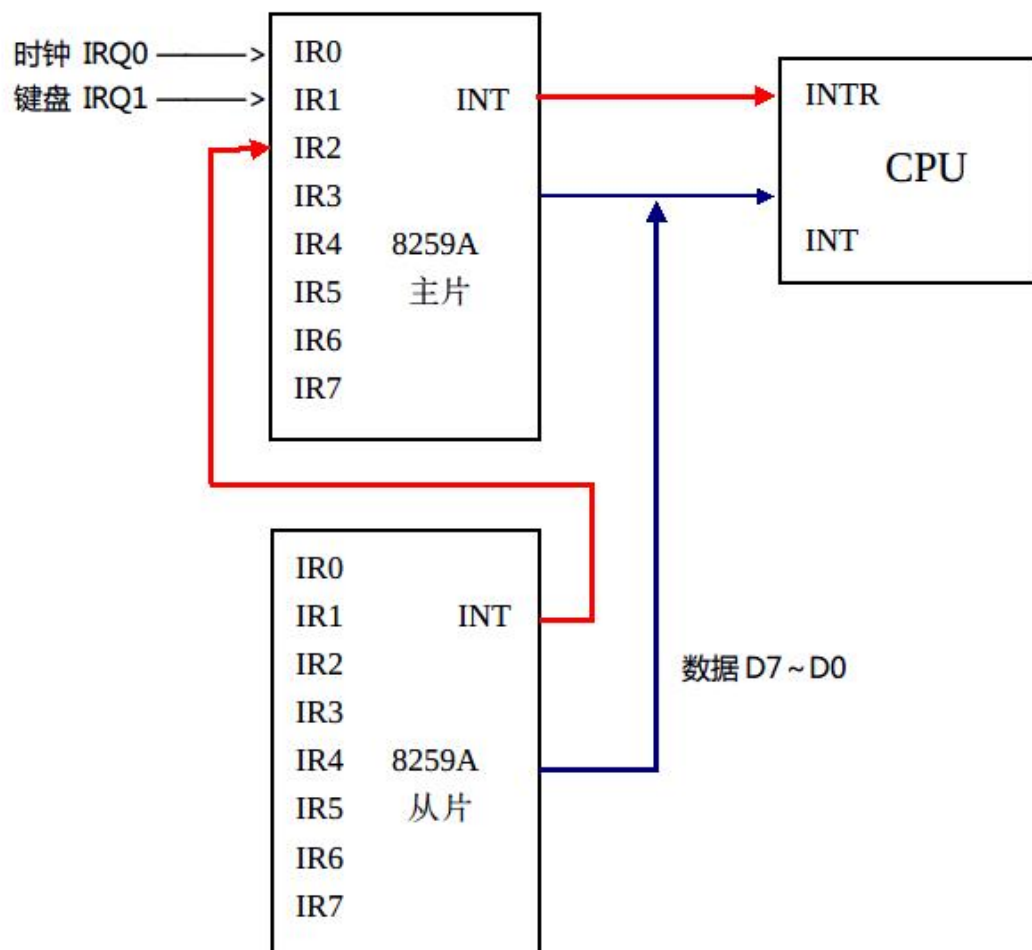
    asm volatile ("int $0x3");
    asm volatile ("int $0x4");

    return 0;
}
```

## 5. 完成中断请求和定时器中断

(1) 中断请求：外设的所有中断由中断控制芯片 8259A 统一汇集之后连接到 CPU 的 INTR 引脚。64 这章我们就来探究 8259APIC 的初始化和实现定时器的中断处理。8259A PIC 每一片可以管理 8 个中断源,显然一般情况下设备数量会超过这个值。这里就要提到 IBM PC/AT 8259A PIC 架构了,IBM 的设计方案是使用 8259APIC 的级联功能,使用两片级联(分为主、从片)的方式来管理硬件中断。其中主片的 INT 端连接到 CPU 的 INTR 引脚,从片的 INT 连接到主片的 IR2 引脚。

结构如下图所示:



图中时钟中断连接在主片的 IRQ0 引脚,键盘中断连接在了主片的 IRQ1 引脚。其它的引脚暂时用不到就不说了。在上一张描述中断描述符表时我们知道了 0~31 号中断是 CPU 使用 and 保留的,用户可以使用的中断从 32 号开始。所以这里的 IRQ0 对应的中断号就是 32 号,IRQ1 就是 33 号,然后以此类推。

## (2) 实现代码

首先对 8259A PIC 初始化,在设置中断描述符表的函数 `init_idt` 最前面加入代码 (参见原码 `idt/idt.c` 中)。

完成初始化之后,继续添加对 IRQ 处理的函数。在 `idt.h` 头文件末尾添加的内容参见 `include/idt.h`。

然后是 `idt_s.s` 中添加相应的处理过程参见 `idt/idt_s.s`

最后添加 `init_idt` 函数,构造 IRQ 的相关描述符和具体的 IRQ 处理函数,参见 `idt/idt.c`。

## (3) 实现时钟中断的产生和处理

时钟中断对于操作系统内核来说很重要的一种中断,它使得 CPU 无论在执行任何用户或者内核的程序时,都能定义的将执行权利交还到 CPU 手中来。65 除了记录时间之外,时钟中断的处理函数里通常都是对进程的调度处理。

具体的时钟中断源是 8253/8254 Timer 产成的,要按照需要的频率产生中断,需要先配置 8253/8254 Timer 芯片。代码参见 `include/timer.h` 和 `timer/timer.c`。

最后,修改入口函数 `sl_main()` 进行测试:

```
#include "gdt.h"
#include "idt.h"
#include "monitor.h"
#include "debug.h"
#include "timer.h"
```

```

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    init_debug();
    // 初始化全局段描述符表

    init_gdt();
    init_idt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
        study.\nYou can copy it freely!\n\n");

    init_timer(200);
    // 开启中断

    asm volatile ("sti");

    return 0;
}

```

## 6. 物理内存管理的实现

在 32 位操作系统下使用 32 位地址总线(暂时原谅我在这里错误的描述吧,其实还有 PAE 这个东西),所以寻址空间有 2 的 32 次方,也就是 4GB。一定要注意,我们强调了很多次了,这个空间里,有一些断断续续的地址实际上是指向了其它的外设,不过大部分还是指向 RAM 的。具体采取的分页大小可以有多种选择,但是过于小的分页会造成管理结构太大,过于大的分页又浪费内存。现在较为常见的分页是 4KB 一个页,也就是 4096 字节一个页。简单计算下,4GB 的内存分成 4KB 一个的页,那就是 1MB 个页,没错吧?每个虚拟页到物理页的映射需要 4 个字节来存储的话(别忘了前提是 32 位环境下),整个 4GB 空间的映射需要 4MB 的数据结构来存储。目前看起来一切都很好,4MB 似乎也不是很大。但是,这只是一个虚拟地址空间的映射啊,别忘了每个进程都有自己的映射,而且操作系统中通常有 N 个进程在运行。这样的话,假如有 100 个进程在运行,就需要 400MB 的内存来存储管理信息!这就太浪费了。怎么办?聪明的工程师们提出了分级页表的实现策略,他们提出了页目录,页表的概念。以 32 位的地址来说,分为 3 段来寻址,分别是地址的低 12 位,中间 10 位和高 10 位。高 10 位表示当前地址项在页目录中的偏移,最终偏移处指向对应的页表,中间 10 位是当前地址在该页表中的偏移,我们按照这个偏移就能查出来最终指向的物理页了,最低的 12 位表示当前地址在该物理页中的

偏移。就这样,我们就实现了分级页表。这样做的话映射 4GB 地址空间需要 4MB+4KB 的内存。在一个进程中,实际使用到的内存大都远没有 4GB 这么大,所以通过两级页表的映射,我们就可以只映射需要的地址就可以了。

多级页表图:

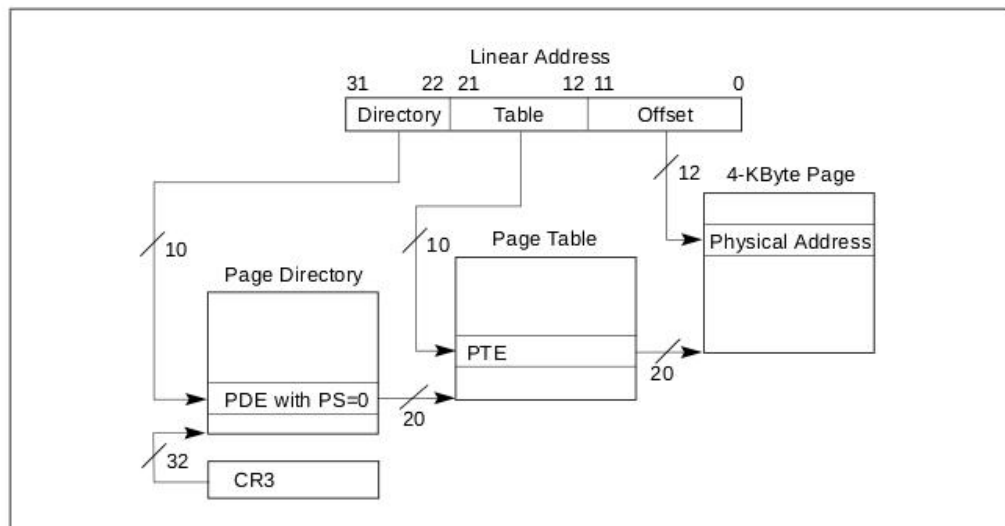


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

物理内存管理算法最著名的就是 Linux 内核所采用的伙伴算法了,这方面的资料也很容易获取到。伙伴算法在申请和释放物理页框的时候会对物理页框进行合并操作,尽可能的保证可用物理内存的连续性。这里需要引入内存碎片这个概念,内存碎片分为内部碎片和外部碎片两种。内部碎片就是已经被分配出去却不能被利用的内存空间,比如我们为了管理方便,按照 4KB 内存块进行管理。此时任何申请内存的操作都会以 4KB 的倍数返回内存块。即使申请 1 个字节也返回指向 4KB 大的内存块的指针,这样的话造成了分配出去的内存没有被有效利用,而且剩余空间无法分配给其它进程(因为最小的管理单位是 4KB)。外部碎片是指内存频繁请求和释放大小不同的连续页框后,导致在已分配页框块周围分散了许多小块空闲的页框,尽管这些空闲页框的总数可以满足接下来的请求,但却无法满足一个大块的连续页框。我们会采用一个很简单的策略来对内存进行管理操作:将物理页面的管理地址设定在 1MB 以上内核加载的结束位置之后,从这个起始位置到 512MB 的地址处将所有的物理内存按页划分,将每页的地址放入栈里存储。这样在需要的时候就可以按页获取到物理内存了。

## (2) 物理内存管理的具体实现

具体实现参见代码清单#include/pmm.h 和 memm/pmm.c。

最后并修改入口函数 sl\_main()进行测试:

```
#include "gdt.h"
#include "idt.h"
#include "monitor.h"
#include "debug.h"
#include "timer.h"
#include "pmm.h"
```

```

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    init_debug();

    // 初始化全局段描述符表

    init_gdt();
    init_idt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
        study.\nYou can copy it freely!\n\n");

    // 初始化物理内存管理

    init_pmm(mboot_ptr);

    //init_timer(200);

    // 开启中断

    //asm volatile ("sti");

    return 0;
}

```

## 7. 虚拟内存管理的实现

### (1)虚拟内存管理的理论

程序代码和数据必须驻留在内存中才能得以运行，然而系统内存数量很有限，往往不能容纳一个完整程序的所有代码和数据，更何况在多任务系统中，可能需要同时打开子处理程序，画图程序，浏览器等很多任务，想让内存驻留所有这些程序显然不太可能。因此我们首先能想到的就是将程序分割成小份，只让当前系统运行它所有需要的那部分留在内存，其它部分都留在硬盘。当系统处理完当前任务片段后，再从外存中调入下一个待运行的任务片段。

下图描述了地址转换的完整过程:



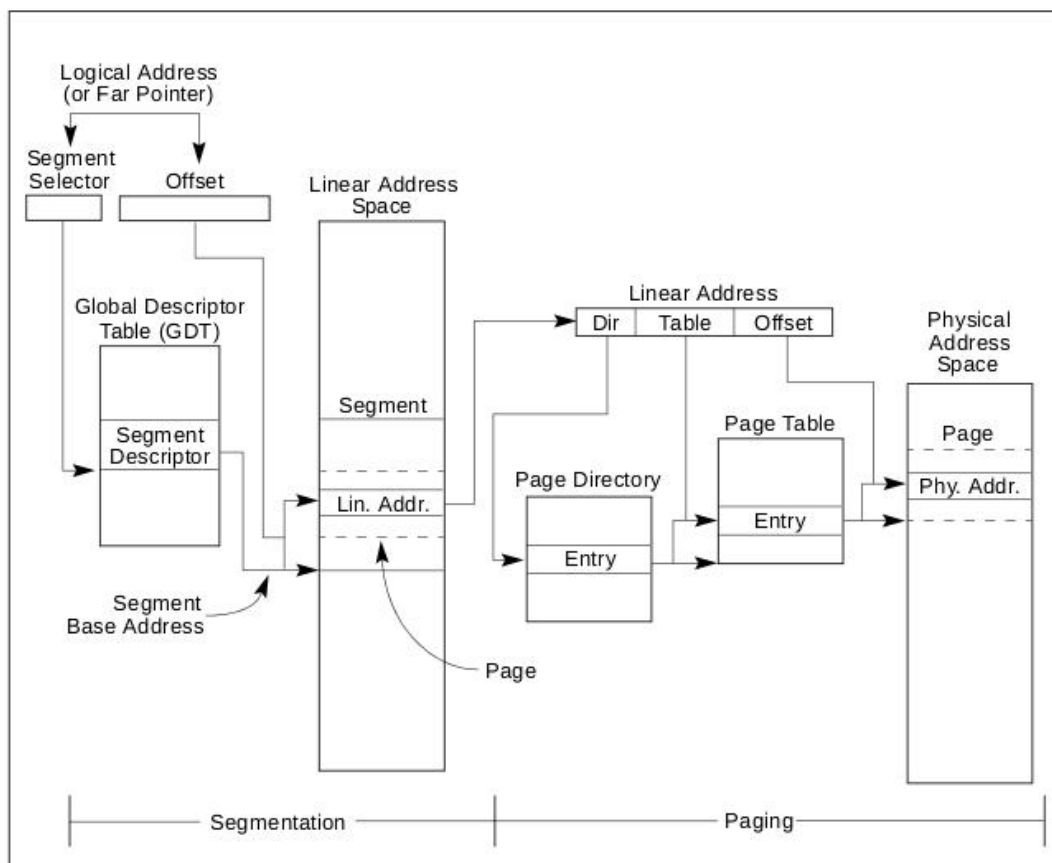


Figure 3-1. Segmentation and Paging

的确，老式系统就是这样处理大任务的，而且这个工作是由程序员自行完成。但是随着程序语言越来越高级，程序员对系统体系的依赖程度降低了，很少有程序员能非常清楚的驾驭系统体系，因此放手让程序员负责将程序片段化和按需调入轻则降低效率，重则使得机器崩溃；再一个原因是随着程序越来越丰富，程序的行为几乎无法准确预测，程序员自己都很难判断下一步需要载入哪段程序。因此很难再靠预见性来静态分配固定大小的内存，然后再机械地轮换程序片进入内存执行。so，start 虚拟内存。

逻辑地址通过段机制后变为一个 32 位的地址，足以覆盖 4G 的内存空间，而系统内存一般也就几百 M 吧，所以当程序需要的虚拟地址不在内存时，只依靠段机制很难进行虚拟空间地换入换出，因为不大方便把整段大小的虚拟空间在内存和硬盘之间调来调去。

通过段机制转换得到的地址仅仅是作为一个中间地址——线性地址了，该地址不代表实际物理地址，而是代表整个进程的虚拟空间地址。在线性地址的基础上，页机制接着会处理线性地址映射：当需要的线性地址（虚拟空间地址）不在内存时，便以页为单位从磁盘中调入需要的虚拟内存；当内存不够时，又会以页为单位把内存中虚拟空间的换出到磁盘上。使用页机制，4G 空间被分成 2 的 20 次方个 4K 大小的页面（页面大小其实有多种选择，只是 4KB 是工程师的经验得来），因此定位页面需要的索引表(页表)中每个索引项至少需要 20 位，但是在页表项中往往还需要附加一些页属性，所以页表项实际为 32 位，其中 12 位用来



存放诸如“页是否存在于内存”或“页的权限”等信息。

上面所说的 2 的 20 次方个 4K 大小的页面，就是 1MB 个也，4GB 的空间需要映射的页存储空间也就是 4K\*1MB，但相对而言这只能算是一个虚拟地址空间的映射，我们知道每个进程都需要自己的映射，如果有 100 个进程，那就是 400MB 的映射存储空间，Are you Kidding me ?当然不现实了，所以产生了所谓的分级页表，即页目录和页表的概念，32 位的地址来说，分为 3 段来寻址，分别是地址的低 12 位，中间 10 位和高 10 位。高 10 位表示当前地址项在页目录中的偏移，最终偏移处指向对应的页表，中间 10 位是当前地址在该页表中的偏移，我们按照这个偏移就能查出来最终指向的物理页了，最低的 12 位表示当前地址在该物理页中的偏移。就这样，我们就实现了分级页表。32 位下一个进程实际使用到的内存不会有 4GB 这么大，所以通过两级页表的映射，我们就可以完成映射。

我们这里的页目录生成后就是需要送到 CR3 寄存器中，因为页目录为我们提供了所有进程页表的映射关系（虚拟地址->物理地址），电路中的 MMU 逻辑单元的寻址就是从 CR3 寄存器的页目录数据开始管理的。之后我们具体作映射、内存分配时需要页异常（page\_fault），如果产生处理器会把引起异常的线性地址存放在 CR2 中。操作系统中的页异常处理程序可以通过检查 CR2 的内容来确定线性地址空间中哪一个页面引发了异常。，所以 OS 的设计远远离不开硬件的支持。

## （2）具体实现虚拟内存管理

具体代码实现参见代码清单 include/vmm.h 和 memm/vmm.c。

最后并修改入口函数 sl\_main()进行测试：

```
#include "gdt.h"
#include "idt.h"
#include "multiboot.h"
#include "debug.h"
#include "timer.h"
#include "mm.h"

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    init_debug();

    // 初始化全局段描述符表

    init_gdt();
    init_idt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
        study.\nYou can copy it freely!\n\n");
```

```

// 初始化物理内存管理
init_pmm(mboot_ptr);
init_vmm();

// 初始化物理内存可用页
init_page_pmm(mboot_ptr);

//init_timer(200);

// 开启中断

//asm volatile ("sti");

return 0;
}

```

## 8. 内核堆管理的实现

之前实现了内存的简单管理,但是目前的内存分配是按页为单位的,这样在需要分配小内存的时候比较容易造成内部碎片。因此需要实现内核的堆管理算法,目的是为了小内存的分配。除了简单的分配内存之外,还需要考虑在内存释放的时候,对连续的内存进行合并。并且堆要实现在空闲内存过多的时候把物理页释放给物理内存管理模块。关于堆的实现有很多种,我们选择最简单的侵入式链表管理方法(较为简单的方式实现)。

首先,在每一个申请的内存块之前插入一个描述当前内存块的结构体。结构体定义以及相关的函数声明参见原码清单 `include/heap.h`。

结构体的定义里使用了 C 语言的位域定义,因为表示这块内存有没有被使用只需要 1 个 bit 就可以,这样节省内存。这里定义的堆的函数形式很简单,就不详细解释了。需要注意的是这里定义了虚拟堆起始地址为 `0xE0000000`,它是内核页表没有使用的空闲区域。

接下来是具体实现。除了以上的外部接口函数,还需要在实现文件里声明内部函数详见 `memm/heap.c`。

然后添加一个简单的测试函数,参见 `memm/heap.c`。

最后并修改入口函数 `sl_main()`进行测试:

```

#include "multiboot.h"
#include "gdt.h"
#include "idt.h"
#include "timer.h"
#include "debug.h"
#include "mm.h"

```

```

// 定义 elf 相关信息数据
elf_t kernel_elf;

int sl_main(multiboot_t *mboot_ptr)
{
    // all our initialisation calls whill go in here.

    // 从 GRUB 提供的信息中获取到内核符号表和代码地址信息
    kernel_elf = elf_from_multiboot(mboot_ptr);
    // 初始化全局段描述符表
    init_gdt();
    // 初始化全局中断描述符表
    init_idt();

    monitor_clear();
    printk_color(rc_black, rc_red, "Hello ShenleOS!\n");
    printk_color(rc_black, rc_green, "This is a simple OS kernel, just for
study.\nYou can copy it freely!\n\n");
    /*****/
    asm volatile("int $0x3");
    asm volatile("int $0x4");
    // 初始化物理内存管理
    init_pmm(mboot_ptr);
    // 初始化虚拟内存管理
    init_vmm();
    // 初始化物理内存可用页
    init_page_pmm(mboot_ptr);
    // 初始化内核态堆管理
    init_heap();
    // 测试内核堆函数
    test_heap();

    // 初始化时钟中断
    //init_timer(20);

    // 解除对 INTR 中断的屏蔽
    asm volatile("sti");

    return 0;
}

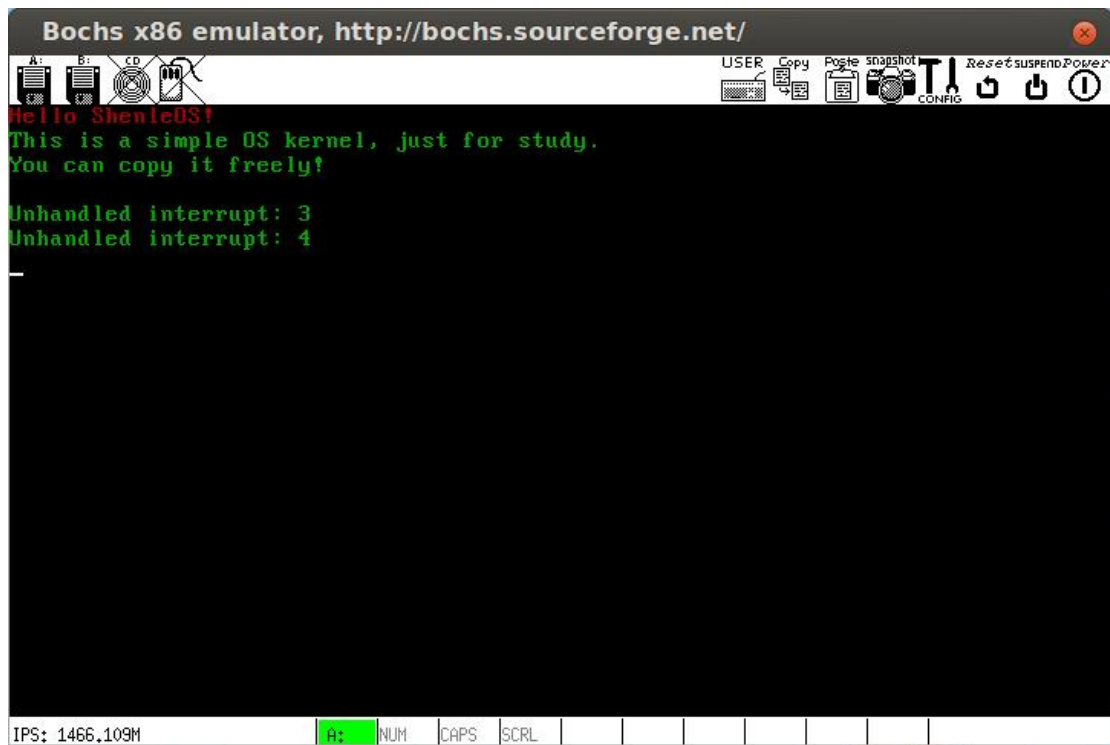
```

## 四. 测试数据及运行结果

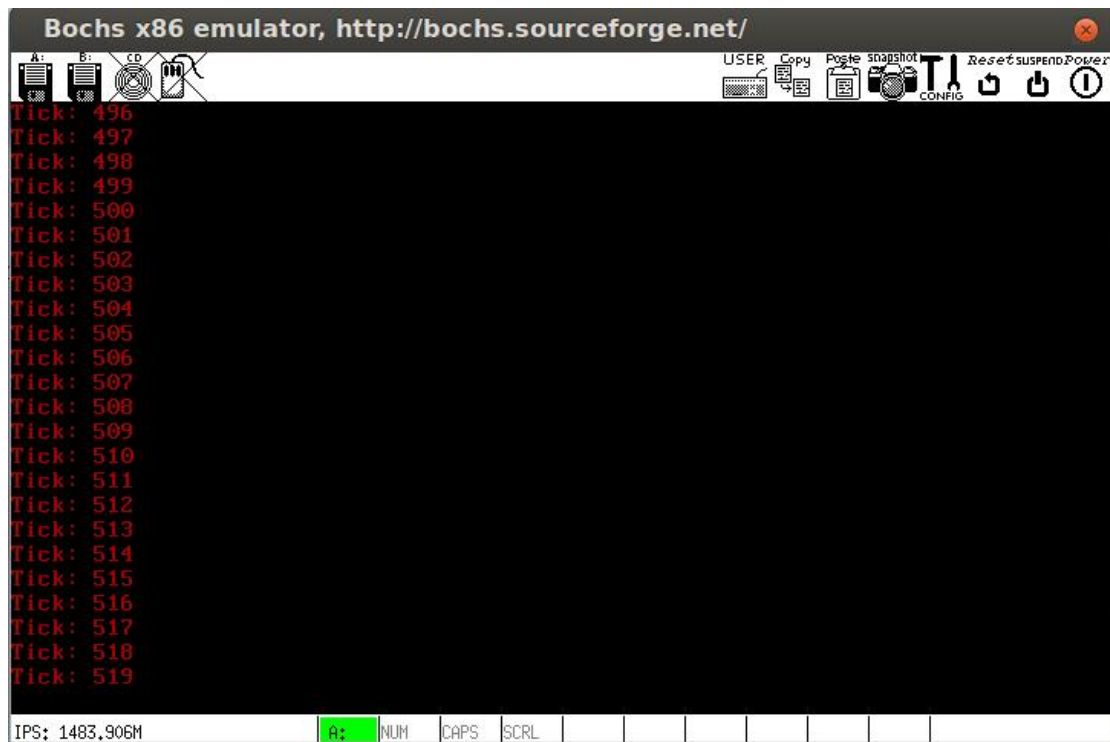
### 1. 测试屏幕操作函数打印"hello ShenleOS!"



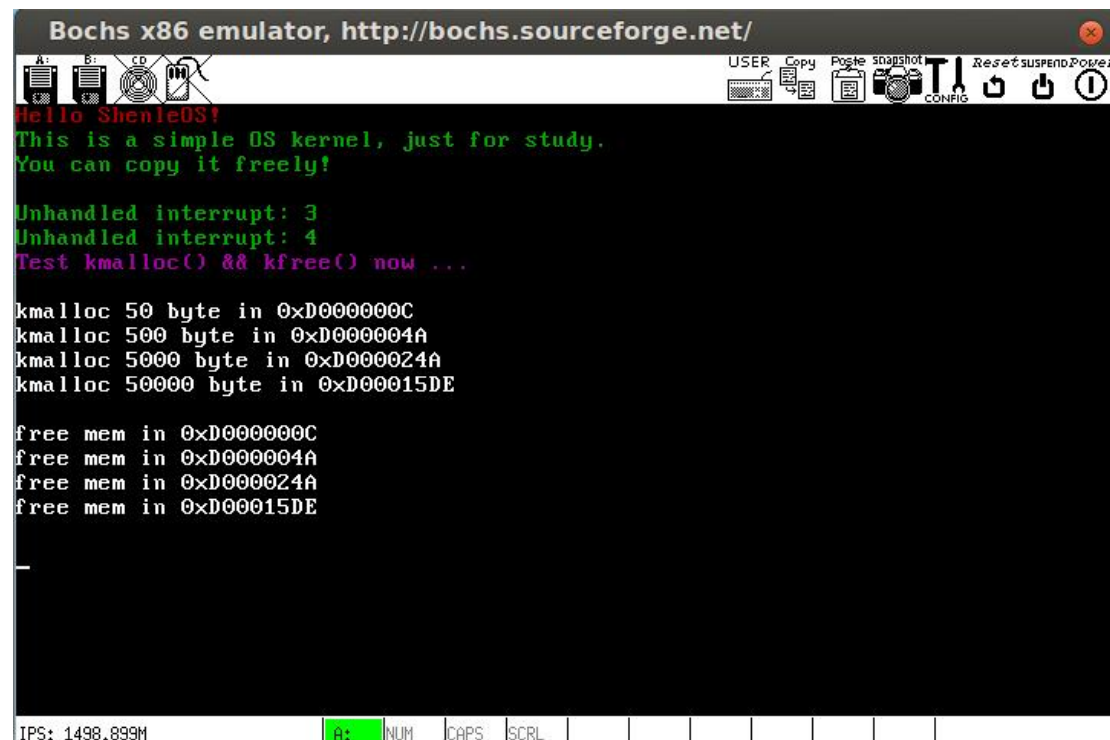
## 2. 测试中断处理函数



## 3. 测试时钟中断



## 4. 测试内存管理



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Hello ShenleOS!
This is a simple OS kernel, just for study.
You can copy it freely!

Unhandled interrupt: 3
Unhandled interrupt: 4
Test kmalloc() && kfree() now ...

kmalloc 50 byte in 0xD000000C
kmalloc 500 byte in 0xD000004A
kmalloc 5000 byte in 0xD000024A
kmalloc 50000 byte in 0xD00015DE

free mem in 0xD000000C
free mem in 0xD000004A
free mem in 0xD000024A
free mem in 0xD00015DE

-
IPS: 1498.899M
```

## 五. 总结

### 1. 实验过程中遇到的问题及解决办法；

这次操作系统实验经历很长时间，也涉及到了许许多多的方面，固然遇到了许多问题：

对于操作系统是如何加载的就理解了很长时间。

关于保护模式，还有 GDT，IDT 也碰到了许多问题。

还有后来的内存管理部分卡了很久都没有解决问题，最后才发现是函数的返回值写错了，导致出现分页错误的提示。

但是最后经过多次调试和同学的帮助都一一解决了。因此，从解决问题的过程中可以学到更多的东西。在此也非常感谢刘欢同学的帮助。

### 2. 对设计及调试过程的心得体会。

这次实验涉及到了不仅仅是操作系统方面的知识，可以说把很多计算机的专业知识都整合在了一起消化，因此学习到了许多平时没有接触过的东西。即使从 0 开始理解的程度也不深，但是整个坚持流程走下来，让我对于 OS 的启动加载

过程，还有保护模式的机制，以及中断处理，内存管理等等有了更多的理解。虽然没有达到完全的吃透的效果，并且是参照着其他人的模版一步步的走下来，但是收获仍然远大于仅仅看书的效果，当然首先看书学习消化理论只是也是非常必要的。

## 六. 附录

### 1. 源代码

详见电子版，还可参考 github: <https://github.com/vmezhang>

### 2. 参考资料

- (1) 《jamesm-tutorials Documentation Release 2.0.0》——jamesm-tutorials.
- (2) 《一个基于 x86 架构的简单内核实现》——hurley
- (3) 《清华大学操作系统课程实验》
- (4) 《x86 汇编语言：从实模式到保护模式》