

# 武汉大学计算机学院

## 本科生课程实验报告

### 操作系统课程设计

专业名称： 计算机科学与技术

课程名称： 操作系统课程设计

指导教师： 蒋晶珏 副教授

学生学号： 2021302111394

学生姓名： 秦瑾

二〇二三年八月

# 郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名: 秦 瑾

日期: 2023.8.8

## 摘 要

操作系统课程设计旨在通过模拟实践，引导学生一步步设计一个操作系统。逐步构建操作系统加深了对引导盘、保护模式、操作系统内核、进程、键盘 IO 以及处理器调度等概念的理解，同时实现了对操作系统设备管理的掌握。此实验帮助学生更深刻地领会课程理论，并直观认识实际操作系统内核。

实验设计遵循操作系统关键理论，包括保护模式、进程、内核、IO 处理。涵盖 NASM 汇编语言和 C 语言等内容。实验步骤包括搭建 Linux 虚拟机和 Bochs 仿真环境，在自己搭建的环境中调试运行《ORANGE'S: 一个操作系统的实现》的前七章代码，并在此基础上自主实现附加实验——实验 9 的要求。

作为计算机系统中至关重要的组成部分，操作系统承担着管理和控制计算机系统软硬件资源的重要任务，其设计过程需要进行繁多且详尽的考量。在实验过程中，我深深感受到操作系统的每一个组成部分都紧密地相互关联、相互影响。从保护模式到内核雏形，再到进程管理和设备管理，每个环节协同工作才能使得一个操作系统运转良好，为用户提供稳定可靠的计算环境。通过编写和调试代码，我逐渐深入地理解了操作系统的运行机制，也学会了如何进行处理器进程调度，如何管理设备输入，以及如何使用汇编语言和 C 语言来实现操作系统的各个功能模块。

这些实践经验不仅深化了我的理论知识，也增强了我解决实际问题的动手能力。在未来的学习和工作中，我将继续保持深入研究操作系统领域的学习热情，不断提升自己的知识水平和技能，并期待能够参与更复杂、更庞大的操作系统项目，

为构建高效、安全的计算环境做出贡献。

**关键词:** 操作系统复现; NASM 汇编语言; Linux; bochs;

# 目 录

|          |                               |          |
|----------|-------------------------------|----------|
| <b>1</b> | <b>操作系统复现内容总览</b>             | <b>1</b> |
| 1.1      | 实验环境的搭建 . . . . .             | 1        |
| 1.2      | 动手写一个最小的操作系统 . . . . .        | 1        |
| 1.3      | 实现保护模式 . . . . .              | 1        |
| 1.4      | 切换到保护模式 . . . . .             | 2        |
| 1.5      | 内核雏形 . . . . .                | 2        |
| 1.6      | 进程及进程调度 . . . . .             | 2        |
| 1.7      | 输入/输出系统 . . . . .             | 2        |
| 1.8      | 操作系统进阶 . . . . .              | 3        |
| <b>2</b> | <b>实验环境及环境搭建</b>              | <b>4</b> |
| 2.1      | 实验所需平台工具 . . . . .            | 4        |
| 2.1.1    | VMware Workstation . . . . .  | 4        |
| 2.1.2    | Ubuntu . . . . .              | 4        |
| 2.1.3    | Bochs . . . . .               | 5        |
| 2.1.4    | NASM、GCC 和 GNU MAKE . . . . . | 5        |
| 2.2      | 调试过程及运行结果 . . . . .           | 5        |
| 2.3      | 实验总结 . . . . .                | 6        |
| <b>3</b> | <b>动手写一个最小的“操作系统”</b>         | <b>8</b> |
| 3.1      | 实验内容 . . . . .                | 8        |
| 3.2      | 代码分析 . . . . .                | 8        |
| 3.2.1    | 核心数据结构 . . . . .              | 8        |
| 3.2.2    | 关键代码分析 . . . . .              | 8        |

|          |                     |           |
|----------|---------------------|-----------|
| 3.3      | 调试过程及运行结果 . . . . . | 9         |
| 3.4      | 实验总结 . . . . .      | 11        |
| <b>4</b> | <b>实现保护模式</b>       | <b>12</b> |
| 4.1      | 实验内容 . . . . .      | 12        |
| 4.2      | 代码分析 . . . . .      | 12        |
| 4.2.1    | 核心数据结构 . . . . .    | 12        |
| 4.2.2    | 关键代码段分析 . . . . .   | 14        |
| 4.3      | 调试过程及运行结果 . . . . . | 16        |
| 4.4      | 实验总结 . . . . .      | 19        |
| <b>5</b> | <b>切换到保护模式</b>      | <b>20</b> |
| 5.1      | 实验内容 . . . . .      | 20        |
| 5.2      | 代码分析 . . . . .      | 20        |
| 5.2.1    | 核心数据结构 . . . . .    | 20        |
| 5.2.2    | 关键代码分析 . . . . .    | 23        |
| 5.3      | 调试过程及运行结果 . . . . . | 23        |
| 5.4      | 实验总结 . . . . .      | 26        |
| <b>6</b> | <b>内核雏形</b>         | <b>27</b> |
| 6.1      | 实验内容 . . . . .      | 27        |
| 6.2      | 代码分析 . . . . .      | 27        |
| 6.2.1    | 核心数据结构 . . . . .    | 27        |
| 6.2.2    | 关键代码分析 . . . . .    | 27        |
| 6.3      | 调试过程及结果分析 . . . . . | 28        |
| 6.4      | 实验总结 . . . . .      | 30        |
| <b>7</b> | <b>进程与进程调度</b>      | <b>31</b> |
| 7.1      | 实验内容 . . . . .      | 31        |
| 7.2      | 代码分析 . . . . .      | 31        |
| 7.2.1    | 核心数据结构 . . . . .    | 31        |
| 7.2.2    | 关键代码分析 . . . . .    | 31        |

|           |                     |           |
|-----------|---------------------|-----------|
| 7.3       | 调试过程及结果分析 . . . . . | 34        |
| 7.4       | 实验总结 . . . . .      | 37        |
| <b>8</b>  | <b>操作系统的输入/输入系统</b> | <b>38</b> |
| 8.1       | 实验内容 . . . . .      | 38        |
| 8.2       | 代码分析 . . . . .      | 38        |
| 8.2.1     | 核心数据结构 . . . . .    | 38        |
| 8.2.2     | 关键代码分析 . . . . .    | 38        |
| 8.3       | 调试过程及结果分析 . . . . . | 42        |
| 8.4       | 实验总结 . . . . .      | 45        |
| <b>9</b>  | <b>操作系统进阶</b>       | <b>46</b> |
| 9.1       | 实验内容 . . . . .      | 46        |
| 9.2       | 代码分析 . . . . .      | 46        |
| 9.2.1     | 系统调用部分 . . . . .    | 46        |
| 9.2.2     | 优先级调度部分 . . . . .   | 48        |
| 9.3       | 调试过程及结果分析 . . . . . | 50        |
| 9.4       | 实验总结 . . . . .      | 51        |
| <b>10</b> | <b>实验心得体会</b>       | <b>52</b> |

# 1 操作系统复现内容总览

## 1.1 实验环境的搭建

实验内容：在 Windows 操作系统下安装虚拟机软件（如 Virtualbox, VMware 等），然后在虚拟机上安装 Ubuntu 操作系统。安装 Bochs 模拟器 Bochs 模拟 X86 硬件平台，并利用其自带 bochsdbg 调试器调试操作系统设计代码。

基本步骤：安装虚拟机；配置 Linux 环境；安装 Bochs 模拟器；安装 nasm、gcc、gnu 等工具；操作系统引导盘的制作；运行引导扇区。

## 1.2 动手写一个最小的操作系统

实验内容：通过编译一段最基本的 asm 代码来初次体验操作系统的设计以及了解 NASM 编译的使用方法、dd 命令写入磁盘的方法以及 Bochs 的使用方法。

基本步骤：NASM 编译；dd 命令写入磁盘，软盘读写工具将文件写到空白软盘的第一个扇区；调用子程序显示字符串，并无限循环；bximage 命令创建一个空白映像文件，再使用 dd 命令将.bin 文件写入；撰写 bochsrc 文件，配置 bochs 基本信息，使用.bin 文件从指定软盘启动；启动后进行调试。

## 1.3 实现保护模式

实验内容：认识保护模式，实现从实模式到保护模式的转换，GDT 描述符；实现实模式大于 1MB 内存的寻址能力，并接着上一次实验，从保护模式返回到实模式，重新设置各个段寄存器的值；LDT 描述符；学会使用挂载指令和运行程序。

实验步骤：借助 DOS 扩展程序范围，安装 FreeDos，并将两个软盘映像都写进 bochs 配置文件；学习数据结构 GDT 及其描述符；学习数据结构 LDT，关注其区别于 GDT 的局部特性；把.asm 文件编译成.com 文件，使用挂载指令将该文件复制到软盘映像文件里；了解保护模式如何获取大于 1MB 寻址能力；认识实模式和保护模式以及互相跳转的方式。



## 1.4 切换到保护模式

实验内容：引导扇区突破 512 个字节的限制，将工作分给 loader；加载 loader 进入内存并运行；将控制权交给 loader。

基本步骤：突破 512 字节的限制：交给 loader 来完成；认识 FAT12，遍历根目录区所有的扇区，将每一个扇区加载入内存，从中寻找 Loader.bin；使用 jmp 指令跳到内存中 loader.bin 的开始处；向 loader 交出控制权。

## 1.5 内核雏形

实验内容：在 Linux 下用汇编写 Hello, World!；进一步，汇编和 C 同时使用；从 loader 到 kernel 内核，把 kernel 内核加载到内存；将控制权交给 kernel 内核；跳入保护模式，并显示内存的使用情况。

基本步骤：linux 汇编下的 hello world: helloworld 编译成 elf 格式；./hello 运行；汇编和 C 同时使用：通过 global 导出，extern 声明；./hello 运行；从 loader 到内核：用 loader 把 kernel 加载到内存；进入保护模式；控制权交给内核：把 Kernel.bin 加载到内存。

## 1.6 进程及进程调度

实验内容：进程切换；丰富中断处理程序，比如让时钟中断处理可以不停地发生而不是只发生一次，进程状态的保存与恢复，进程调度，解决中断重入问题。

基本步骤：学习最简单的进程：进程表、进程栈、内核栈；特权级变换；寄存器值的压栈和恢复；准备进程体；初始化相应描述符；准备进程表；完成特权级别的跳转；丰富中断处理：设置 EOI、现场的保护和恢复、中断重入；多进程处理：读取不同的任务地址入口、堆栈栈顶和进程名；进程切换；系统调用；进程调度。

## 1.7 输入/输出系统

实验内容：实现简单的 I/O，从键盘输入字符的中断开始；获取并打印扫描码；创建对应打印扫描码解析数组，打印对应字符。

基本步骤：添加中断指令，多次输入；打印 make code 和 break mode；建立

键盘输入缓冲区分析字符；处理 shift、alt 和 ctrl 相关组合键；

## 1.8 操作系统进阶

实验内容：自定义一个系统调用，能够统计一个进程在执行的过程中被调度的次数。编写一个简单的用户程序，调用该自定义的系统调用，从而将进程及其调度的次数输出在屏幕上。并在实现了三个进程的优先级调度的基础上，将三个进程的循环次数从无限循环修改为有限次数，当三个进程执行完成时，计算三个进程在优先级调度算法下的周转时间、等待时间以及该系统的平均周转时间、平均等待时间和吞吐量。也可以添加新的进程，然后计算该系统的平均周转时间、平均等待时间和吞吐量。

基本步骤：从系统调用开始：定义系统调用函数体；定义函数声明；添加 sys\_call\_table 成员；增加 NR\_SYS\_CALL 的值；返回输出用户进程的函数调用次数；初始化进程表同时定义进程优先级和调用次数；设计进程调度算法；返回输出系统的平均周转时间、平均等待时间和吞吐量、

## 2 实验环境及环境搭建

### 2.1 实验所需平台工具

#### 2.1.1 VMware Workstation

VMware 是虚拟机软件，能在一台机器上同时运行多个不同操作系统，与多启动系统不同，系统切换时不需要重启。它支持隔离的操作环境、互动操作和复原功能。如下图 2-1 所示为其主界面。

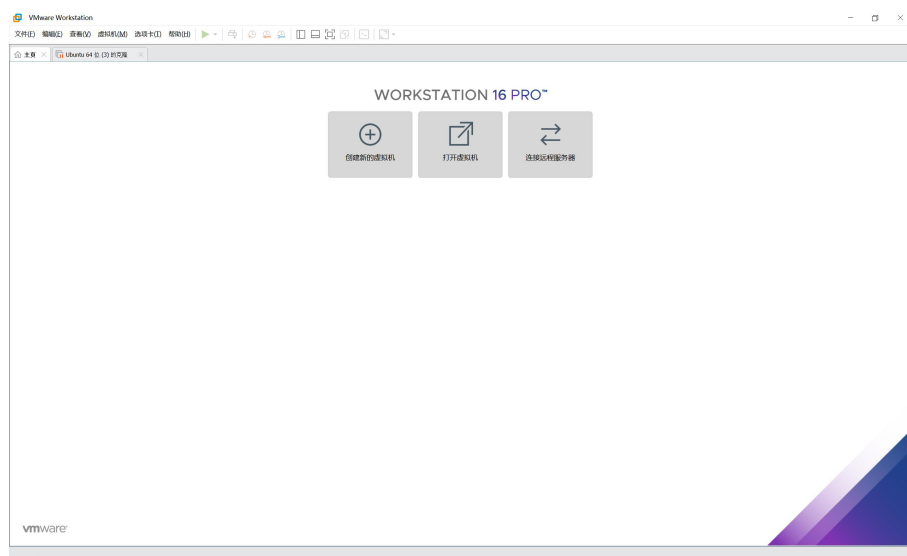


图 2.1 VMware Workstation 16 主界面

#### 2.1.2 Ubuntu

Ubuntu 是基于 Debian 的桌面 Linux 操作系统，提供稳定和新颖的自由软件，有庞大的社区力量。如下图 2-2 所示为 Ubuntu 操作系统的截图。

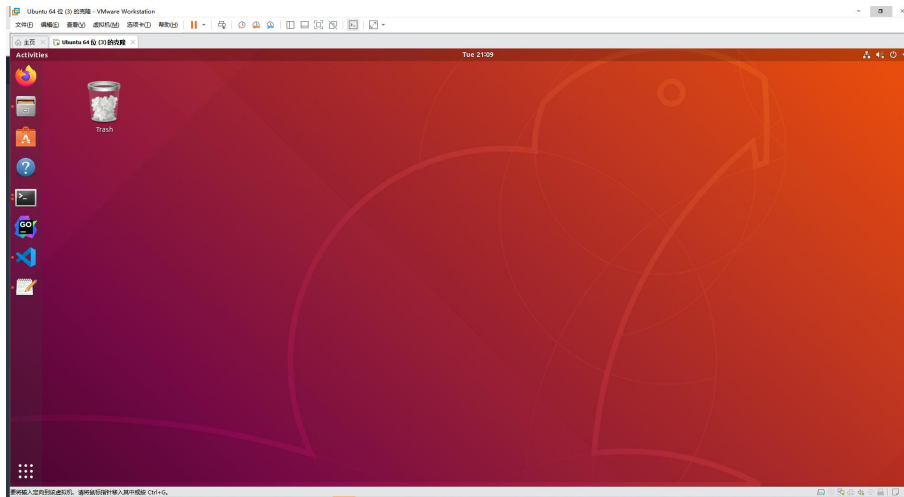


图 2.2 Ubuntu 操作系统（虚拟机）主界面

### 2.1.3 Bochs

Bochs 是一款开源的 x86 硬件平台模拟器，模拟整个 PC 平台，包括 I/O 设备、内存和 BIOS。它能在多种平台上运行，并支持模拟多台 PC。

### 2.1.4 NASM、GCC 和 GNU MAKE

Netwide Assembler (NASM) 是 x86 架构的汇编与反汇编工具，支持多种二进制格式输出，适用于编写 16 位、32 位和 64 位程序，以及创建引导加载程序等。遵循 Intel 风格汇编。

## 2.2 调试过程及运行结果

在安装完虚拟机和相应的 Linux 环境之后，为了能够对编写的源代码进行编译和仿真，还必须下载 Bochs。选择 2.3.5 版本。编译过程如下：

```
1 \tar vxzf bochs-2.3.5.tar.gz
2 \cd bochs-2.3.5
3 \./configure --enable-debugger --enable-disasm
4 \make
5 \sudo make install
```

如下图 2-3 所示为 Bochs 的运行界面。

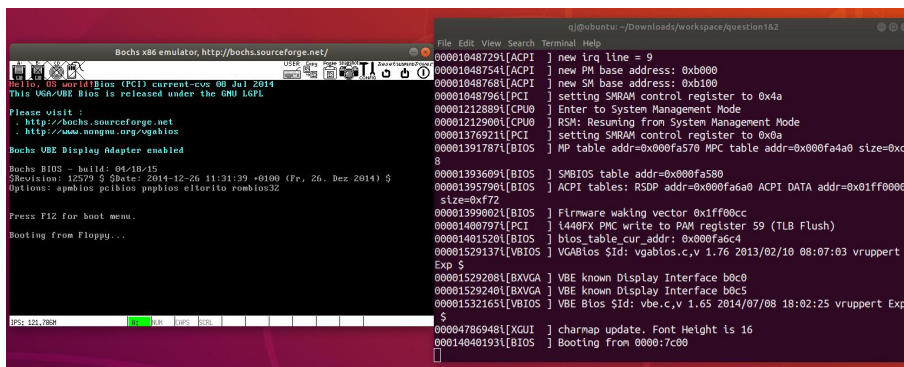


图 2.3 Bochs 运行界面

为了编译汇编语言，必须安装 NASM 程序，同时，还必须安装 GNU Make，用于自动化编译和链接。

在 Ubuntu 中已经预安装了 GCC，对于 GNU Make，可以通过以下的指令完成：

```
1 \sudo apt-get install build-essential nasm
```

从官网 <https://www.nasm.us/> 下载 NASM 安装包，然后提取文件到指定文件夹，在文件夹下打开终端，依次输入下列指令：

```
1 \./configure
2 \make
3 \sudo make install
```

输入 `nasm -version` 指令来检验是否安装成功。

## 2.3 实验总结

搭建环境实验时，遇到了以下问题。报错：`00000000000p[] »PANIC« bochsrc:10: vgaromimage directive malformed`。通过该报错信息，尝试修改 `bochsrc` 与本地 `bochs` 组件路径对应：

1. `romimage: file=/usr/share/bochs/BIOS-bochs-latest`
  2. `vgaromimage: /usr/share/vgabios/vgabios.bin`
  3. `keyboard_mapping:enabled=1,map=/usr/share/bochs/keymaps/x11-pc-us.map`
- 修改为：

1. `romimage:file=/home/qj/Downloads/workspace/bochs-2.6.8/bios/BIOS-bochs-latest`

2. vgaromimage:file=/home/qj/Downloads/workspace/bochs-2.6.8/bios/VGABIOS-lgpl-latest
3. keyboard:keymap=/home/qj/Downloads/workspace/bochs-2.6.8/gui/keymaps/x11-pc-us.map

即可解决问题。

总而言之，本章实验中出现了一些配置文件路径更改的问题，需要花费一些时间去查验。但我认为单纯解决问题还不够，应该了解问题出现的具体原因、解决方法以及解决方法的原理。因此，我去查阅了相关资料 [2]，认识到 vgaromimage 指令用于指定 Bochs 模拟器使用的 VGA BIOS ROM 映像文件。VGA BIOS 用于控制 VGA 显示适配器的固件程序，它包含了一些基本的图形和显示功能。在 Bochs 模拟器中，通过设置 vgaromimage 指令，可以指定要加载的 VGA BIOS ROM 映像文件的路径和文件名。Bochs 在启动时会读取该指令并加载相应的 VGA BIOS ROM 映像文件。而 VGA BIOS ROM 映像文件通常是一个二进制文件，其中包含了 VGA BIOS 的固件代码。这些代码实现了 VGA 显示适配器的初始化、显示模式设置、图形绘制和显示输出等功能。

## 3 动手写一个最小的“操作系统”

### 3.1 实验内容

通过编译一段最基本的 asm 代码来初次体验操作系统的设计以及了解 NASM 编译的使用方法、dd 命令写入磁盘的方法以及 Bochs 的使用方法。

### 3.2 代码分析

#### 3.2.1 核心数据结构

常量 BootMessage: db "Hello, OS world!"

#### 3.2.2 关键代码分析

```
1  org 07c00h      ; 告诉编译器程序加载到 7c00 处
2  #使 ds 和 es 两个寄存器指向与 cs 相同的段
3  mov ax, cs
4  mov ds, ax
5  mov es, ax
6  call DispStr ; 调用显示字符串例程
7  jmp $ ; 无限循环
8  DispStr:
9  mov ax, BootMessage
10 mov bp, ax      ; ES:BP = 串地址
11 mov cx, 16      ; CX = 串长度
12 mov ax, 01301h ; AH = 13, AL = 01h
13
14 mov bx, 000ch      ; 页号为 0(BH = 0) 黑底红字(BL = 0Ch, 高亮)
15 mov dl, 0
16 int 10h          ; 10h 号中断
17 ret
18 BootMessage: db "Hello, OS world!"
19 times 510-($-$$) db 0 ; 填充剩下的空间, 使生成的二进制代码恰
    好为512 字节
20 dw 0xaa55 ; 结束标志
```

值得注意的是, 未被方括号括起来的变量名被看作是地址, 如果不加方括号时出

现数字，则视作 offset；\$ 表示当前行被汇编后的地址，\$\$ 表示一个节的开始处被汇编之后的地址。

### 3.3 调试过程及运行结果

想要进行操作系统的调试，首先必须创建一个软盘映像，在 terminal 中输入 bxiimage 即可打开相应的界面。根据引导界面，依次输入 1->fd-> 回车-> 回车，即可生成一个名为 a.img 的软盘映像，如下图 3-1 所示。

```
qj@ubuntu:~/Downloads/workspace/bochs-2.6.8$ bxiimage
=====
                bxiimage
Disk Image Creation / Conversion / Resize and Commit Tool for Bochs
$Id: bxiimage.cc 12690 2015-03-20 18:01:52Z vruppert $
=====

1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd

Choose the size of floppy disk image to create, in megabytes.
Please type 160k, 180k, 320k, 360k, 720k, 1.2M, 1.44M, 1.68M, 1.72M, or 2.88M.
[1.44M]

What should be the name of the image?
[a.img]

Creating floppy image 'a.img' with 2880 sectors

The following line should appear in your bochsrc:
floppya: image="a.img", status=inserted
```

图 3.1 bxiimage 操作界面

创建完软盘映像后，需要编译源代码，输入以下命令即可完成编译。

```
1 \nasm boot.asm -o boot.bin
```

完成编译之后，我们需要使用软盘绝对扇区读写工具将这个文件写到一张空白软盘的第一个扇区，输入以下代码即可完成读入。

```
1 \dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

一切准备就绪之后，需编写 Bochs 的配置文件 bochsrc.disk，即告诉 Bochs 对虚拟机中内存，硬盘映像和软盘映像等的配置信息：

```
# what disk images will be used

floppya: 1_44=a.img, status=inserted
```



# choose the boot disk.

boot: a

上述步骤完成之后，一切准备就绪，正式进行调试，输入命令：

```
1 \bochs -f bochsrc
```

按下回车键，再选择 6，会出现如下图 3-2 所示的界面。

```
This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate. Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found. When you are satisfied with the configuration, go
ahead and start the simulation.

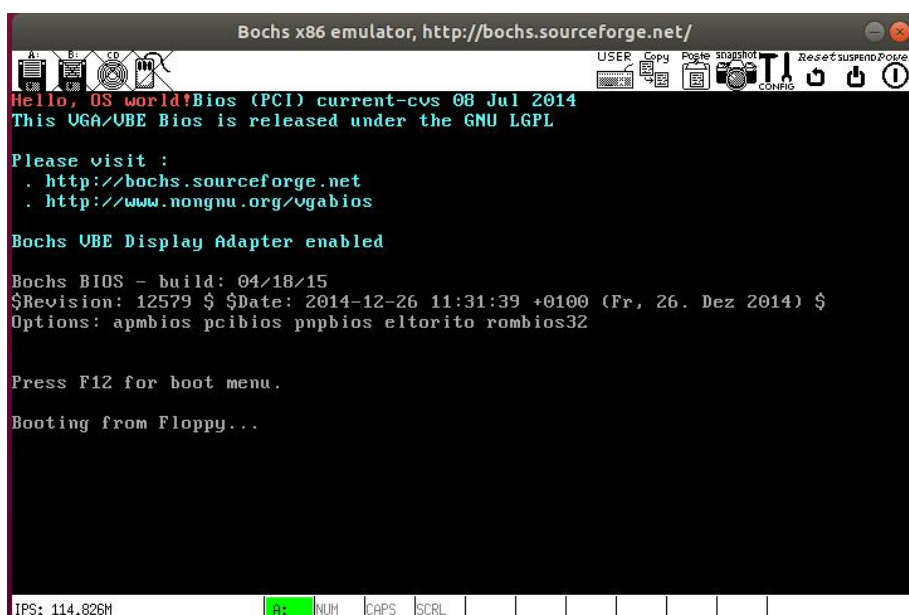
You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6] 6
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Next at t=0
(0) [0x000fffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b      ; ea5be000
f0
<bochs:1>
```

图 3.2 调试过程

输入 c，即会进入 bochs 页面，这时，屏幕的左上角会出现一行红色的“Hello, OS world!”，即代表调试成功。调试结果如下图 3-3 所示。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
USER Copy Paste Snapshot CONFIG Reset suspend power
A: B: C: D:
Hello, OS world! Bios (PCI) current-cvs 08 Jul 2014
This VGA/UBE Bios is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios

Bochs UBE Display Adapter enabled

Bochs BIOS - build: 04/18/15
$Revision: 12579 $ $Date: 2014-12-26 11:31:39 +0100 (Fr, 26. Dez 2014) $
Options: apmbios pcibios pnpbios eltorito rombios32

Press F12 for boot menu.
Booting from Floppy...

IPS: 114,826M A: NUM CAPS SCRL
```

图 3.3 调试结果

### 3.4 实验总结

在本实验中，我们编写了一个最简单的“操作系统”，并且使用 Bochs 软件进行仿真。但是事实上，我们编写的程序只是一个最简单的引导扇区，之所以可以称之为最简单的操作系统，是因为它可以直接在裸机上运行。计算机电源打开时，首先会加电自检，然后寻找启动盘，如果是选择从软盘启动，计算机就会检查软盘的 0 面 0 磁道 1 扇区，如果发现它以 0xAA55 结束，那么 BIOS 就会认为它是一个引导扇区。

当然，一个正确的引导扇区除了以 0xAA55 结束之外，还应该包含一段少于 512 字节的执行码。而一旦 BIOS 发现了引导扇区，就会将这 512 字节的内容装载到内存地址 0000:7c00 处，然后跳转到 0000:7c00 处，将控制权彻底交给这段引导代码。至此为止，计算机不再由 BIOS 中固有的程序来控制，而是变为由操作系统的一部分来控制。我们编写的代码中第一行就是将程序加载到 0x7c00 处，第十八行是让代码以 0xAA55 结束，因而可以作为一个独立的引导扇区运行。

总而言之，第二个实验的任务是一个最小的操作系统，也就是在虚拟机上运行 Helloworld。首先是 nasm 的编译，编译.asm 变为.bin，可以制作引导扇区，编译变为.com，可以调试。之后是 dd 命令写入磁盘，主要是 dd if=boot.bin of=/dev/fd0 bs=512 count=1 conv=notrunc（防止截断）。软盘读写工具将文件写到空白软盘的第一个扇区，然后让 ds 和 es 两个段寄存器指向与 cs 相同的段，调用子程序显示字符串，然后无限循环。最后是 bochs 的用法，用 bximage 命令创建一个空白映像文件，再使用 dd 命令将.bin 文件写入，撰写一个 bochsrc 文件，指定从软盘启动，并且使用.bin 文件。启动后使用以下命令进行调试：b 设置断点，c 继续执行，dump\_cpu 查看寄存器，x 查看内存，n 运行下一条指令。实验二思维导图如下图 3-4 所示：



图 3.4 实验二思维导图

## 4 实现保护模式

为了使实验报告清晰简洁，后续实验均省略了部分具体代码实现部分，仅重点说明主要数据和代码结构，以体现设计的核心思想。

### 4.1 实验内容

认识保护模式，实现从实模式到保护模式的转换，GDT 描述符；实现实模式大于 1MB 内存的寻址能力，并接着上一次实验，从保护模式返回到实模式，重新设置各个段寄存器的值；LDT 描述符；学会使用挂载指令和运行程序。

### 4.2 代码分析

#### 4.2.1 核心数据结构

表 4.1 [SECTION .gdt] 段 gdt 数据段

|        |                                 |
|--------|---------------------------------|
| 数组 GDT | LABEL_GDT 空描述符                  |
|        | LABEL_DESC_CODE32 非一致代码段 32 描述符 |
|        | LABEL_DESC_VIDEO 显存首地址          |
| 变量     | Gdtlen 表示 GDT 的长度               |
|        | GdtPtr dw 为 GDT 界限 dd 为 GDT 基址  |
| 选择子    | SelectorCode32 非一致代码段 32 选择子    |
|        | SelectorVideo 视频段选择子            |

如表 4-1 所示，该部分对应“认识保护模式，实现从实模式到保护模式的转换，认识 GDT 描述符，并进入保护模式”的数据结构。

**表 4.2 [SECTION .gdt] 段 gdt 数据段**

|        |                   |                                |
|--------|-------------------|--------------------------------|
| 数组 GDT | LABEL_GDT         | 空描述符                           |
|        | LABEL_DESC_NORMAL | Normal 描述符                     |
|        | LABEL_DESC_CODE32 | 非一致代码段 32 描述符                  |
|        | LABEL_DESC_CODE16 | 非一致代码段 16 描述符                  |
|        | LABEL_DESC_DATA   | 数据段描述符                         |
|        | LABEL_DESC_STACK  | 32 位堆栈段描述符                     |
|        | LABEL_DESC_TEST   | 测试段描述符                         |
|        | LABEL_DESC_VIDEO  | 显存描述符                          |
| 变量     | Gdtlen            | 表示 GDT 的长度                     |
|        | GdtPtr            | 前两个字节为 GDT 界限<br>后四个字节为 GDT 基址 |
| 选择子    | SelectorNormal    | Normal 选择子                     |
|        | SelectorCode32    | 非一致代码段 32 选择子                  |
|        | SelectorCode16    | 非一致代码段 16 选择子                  |
|        | SelectorData      | 数据段选择子                         |
|        | SelectorStack     | 堆栈段选择子                         |
|        | SelectorTest      | 测试段选择子                         |
|        | SelectorVideo     | 视频段选择子                         |

**表 4.3 [SECTION .data1] 数据段**

|    |                   |                                       |
|----|-------------------|---------------------------------------|
| 常量 | SPValueInRealMode | 字符串                                   |
|    | PMMessage         | 在保护模式中显示的信息”In Protect Mode now. ^-^” |
|    | OffsetPMMessage   | PMMessage 段偏移                         |
|    | StrTest           | “ABCDEFGHIJKLMNOPQRSTUVWXYZ”          |
|    | OffsetStrTest     | StrTest 段偏移                           |
|    | DataLen           | 表示数据段的长度                              |

**表 4.4 [SECTION .gs] 全局堆栈段**

|    |            |      |
|----|------------|------|
| 常量 | TopOfStack | 栈顶指针 |
|----|------------|------|

如表 4-2、4-3、4-4 所示，该部分对应“实现实模式大于 1MB 内存的寻址，并接着上一次实验，在 DOS 模式下从保护模式返回到实模式，重新设置各个段寄存器的值”的数据结构。

**表 4.5 [SECTION .ldt] 段 LDT 数据段**

|        |                      |            |
|--------|----------------------|------------|
| LDT 数组 | LABEL_LDT_DESC_CODEA | LDT 描述符    |
|        | LDTLen               | 表示 LDT 的长度 |
|        | SelectorLDTCodeA     | LDT 选择子    |

如表 4-2、4-3、4-4、4-5 所示，该部分对应“了解 LDT 描述符；学会使用挂载指令将文件挂载在临时目录和并运行程序”的数据结构。

#### 4.2.2 关键代码段分析

1. [SECTION .s16] [BITS 16] — 代码段：从实模式跳到保护模式

- 初始化 32 位代码段描述符：将 [SECTION .s32] 段的物理地址分三部分赋给空白描述符 DESC\_CODE32
  - 为加载 GDTR 做准备：将 GDT 的物理地址填充到 GdtPtr 中
  - 加载 GDTR：将 GdtPtr 加载到寄存器 gdtr 中
  - 关中断
  - 打开 A20 地址线：通过操作端口 92h
  - 准备切换到保护模式：把寄存器 cr0 的第 0 位置为 1
  - 进入保护模式：Jump
2. [SECTION .s32] [BITS 32] — 32 位代码段，由实模式跳入。将视频段选择子 SelectorVideo 的地址赋给 gs
- 设置输出位置为屏幕 11 行 79 列
  - 设置输出为黑底红字
  - 往显存中 edi 偏移处写入 P
  - 无限循环
3. 从保护模式跳回到实模式
- 程序重设各个段寄存器值，使 ds、es、ss 指向 as
  - 恢复 sp 的值
  - 关闭 A20
  - 打开中断
  - 回到 DOS
4. [SECTION .s32] — 32 位代码段，由实模式跳入。初始化 ds、es、gs、ss：
- ds 指向数据段
  - es 指向新增的 5MB 的内存段
  - gs 指向显存
  - ss 指向测试段
  - 显示一个字符串
  - 调用读内存函数 TestRead
  - 调用写函数 TestWrite
  - 调用函数 TestRead
  - 跳转到 [SECTION .s16code] 代码段的 TestRead 函数

5. [SECTION .s16code] — 16 位代码段，由 32 位代码段跳入，跳出后到实模式

- 把选择子 SelectorNormal 加载到 ds、es、ss
- 清 cr0 的 PE 位，跳回实模式
- 跳转到 LABEL\_REAL\_ENTRY 即段 [SECTION .s16] 里

## 4.3 调试过程及运行结果

在之前的实验过程中，我们将文件写到了引导扇区运行，但引导扇区只有 512 个字节。空间有限。为使程序扩大，可以选择先借用其他操作系统的引导扇区，我们借用 DOS 操作系统，将程序编译成 COM 文件，然后用 DOS 来执行它。第一步需要到 Bochs 官方网站下载一个 FreeDos，解压后将其中的 a.img 复制到我们的工作目录中，并改名为 freedos.img。紧接着，我们需要使用 bximage 生成一个软盘映像，取名为 pm.img。再其次，我们需要修改我们的 bochsrc 与上述 img 文件对应。最后，直接启动 Bochs 开始调试，即可进入 FreeDos 界面，输入 format b 的指令以便格式化 freedos 的 B: 盘。格式化及格式化结果如下图 4-1 所示。



图 4.1 freedos 格式化 B 盘示意图

格式化完成后，将上述代码第 8 行中的 07c00h 改为 0100h，并重新编译为 com 文件：

```
1 \nasm pmttest1.asm -o pmttest1.com
```

紧接着, 我们需要输入以下三条指令, 将 pmtest1.com 复制到虚拟软盘 pm.img 中, 所需要的指令如下:

```
1 \sudo mount -o loop pmimg /mnt/floppy
2 \sudo cp pmtest1.com /mnt/floppy
3 \sudo umount /mnt/floppy
```

此时若出现挂载点不存在的错误, 输入以下指令即可解决:

```
1 \sudo mkdir /mnt/floppy
```

接下来我们需要重新回到 FreeDos 中 (即原先打开的 Bochs 虚拟机), 并且输入如下命令:

```
1 \b:\pmtest1.com
```

pmtest1.com 成功运行。可以注意到, 一个红色的字母 “P” 出现在了屏幕的右侧的中部, 这代表调试成功, 程序已经进入了保护模式。调试结果如下图 4-2 所示。

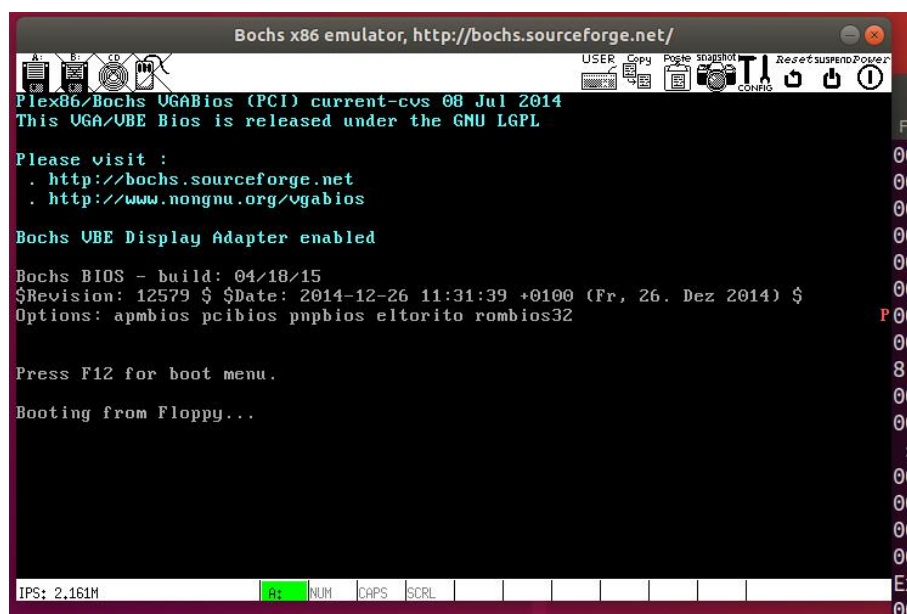


图 4.2 pmtest1.com 调试结果

下一步以同样方式运行 pmtest2.com。调试结果如下图 4-3 所示, 可以看到, 程序打印出了两行数字, 第一行全部是 0, 说明开始内存 5MB 处都是 0, 而下一行已经变成了 41、42、43..., 说明写操作成功 (十六进制的 41、42、43...48 正是 A、B、C...H)。同时可以看到, 程序执行结束后没有进入死循环, 而是重新出现了 DOS 提示符, 这说明我们已经重新到了实模式下的 DOS。



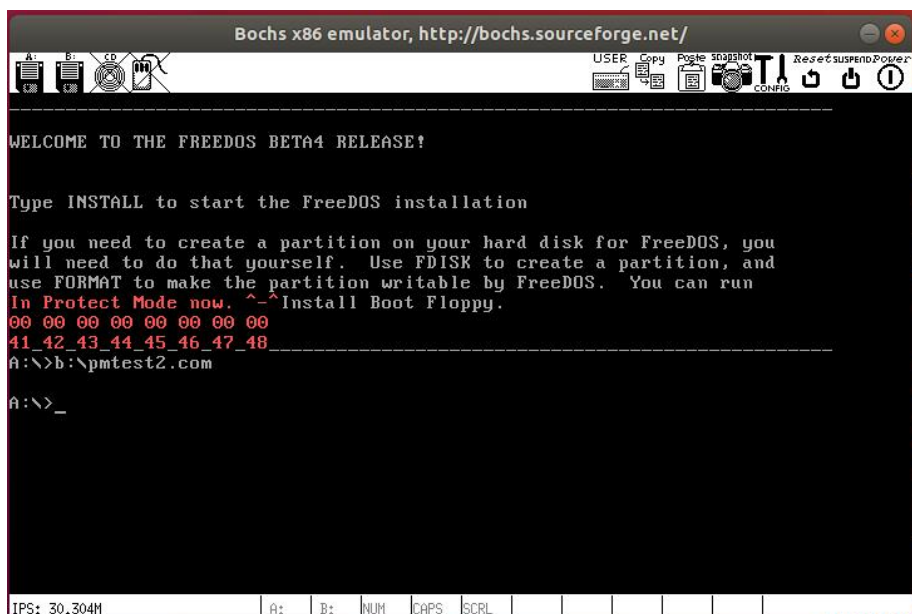


图 4.3 pmtest2.com 运行结果图

最后再以同样方式运行 pmtest3.com。此处 LDT 中的代码段只是打印一个字符 L，因此，在 [SECTION .s32] 中打印完 “In Protect Mode Now.” 这个字符串之后，一个红色的字符 L 将会出现。可以看到在下图 4-4 中，的确出现了 “In Protect Mode Now.” 字符串和一个红色的 L。

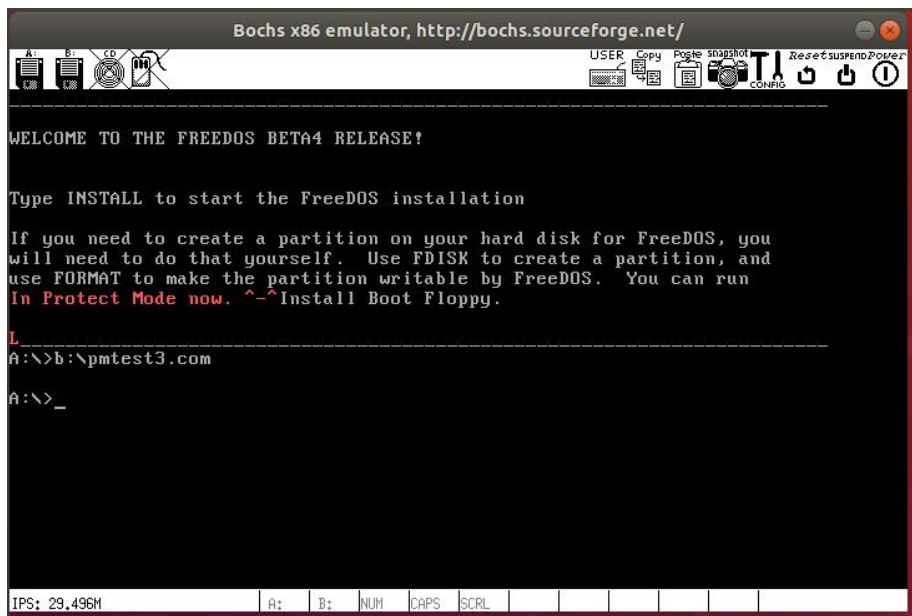


图 4.4 pmtest3.com 运行结果图

## 4.4 实验总结

CPU 通常情况下有两种工作模式：实模式和保护模式。打开计算机时 CPU 默认工作在实模式下，现在需要让其进入保护模式，发挥其寻址能力，并为 32 位系统提供硬件保障。本实验中，在 `pm.inc` 文件中定义了全局描述表 GDT，定义了相应的段描述符，并且通过 `pmtest1.com` 由实模式进入保护模式，最后在保护模式中在屏幕右边中央打印了一个红色的“P”，代表实验成功。在调试过程中，我发现只需要将 b 盘格式化一次，后面就不需要再格式化了。

接着，系统成功进入保护模式，打印了一个红色的 P，在本实验中，我们重新建立了一个以 5MB 为基址的段，随后先读出开始处 8 字节的内容，然后写入一个字符串，再从中读出 8 字节。如果读写成功，两次读出的内容不同，而且第二次读出的内容应该前面写进的字符串。然后返回实模式，我们加载了一个合适的描述符选择有关段寄存器，以使对应的段描述符和高速缓存寄存器中含有合适的段界限和属性。而且，我们不能从 32 位代码段返回实模式，只能从 16 位代码段中返回，这是因为无法实现从 32 位代码段返回时 `cs` 高速缓存寄存器中的属性符合实模式的要求（实模式不能改变段属性）。

最后，我了解并学习了描述符表 LDT，LDT 与 GDT 类似，但它的段选择子的 T1 位必须为 1。类似地，在运用它时，必须先用 `lldt` 指令加载 `ldtr`，而 `ldtr` 的操作数是 GDT 中用来描述 LDT 的段描述符。

为了补充相关理论知识，我去查阅相关资料 [1]。结合上述实验，我认识到保护模式是 x86 处理器中的一种特殊的操作模式，它与硬件紧密结合，提供了更高级的内存保护和多任务支持。通过将内存划分为多个段并为每个段设置不同的访问权限和保护级别，从而可以防止程序越界访问内存和修改其他程序的数据，提高系统的稳定性和安全性。同时，在保护模式下的操作系统可以实现多任务，每个程序拥有独立的内存空间和代码段，通过任务切换来实现并发执行。结合并发这一点，LDT 的引入便显得十分自然，GDT 是全局描述符表，存储了系统中所有进程和任务的段描述符；LDT 是局部描述符表，用于存储特定任务或进程的私有段描述符。通过使用 GDT 和 LDT，操作系统可以对不同任务或进程的内存访问权限进行细粒度控制，提高系统的安全性和灵活性，从而更好地实现并发。

## 5 切换到保护模式

### 5.1 实验内容

引导扇区突破 512 个字节的限制，将工作分给 loader；加载 loader 进入内存并运行；将控制权交给 loader。

### 5.2 代码分析

#### 5.2.1 核心数据结构

如下表 5-1 所示，此处对应的是“引导扇区加上 BPB 等头信息，可以被 DOS 识别，引导扇区突破 512 个字节的限制，将工作分给 loader”部分的主要数据结构。



**表 5.2 对 boot.asm 增添的部分**

|     |                         |                         |
|-----|-------------------------|-------------------------|
| 变量  | wRootDirSizeForLoop     | Root Directory 占用的扇区数   |
|     | wSectorNo               | 要读取的扇区号                 |
|     | bOdd                    | 奇数还是偶数                  |
| 字符串 | LoaderFileName          | LOADER.BIN 文件名          |
|     | BootMessage             | "Booting"               |
|     | Message1                | "Ready."                |
|     | Message2                | "No LOADER"             |
| 宏   | BaseOfStack             | 调试状态下堆栈基地址              |
|     | BaseOfLoader            | LOADER.BIN 被加载到的位置—段地址  |
|     | OffsetOfLoader          | LOADER.BIN 被加载到的位置—偏移地址 |
|     | RootDirSectors          | 根目录占用空间                 |
|     | SectorNoOfRootDirectory | Root Directory 的第一个扇区号  |

如表 5-2 所示，此处对应的是“加载 loader 进入内存并运行”部分在 boot.asm 基础上增添的主要数据结构。

**表 5.3 对 boot.asm 再次增添的部分**

|   |                |   |
|---|----------------|---|
| 宏 | SectorNoOfFAT1 | FAT1 的第一个扇区号 =BPB_RsvdSecCnt                                  |
|   | DeltaSectorNo  | DeltaSectorNo = BPB_RsvdSecCnt<br>+ (BPB_NumFATs * FATSz) - 2 |

如表 5-3 所示，此处对应的是“将控制权交给 loader”部分在上述 boot.asm 基础上再次增添的主要数据结构。

### 5.2.2 关键代码分析

1. 使 `ds`、`es`、`ss` 三个段寄存器指向与 `cs` 相同的段，令栈指针 `sp` 指向栈底。清屏并显示字符串 `Booting`。
2. 在 A 盘的根目录寻找 `LOADER.BIN`：遍历根目录取所有的扇区，将每一个扇区加载到内存，从中寻找文件名为 `Loader.bin` 的条目。每读一个扇区就在“`Booting`”后面打一个点。正式跳转到已加载到内存中的 `LOADER.BIN` 的开始处后显示字符串“`Ready`”，开始执行 `LOADER.BIN` 的代码。
3. `ReadSector` 函数：从第 `ax` 个 Sector 开始，将 `cl` 个 Sector 读入 `es:bx` 中。
4. `GetFATEntry` 函数：找到序号为 `ax` 的 Sector 在 FAT 中的条目，结果放在 `ax` 中。需要注意的是，在中间需要读 FAT 的扇区到 `es:bx` 处，所以函数一开始就保存了 `es` 和 `bx`。
5. `Loader.asm`：令 `gs` 指向 `0B8h00h` 处，设置黑底白字。设置 `AL='L'`。在 `gs` 偏移处（屏幕第 0 行 39 列）写入 `L`。

## 5.3 调试过程及运行结果

调试方法与上一个实验类似，不作重复阐述。调试结果如下图 5-1 所示。

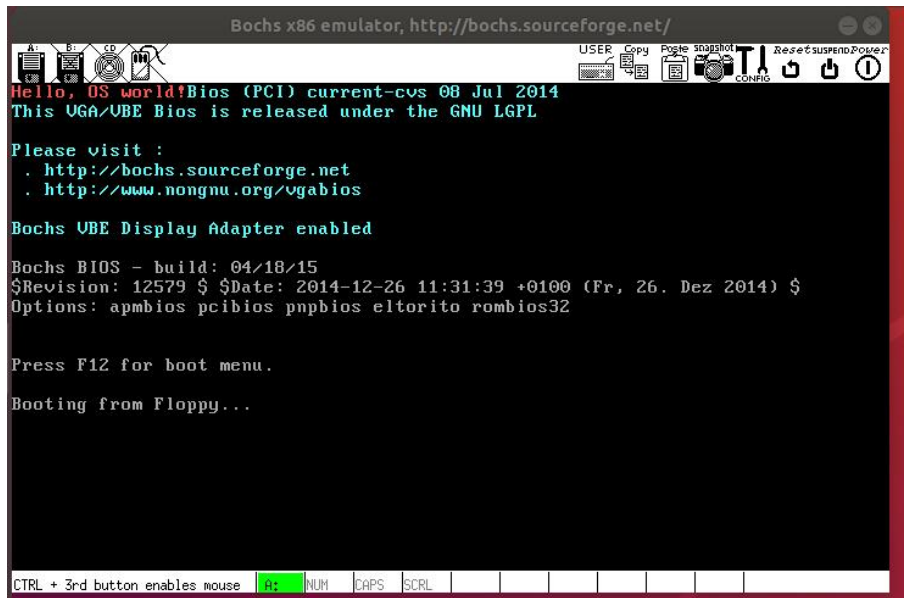


图 5.1 引导扇区被 DOS 识别

这时我们已经为我们的引导扇区加入了 BPB 等头信息，使其可以被 DOS 识别。接着我们根据实验要求修改 boot.asm，使其寻找 loader.bin 文件。由于我们仅仅是找到 Loader.bin 就停在那里，这时我们应该使用 Bochs 的调试功能。调试命令如下：

```
1  \b 0x7c00
2  \c
3  \n
4  \u /45
5  \b 0x7cb4
6  \c
7  \x /32xb es:di -16
8  \sreg
9  \r
```

调试过程如下图 5-2、5-3、5-4 所示：

```
<bochs:1> b 0x7c00
<bochs:2> c
00000004661i[BIOS ] $Revision: 12579 $ $Date: 2014-12-26 11:31:39 +0100 (Fr, 26. Dez 2014) $
00000318049i[KBD ] reset-disable command received
00000320818i[BIOS ] Starting rombios32
00000321256i[BIOS ] Shutdown flag 0
00000321839i[BIOS ] ram_size=0x02000000
00000322260i[BIOS ] ram_end=32MB
00000322770i[BIOS ] Found 1 cpu(s)
00000376974i[BIOS ] bios_table_addr: 0x000fa498 end=0x000fcc00
00000704769i[PCI ] i440FX PMC write to PAM register 59 (TLB Flush)
00001032698i[P2ISA] PCI IRQ routing: PIRQA# set to 0x0b
00001032717i[P2ISA] PCI IRQ routing: PIRQB# set to 0x09
00001032736i[P2ISA] PCI IRQ routing: PIRQC# set to 0x0b
00001032755i[P2ISA] PCI IRQ routing: PIRQD# set to 0x09
00001032765i[P2ISA] write: ELCR2 = 0x0a
00001033535i[BIOS ] PIIX3/PIIX4 init: elcr=00 0a
00001041216i[BIOS ] PCI: bus=0 devfn=0x00: vendor_id=0x8086 device_id=0x1237 class=0x0600
00001043495i[BIOS ] PCI: bus=0 devfn=0x08: vendor_id=0x8086 device_id=0x7000 class=0x0601
00001045613i[BIOS ] PCI: bus=0 devfn=0x09: vendor_id=0x8086 device_id=0x7010 class=0x0601
00001045838i[PIDE ] new BM-DMA address: 0xc000
00001046454i[BIOS ] region 4: 0x0000c000
00001048488i[BIOS ] PCI: bus=0 devfn=0x0b: vendor_id=0x8086 device_id=0x7113 class=0x0680
00001048720i[ACPI ] new irq line = 11
00001048732i[ACPI ] new irq line = 9
00001048757i[ACPI ] new PM base address: 0xb000
00001048771i[ACPI ] new SM base address: 0xb100
00001048799i[PCI ] setting SMRAM control register to 0x4a
00001212892i[CPU0 ] Enter to System Management Mode
00001212903i[CPU0 ] RSM: Resuming from System Management Mode
00001376924i[PCI ] setting SMRAM control register to 0x0a
00001391790i[BIOS ] MP table addr=0x000fa570 MPC table addr=0x000fa4a0 size=0xc8
00001393612i[BIOS ] SMBIOS table addr=0x000fa580
00001395793i[BIOS ] ACPI tables: RSDP addr=0x000fa6a0 ACPI DATA addr=0x01ff0000 size=0xf72
00001399005i[BIOS ] Firmware waking vector 0x1ff00cc
00001400800i[PCI ] i440FX PMC write to PAM register 59 (TLB Flush)
00001401523i[BIOS ] bios_table_cur_addr: 0x000fa6c4
00001529140i[VBIOS ] VGABios $Id: vgabios.c,v 1.76 2013/02/10 08:07:03 vruppert Exp $
00001529211i[BXVGA] VBE known Display Interface b0c0
00001529243i[BXVGA] VBE known Display Interface b0c5
00001532168i[VBIOS] VBE Bios $Id: vbe.c,v 1.65 2014/07/08 18:02:25 vruppert Exp $
00003795324i[XGUI ] charmap update. Font Height is 16
00014040196i[BIOS ] Booting from 0000:7c00
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=14040251
(0) [0x000000007c00] 0000:7c00 (unk. ctxt): jmp .+60 (0x00007c3e) ; eb3c
<bochs:3> n
Next at t=14040252
(0) [0x000000007c3e] 0000:7c3e (unk. ctxt): mov ax, cs ; 8cc8
```

图 5.2 调试过程 1



```

<bochs:5> b 0x7cad
<bochs:6> c
(0) Breakpoint 2, 0x00007cad in ?? ()
Next at t=14086043
(0) [0x000000007cad] 0000:7cad (unk. ctxt): jmp .-2 (0x00007cad) ; ebf
<bochs:7> 13. x /32xb es:di -16
:7: syntax error at '13'
<bochs:8> x /32xb es:di -16
[bochs]:
0x0009011b <bogus+ 0>: 0x00 0xff 0xff 0xff 0xff 0x4c 0x4f 0x41
0x00090123 <bogus+ 8>: 0x44 0x45 0x52 0x20 0x20 0x42 0x49 0x4e
0x0009012b <bogus+ 16>: 0x20 0x00 0x64 0x1a 0x29 0x77 0x3a 0x77
0x00090133 <bogus+ 24>: 0x3a 0x00 0x00 0x1a 0x29 0x77 0x3a 0x03
<bochs:9> x /13xcb es:di -11
:9: syntax error at '13'
<bochs:10> x /13xcb es:di -11
[bochs]:
0x00090120 <bogus+ 0>: L O A D E R
0x00090128 <bogus+ 8>: B I N \0
<bochs:11>

```

图 5.3 调试过程 2

```

<bochs:11> sreg
es:0x9000, dh=0x00009309, dl=0x0000ffff, valid=3
Data segment, base=0x00090000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
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=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x000fa1f7, limit=0x30
idtr:base=0x00000000, limit=0x3ff
<bochs:12> r
eax: 0x0000004e 78
ecx: 0x00090000 589824
edx: 0x0000000e 14
ebx: 0x00000100 256
esp: 0x00007c00 31744
ebp: 0x00000000 0
esi: 0x000e7cbf 949439
edi: 0x0000012b 299
eip: 0x00007cad
eflags 0x00000046: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
<bochs:13>

```

图 5.4 调试过程 3

最后，再次修改 boot.asm 使其将控制权交给 loader.bin。调试结果如下图 5-5 所示：





图 5.5 控制权交给 loader

## 5.4 实验总结

为突破 512 字节限制，我们进入保护模式的具体过程是：引导-> 加载内核入内存-> 跳入保护模式-> 开始执行内核，这样的工作如果全部交给引导扇区做，空间会不够，因此我们将其交给 Loader 模块完成，引导扇区负责把 Loader 载入内存并将控制权移交给它。我使用的软盘是 FAT12 格式。FAT12 格式分为若干个扇区，引导扇区是整个磁盘的第 0 个扇区，在这个扇区中有一个重要数据结构 BPB，说明 FAT 的内容，之后则依次是 FAT1、FAT2、根目录区及数据区。而引导扇区需要有 BPB 等头信息才能被识别，故将其添加在 boot.asm 开头。

在本实验中，我们实现了 boot.asm，使其可以从软盘中读出 Loader.bin 文件，并且加载入内核，同时移交控制权。只要一个.COM 文件中不含有 DOS 系统调用，我们就可以将它当成 Loader 使用，现在的程序已经被看做是一个在保护模式下执行的“操作系统”了。但是，我们目前的 Loader 仅仅只是一个 Loader，它不是操作系统内核，也不能当做操作系统内核，操作系统内核应该至少可以在 Linux 下用 GCC 编译链接，摆脱汇编语言，而 Loader 至少要实现两个功能：将内核加载入内存和跳入保护模式。

## 6 内核雏形

### 6.1 实验内容

进程切换；丰富中断处理程序，比如让时钟中断处理可以不停地发生而不是只发生一次，进程状态的保存与恢复，进程调度，解决中断重入问题。

### 6.2 代码分析

#### 6.2.1 核心数据结构

表 6.1 主要数据结构

|     |                  |           |
|-----|------------------|-----------|
| 结构体 | STACK_FRAME      | 进程表结构体的定义 |
|     | PROCESS          | 进程结构体的定义  |
| 宏   | NR_TASKS         | 最大允许进程：1  |
|     | STACK_SIZE_TOTAL | 栈的大小      |

#### 6.2.2 关键代码分析

1. [section .text] global \_start: 让 \_start 符号成为可见的标识符，这样链接器就知道跳转到程序中的什么地方并开始执行程序。\_start: 设置参数三：字符串长度设置参数二：要显示的字符串设置参数一：文件描述符 (stdout)
2. 系统调用号 (sys\_write): 调用内核功能进行写操作系统调用号 (sys\_exit): 调用内核功能进行退出操作
3. [section .text] 外部函数 extern choose \_start: 将 num2nd 和 num1st 推入栈中调用外部函数 choose 增加栈指针 8 个字节系统调用号 (sys\_exit): 调用内核功能进行退出操作

4. `myprint` 函数：设置参数二：`len`，设置参数一：`msg` 设置参数：文件描述符 (`stdout`)
5. 系统调用号 (`sys_write`)：调用内核功能进行写操作
6. 文件 `bar.c`：声明外部函数 `myprint()` 定义 `choose()` 函数若 `a >= b`：调用 `myprint` 输出“the 1st one” 否则：调用 `myprint` 输出“the 2nd one”
7. 文件 `loader.asm`
8. 使 `ds`、`es`、`ss` 三个段寄存器指向与 `cs` 相同的段，设置栈指针 `sp` 指向栈底
9. 清屏并显示字符串“Booting”
10. 软驱复位
11. 在 A 盘的根目录中查找 `LOADER.BIN`：遍历根目录，加载每个扇区到内存中，然后从中寻找文件名为 `Loader.bin` 的条目，直到找到为止。每读取一个扇区，在“Booting”后面打一个点。在成功加载和执行后显示字符串“Ready”。将控制权正式转移到已加载到内存中的 `LOADER.BIN` 的开头，开始执行其代码。
12. `DispStr` 函数：显示一个字符串
13. `ReadSector` 函数：从第 `ax` 个扇区开始，将 `cl` 个扇区读入 `es:bx` 中
14. `GetFATEntry` 函数：在 FAT 中找到序号为 `ax` 的扇区的条目，结果存储在 `ax` 中。需要注意的是，在过程中需要将 FAT 扇区读入 `es:bx`，因此函数开始时保存了 `es` 和 `bx`。
15. `KillMotor` 函数：关闭软驱马达
16. 文件 `Kernel.asm`：[`section .text`] `global _start`；导出 `_start` 跳转到 `_start`，假设 `gs` 指向视频内存。设置黑底白字。设置 `AL='K'`。在 `gs` 的偏移位置（屏幕第 1 行 39 列）写入字符‘L’。无限循环。
17. 文件 `loader.asm`：重新整理和对齐 `KERNEL.BIN` 的内容，将其放置在一个新位置。迭代每个程序头，根据程序头中的信息确定将什么内容放入内存中，放在何处，以及放置多少。

## 6.3 调试过程及结果分析

调试时主要出现以下问题，运行链接指令：`ld -s hello.o -o hello` 报错，查阅资料后得知该指令只适用于 32 位操作系统的环境，而我们现在使用的操作系统为

64 位。因此修改为：ld -m elf\_i386 -o hello hello.o，修改后成功运行，结果如下图 6-1 所示：

```
qj@ubuntu:~/Downloads/workspace/question5/1$ nasm -f elf hello.asm -o hello.o
qj@ubuntu:~/Downloads/workspace/question5/1$ ld -m elf_i386 -s -o hello hello.o
qj@ubuntu:~/Downloads/workspace/question5/1$ ./hello
Hello, world!
```

图 6.1 汇编语言 Hello World 运行结果

同样地，在编译时也需要指定 32 位的方式，使用以下指令：gcc -m32 -c -o bar.o bar.c。foobar 运行结果如下图 6-2 所示：

```
qj@ubuntu:~/Downloads/workspace/question5/2$ nasm -f elf -o foo.o foo.asm
qj@ubuntu:~/Downloads/workspace/question5/2$ gcc -m32 -c -o bar.o bar.c
qj@ubuntu:~/Downloads/workspace/question5/2$ ls -m elf_i386 -s -o foobar foo.o foo.bar
ls: cannot access 'elf_i386': No such file or directory
ls: cannot access 'foo.bar': No such file or directory
8 -rwxrwxr-x 1 qj 4536 Aug  6 07:05 foobar
4 -rw-rw-r-- 1 qj 720 Aug  9 10:49 foo.o
qj@ubuntu:~/Downloads/workspace/question5/2$ ld -m elf_i386 -s -o foobar foo.o foo.bar
ld: cannot find foo.bar: No such file or directory
qj@ubuntu:~/Downloads/workspace/question5/2$ ld -m elf_i386 -s -o foobar foo.o bar.o
qj@ubuntu:~/Downloads/workspace/question5/2$ ./foobar
the 2nd one
```

图 6.2 foobar 执行结果

在此处，定义了 num1=3，num2=4，程序输出大的那个数的结果，可以看到，程序输出了“the 2nd one”的字样，成功完成了任务。接着，我们以同样方式编译 boot，loader 和 kernel，调试运行结果如下图 6-3 所示：



图 6.3 载入内核

可以看到，在上一个实验的基础上，这次的实验结果多出了“Loading……”及“Ready.”这样的两行，说明我们已经载入了内核，并且由 Loader 读取了扇区。此外，还要将控制权交给内核，并重新进行调试，结果如下图 6-4 所示：

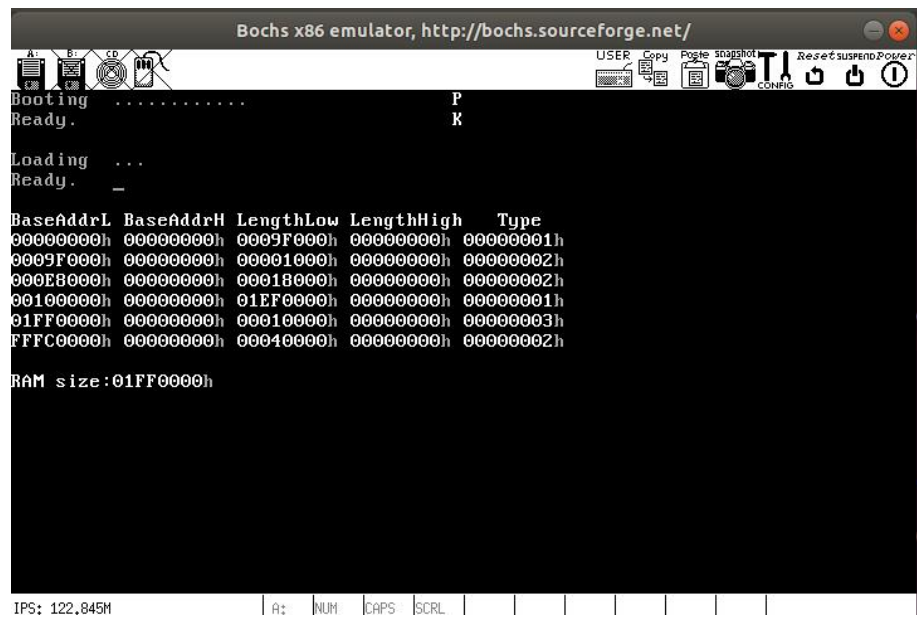


图 6.4 控制权交给内核

## 6.4 实验总结

在本实验中，我们的程序定义了两个节（Section），一个放数据，一个放代码。在代码中值得注意的一点是，入口点默认的是“\_start”，在定义它之后还要通过 global 这个关键字让连接程序找到它。代码本身则利用了两个系统调用。

实验要结合本机的环境进行，书上使用的是 32 位虚拟机，而现在的 Ubuntu 更多地是 64 位环境，因此需要针对这一点进行更改。

另一方面，本章节所实现的操作系统内核已经达到可以使用高级语言程序的地步了，这使得我们后面增加新功能的时候更加方便，可以更多地使用我们更习惯的 c 语言而不需要再去写底层的汇编语言。除此之外我们也为后面使用 Makefile 文件做好了铺垫，以便我们可以免去那些繁杂重复的工作。这个章节操作和实现上均较简单，但是是意义重大的承上启下的一步——我们将进入保护模式的复杂操作交给了内核，以后这些重复性工作将由内核替我们完成。

## 7 进程与进程调度

### 7.1 实验内容

进程切换；丰富中断处理程序，比如让时钟中断处理可以不停地发生而不是只发生一次，进程状态的保存与恢复，进程调度，解决中断重入问题。

### 7.2 代码分析

#### 7.2.1 核心数据结构

表 7.1 主要数据结构

|     |                  |           |
|-----|------------------|-----------|
| 结构体 | STACK_FRAME      | 进程表结构体的定义 |
|     | PROCESS          | 进程结构体的定义  |
| 宏   | NR_TASKS         | 最大允许进程    |
|     | STACK_SIZE_TOTAL | 栈的大小      |

#### 7.2.2 关键代码分析

1. 进程切换，实现从 Ring 0 到 Ring 1 的切换：

当发生中断时，从内核栈切换到进程栈，指向就绪队列中下一个应该被执行的进程，同时用 TSS 保存 Ring 0 的信息，如图 7-1 所示：

```
1      mov esp, [p_proc_ready]
2      lldt [esp + P_LDT_SEL]
3      lea eax, [esp + P_STACKTOP]
4      mov dword [tss + TSS3_S_SP0], eax
```

取出下一个进程所需的寄存器值，利用 `iretd` 转到 Ring 1 特权级，执行该进程，如下所示：

```

1      pop gs
2      pop fs
3      pop es
4      pop ds
5      popad
6      add esp, 4
7      iretd

```

2. 丰富中断处理程序，让时钟中断处理可以不停发生，并在进行进程调度的时候做到进程的保存与恢复。让中断可以不停发生，设置 EOI，如下所示：

```

1      hwint00:
2          mov al, EOI
3          out INT_M_CTL, al
4          iretd

```

为了让被中断的进程能够顺利恢复，在进程表中，我们给每一个寄存器预留了位置，以便让他们把所有的值都保存下来，这样在进程切换的时候就不会对进程产生不良影响。使用 `push` 保存原寄存器值，如下所示：

```

1      pushad
2      push ds
3      push es
4      push fs
5      push gs

```

使用 `pop` 恢复原寄存器值，如下所示：

```

1      pop gs
2      pop fs
3      pop es
4      pop ds
5      popad

```

为了保存内核栈信息，在从 Ring 0 到 Ring 1 的时候，我们使用 TSS 保存 Ring 0 的信息。在每一次重新开始前，使用 `iretd` 返回 Ring 1 之前，我们都要进行检查，以确保 `tss.esp0` 是正确的，如下所示：

```

1      lea eax, [esp + P_STACKTOP]
2      mov dword [tss + TSS3_S_SP0], eax

```

3. 解决中断重入问题：如果中断处理完成之前就会有下一个中断产生，我们会发现中断一直在进行，不会回到我们的程序。为解决中断重入问题，我们在 `main.c` 中设置全局变量 `k_reenter`。该全局变量初值为 -1，中断处理程序开始执行时它自加，结束时自减。在处理程序开头处，这个变量需要被检查一

下，如果值不是 0，则说明在一次中断未处理完之前就又发生了一次中断，这时直接跳到最后，结束中断处理程序的执行。然后判断是否需要跳转，如果 `k_reenter` 不为 0，则直接结束当前中断，恢复上一个进程的执行，如下所示：

```

1      k_reenter = -1;

1      inc dword [k_reenter]
2      cmp dword [k_reenter], 0
3      jne .re_enter

1      .re_enter:
2      dec dword [k_reenter]
3      pop gs
4      pop fs
5      pop ds
6      popad
7      add esp, 4
8      iretd

```

4. 进程调度：发生时钟中断时选择进程表中的下一个进程来执行，现在我们希望设定进程的优先级。为每个进程添加一个变量 `ticks`，初始值为进程的优先级。每次进程获得一个运行周期，`ticks` 减 1，减到 0 时，此进程不再获得执行机会，直到所有进程的 `ticks` 都减为 0。首先定义变量，如下所示：

```

1      proc_table[0].ticks = proc_table[0].priority = 150;
2      proc_table[1].ticks = proc_table[1].priority = 50;
3      proc_table[2].ticks = proc_table[2].priority = 30;

```

然后编写进程调度函数 `schedule()`，放在 `proc.c` 中，如下所示：

```

1      PUBLIC void schedule(){
2          PROCESS* p;
3          int greatest_ticks = 0;
4          while (!greatest_ticks){
5              for (p = proc_table; p < proc_table + NR_TASKS; p++){
6                  if (p->ticks > greatest_ticks){
7                      greatest_ticks = p->ticks;
8                      p_proc_ready = p;
9                  }
10             if (!greatest_ticks){
11                 for (p = proc_table; p < proc_table + NR_TASKS; p++){
12                     p->ticks = p->priority;
13                 }
14             }
15         }
16     }
17 }

```



同时修改时钟中断处理，如下所示：

```
1 PUBLIC void clock_handler(int irq){
2     ticks++;
3     p_proc_ready -> ticks++;
4     if (k_reenter != 0){
5         return;
6     }
7     schedule();
8 }
```

### 7.3 调试过程及结果分析

由于使用的是 64 位系统，与参考书版本不一致，需对 Makefile 作一定修改：

CFLAGS = -I include/ -c -fno-builtin LDFLAGS = -s -Ttext \$(ENTRYPOINT)

DASMFLAGS = -u -o \$(ENTRYPOINT) -e \$(ENTRYOFFSET)

改为：

CFLAGS = -m32 -I include/ -c -fno-builtin -fno-stack-protector LDFLAGS =  
-m elf\_i386 -s -Ttext \$(ENTRYPOINT)

DASMFLAGS = -u -o \$(ENTRYPOINT) -e \$(ENTRYOFFSET) -m32

即可使用 make 指令直接完成编译等工作，后续调试方法与之前实验一致，进程切换实验结果如下图 7-1 所示：

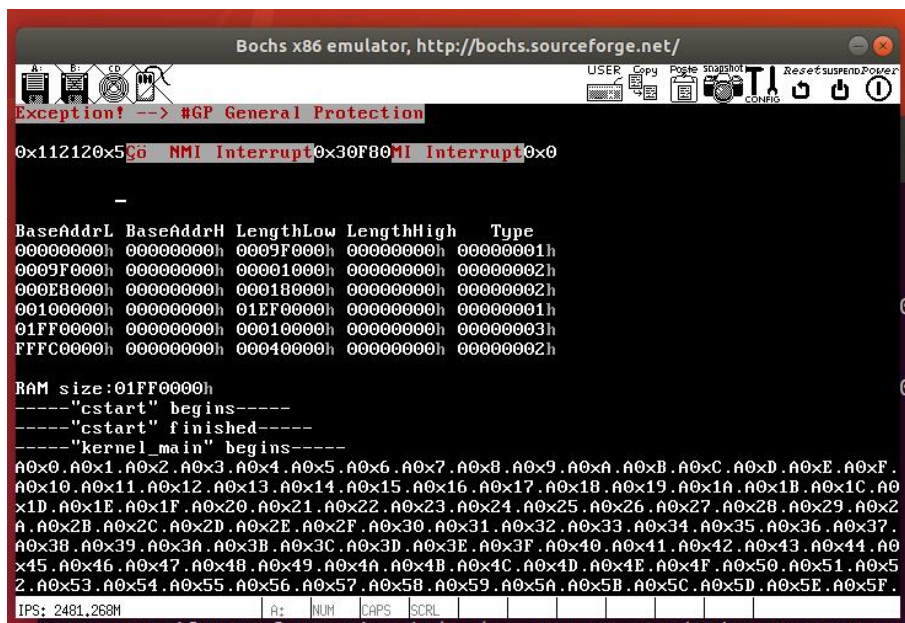


图 7.1 进程切换实验结果

接着丰富中断处理程序，让时钟中断处理可以不停地发生而不是只发生一次，如下图 7-2 所示：

```
Bochs x86 emulator, http://bochs.sourceforge.net/

USER Copy Paste Snapshot T Reset suspend Power
CONFIG

rooting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00001800h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
-----"cstart" begins-----
-----"cstart" finished-----
-----"kernel_main" begins-----
A0x0. ^A0x1. ^A0x2. ^A0x3. ^A0x4. ^^A0x5. ^A0x6. ^^A0x7. ^A0x8. ^A0x9. ^^A0xA. ^A0xB. ^A0xC.
^A0xD. ^A0xE. ^A0xF. ^^A0x10. ^A0x11. ^^A0x12. ^A0x13. ^A0x14. ^A0x15. ^A0x16. ^A0x17.
^A0x18. ^A0x19. ^^A0x1A. ^A0x1B. ^A0x1C. ^^A0x1D. ^A0x1E. ^A0x1F. ^^A0x20. ^A0x21. ^A0x22.
^^A0x23. ^A0x24. ^^A0x25. ^A0x26. ^A0x27. ^A0x28. ^A0x29. ^A0x2A. ^^A0x2B. ^A0x2C. ^^A0x2D.
^A0x2E. ^A0x2F. ^^A0x30. ^A0x31. ^A0x32. ^^A0x33. ^A0x34. ^A0x35. ^^A0x36. ^A0x37. ^^A0x38.
^A0x39. ^A0x3A. ^^A0x3B. ^A0x3C. ^A0x3D. ^^A0x3E. ^A0x3F. ^A0x40. ^^A0x41. ^A0x42. ^^A0x43.
^A0x44. ^A0x45. ^A0x46. ^A0x47. ^A0x48. ^^A0x49. ^A0x4A. ^^A0x4B. ^A0x4C. ^A0x4D. ^^A0x4E.
^A0x4F. ^^A0x50. ^A0x51. ^^A0x52. ^A0x53. ^A0x54. ^^A0x55. ^A0x56. ^A0x57. ^^A0x58. ^A0x59.
^^A0x5A. ^A0x5B. ^A0x5C. ^^A0x5D. ^A0x5E. ^A0x5F. ^^A0x60. ^A0x61. ^A0x62. ^^A0x63. ^A0x64.
^A0x65. ^A0x66. ^^A0x67. ^A0x68. ^A0x69. ^^A0x6A. ^A0x6B. ^A0x6C. ^^A0x6D. ^A0x6E. ^A0x6F.
^^A0x70. ^A0x71. ^A0x72. ^^A0x73. ^A0x74. ^A0x75. ^^A0x76. ^A0x77. ^A0x78. ^^A0x79. ^A0x7A.
^A0x7B. ^A0x7C. ^^A0x7D. ^A0x7E. ^A0x7F. ^^A0x80. ^A0x81. ^A0x82. ^^A0x83. ^A0x84. ^A0x85.
^^A0x86. ^A0x87. ^A0x88. ^^A0x89. ^A0x8A. ^A0x8B. ^^A0x8C. ^A0x8D. ^A0x8E. ^^A0x8F. ^A0x90.
^A0x91. ^A0x92. ^^A0x93. ^A0x94. ^A0x95. ^^A0x96. ^A0x97. ^A0x98. ^^A0x99. ^A0x9A. ^A0x9B.
^^A0x9C. ^A0x9D. ^A0x9E. ^^A0x9F. ^A0xA0. ^A0xA1. ^^A0xA2. ^A0xA3. ^A0xA4. ^^A0xA5. ^A0xA6.
^A0xA7. ^^A0xA8. ^A0xA9. ^A0xAA. ^^A0xAB. ^A0xAC. ^A0xAD. ^^A0xAE. ^A0xAF. ^^A0xB0. ^A0xB1.
^A0xB2. ^^A0xB3. ^A0xB4. ^A0xB5. ^^A0xB6. ^A0xB7. ^A0xB8. ^^A0xB9. ^A0xBA. ^A0xBB. ^^A0xBC.
^A0xBD. ^A0xBE. ^^A0xBF. ^A0xC0. ^A0xC1. ^^A0xC2. ^A0xC3. ^A0xC4. ^^A0xC5. ^A0xC6. ^A0xC7.
^^A0xC8. ^A0xC9. ^A0xCA. ^^A0xCB. ^A0xCC. ^A0xCD. ^^A0xCE. ^A0xCF. ^A0xD0. ^^A0xD1. ^A0xD2.
^A0xD3. ^^A0xD4. ^A0xD5. ^A0xD6. ^^A0xD7. ^A0xD8. ^A0xD9. ^^A0xDA. ^A0xDB. ^A0xDC. ^^A0xDD.
^A0xDE. ^A0xDF. ^^A0xE0. ^A0xE1. ^A0xE2. ^^A0xE3. ^A0xE4. ^A0xE5. ^^A0xE6. ^A0xE7. ^A0xE8.
^^A0xE9. ^A0xEA. ^A0xEB. ^^A0xEC. ^A0xED. ^A0xEE. ^^A0xEF. ^A0xF0. ^A0xF1. ^^A0xF2. ^A0xF3.
^A0xF4. ^^A0xF5. ^A0xF6. ^A0xF7. ^^A0xF8. ^A0xF9. ^A0xFA. ^^A0xFB. ^A0xFC. ^A0xFD. ^^A0xFE.
^A0xFF.

CTRL + 3rd button enables mouse
A: NUM CAPS SCRL
```

图 7.2 中断不停发生实验结果

再实现进程状态的保存与恢复，如下图 7-3 所示：

```
Bochs x86 emulator, http://bochs.sourceforge.net/

Booting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000EB000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01FF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
A0x0.^A0x1.^A0x2.^A0x3.^A0x4.^A0x5.^A0x6.^A0x7.^A0x8.^A0x9.^A0xA.^A0xB.^A0xC.^A0xD.^A0xE.^A0xF.^A0x10.^A0x11.^A0x12.^A0x13.^A0x14.^A0x15.^A0x16.^A0x17.^A0x18.^A0x19.^A0x1A.^A0x1B.^A0x1C.^A0x1D.^A0x1E.^A0x1F.^A0x20.^A0x21.^A0x22.^A0x23.^A0x24.^A0x25.^A0x26.^A0x27.^A0x28.^A0x29.^A0x2A.^A0x2B.^A0x2C.^A0x2D.^A0x2E.^A0x2F.^A0x30.^A0x31.^A0x32.^A0x33.^A0x34.^A0x35.^A0x36.^A0x37.^A0x38.^A0x39.^A0x3A.^A0x3B.^A0x3C.^A0x3D.^A0x3E.^A0x3F.^A0x40.^A0x41.^A0x42.^A0x43.^A0x44.^A0x45.^A0x46.^A0x47.^A0x48.^A0x49.^A0x4A.^A0x4B.^A0x4C.^A0x4D.^A0x4E.^A0x4F.^A0x50.^A0x51.^A0x52.^A0x53.^A0x54.^A0x55.^A0x56.^A0x57.^A0x58.^A0x59.^A0x5A.^A0x5B.^A0x5C.^A0x5D.^A0x5E.^A0x5F.^A0x60.^A0x61.^A0x62.^A0x63.^A0x64.^A0x65.^A0x66.^A0x67.^A0x68.^A0x69.^A0x6A.^A0x6B.^A0x6C.^A0x6D.^A0x6E.^A0x6F.^A0x70.^A0x71.^A0x72.^A0x73.^A0x74.^A0x75.^A0x76.^A0x77.^A0x78.^A0x79.^A0x7A.^A0x7B.^A0x7C.^A0x7D.^A0x7E.^A0x7F.^A0x80.^A0x81.^A0x82.^A0x83.^A0x84.^A0x85.^A0x86.^A0x87.^A0x88.^A0x89.^A0x8A.^A0x8B.^A0x8C.^A0x8D.^A0x8E.^A0x8F.^A0x90.^A0x91.^A0x92.^A0x93.^A0x94.^A0x95.^A0x96.^A0x97.^A0x98.^A0x99.^A0x9A.^A0x9B.^A0x9C.^A0x9D.^A0x9E.^A0x9F.^A0xA0.^A0xA1.^A0xA2.^A0xA3.^A0xA4.^A0xA5.^A0xA6.^A0xA7.^A0xA8.^A0xA9.^A0xAA.^A0xAB.^A0xAC.^A0xAD.^A0xAE.^A0xAF.^A0xB0.^A0xB1.^A0xB2.^A0xB3.^A0xB4.^A0xB5.^A0xB6.^A0xB7.^A0xB8.^A0xB9.^A0xBA.^A0xBB.^A0xBC.^A0xBD.^A0xBE.^A0xBF.^A0xC0.^A0xC1.^A0xC2.^A0xC3.^A0xC4.^A0xC5.^A0xC6.^A0xC7.^A0xC8.^A0xC9.^A0xCA.^A0xCB.^A0xCC.^A0xCD.^A0xCE.^A0xCF.^A0xD0.^A0xD1.^A0xD2.^A0xD3.^A0xD4.^A0xD5.^A0xD6.^A0xD7.^A0xD8.^A0xD9.^A0xDA.^A0xDB.^A0xDC.^A0xDD.^A0xDE.^A0xDF.^A0xE0.^A0xE1.^A0xE2.^A0xE3.^A0xE4.^A0xE5.^A0xE6.^A0xE7.^A0xE8.^A0xE9.^A0xEA.^A0xEB.^A0xEC.^A0xED.^A0xEE.^A0xEF.^A0xF0.^A0xF1.^A0xF2.^A0xF3.^A0xF4.^A0xF5.^A0xF6.^A0xF7.^A0xF8.^A0xF9.^A0xFA.^A0xFB.^A0xFC.^A0xFD.^A0xFE.^A0xFF.

IPS: 86.139M A: NUM CAPS SCRL
```

图 7.3 中断不停发生实验结果

然后支持中断重入，如下图 7-4 所示：

```

Bochs x86 emulator, http://bochs.sourceforge.net/
tooting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01FF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
A0x0. #A0x1. ##A0x2. #A0x3. #A0x4. ##A0x5. #A0x6. ##A0x7. #A0x8. #A0x9. ##A0xA. #A0xB. #A0xC
. ##A0xD. #A0xE. ##A0xF. #A0x10. #A0x11. ##A0x12. #A0x13. #A0x14. ##A0x15. #A0x16. #A0x17. #
#A0x18. #A0x19. ##A0x1A. #A0x1B. #A0x1C. ##A0x1D. #A0x1E. #A0x1F. ##A0x20. #A0x21. #A0x22.
. ##A0x23. #A0x24. ##A0x25. #A0x26. #A0x27. ##A0x28. #A0x29. #A0x2A. ##A0x2B. #A0x2C. ##A0x2
D. #A0x2E. #A0x2F. ##A0x30. #A0x31. #A0x32. ##A0x33. #A0x34. #A0x35. ##A0x36. #A0x37. ##A0x
38. #A0x39. #A0x3A. ##A0x3B. #A0x3C. #A0x3D. ##A0x3E. #A0x3F. ##A0x40. #A0x41. #A0x42. ##A0
x43. #A0x44. #A0x45. ##A0x46. #A0x47. #A0x48. ##A0x49. #A0x4A. ##A0x4B. #A0x4C. #A0x4D. #A0
CTRL + 3rd button enables mouse A: NUM CAPS SCRL

```

图 7.4 中断重入实验结果

最后执行进程调度，如下图 7-5 所示：

```

Bochs x86 emulator, http://bochs.sourceforge.net/
tooting .....
Ready.

Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01FF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
A0x0. #B0x1000. #A0x1. ##B0x1001. ##B0x1002. #A0x2. ##A0x3. #B0x1003. #A0x4. #B0x1004. ##B0
x1005. #A0x5. ##A0x6. #B0x1006. ##B0x1007. #A0x7. #B0x1008. #A0x8. ##A0x9. #B0x1009. ##B0x
100A. #A0xA. ##A0xB. #B0x100B. #A0xC. #B0x100C. ##B0x100D. #A0xD. ##A0xE. #B0x100E. ##B0x1
00F. #A0xF. #B0x1010. #A0x10. ##A0x11. #B0x1011. ##B0x1012. #A0x12. #B0x1013. #A0x13. ##A0
x14. #B0x1014. ##B0x1015. #A0x15. ##A0x16. #B0x1016. #A0x17. #B0x1017. ##B0x1018. #A0x18.
. ##A0x19. #B0x1019. ##B0x101A. #A0x1A. #B0x101B. #A0x1B. ##A0x1C. #B0x101C. ##B0x101D. #A0
x1D. ##A0x1E. #B0x101E. #A0x1F. ##B0x101F. ##B0x1020. #A0x20. ##A0x21. #B0x1021. #A0x22. #B
IPS: 87,801M A: NUM CAPS SCRL

```

图 7.5 进程调度实验结果

## 7.4 实验总结

本实验将重点放到了进程上。从进程切换，中断处理程序的丰富，再到进程切换时进程状态的保存与恢复，最后到进程调度，同时考虑并解决了中断重入问题。虽然还没办法进行输入输出，但已经可以说这个操作系统已经是一个基本完善的操作系统了。

本次实验使用了 Makefile 来化简我们的操作步骤以及展现不一样的内核效果。操作途中遇到了 Makefile 需要更改的问题，因为里面的内容是以 32 位虚拟机为对象来做的，而我们使用的是 64 位虚拟机，将对应代码更改完后直接运行 make 指令，后续实验步骤和之前相同。

总而言之，第六个实验的任务是实现进程和进程调度。首先准备好进程体，初始化相应的描述符，准备进程表，完成特权级别的跳转，从 ring0 到 ring1。接下来要完善时钟中断处理程序，为此我们需要设置 EOI，运行可以看到 0 行 0 列的字符不断变化，说明中断处理程序正在运行。然后是现场的保护和恢复，我们需要用进程表保存进程的状态，具体需要保存各寄存器的值，切内核栈和重新将 esp 切到进程表。然后是中断重入，如果中断未处理完之前又发生中断，直接结束中断处理程序的运行。实现多进程时，我们读取不同的任务地址入口、堆栈栈顶和进程名，在进程切换时重新加载 ldt。进程切换时，为 esp 赋不同的值。我们只打开时钟中断的时候，屏蔽掉时间中断，并进行相关系统调用。在进程调度中，我们设置不同的延迟，进行了简单的优先级设置。整理的实验六的思维导图如下：



图 7.6 实验六思维导图

## 8 操作系统的输入/输入系统

### 8.1 实验内容

实现简单的 I/O，从键盘输入字符的中断开始；获取并打印扫描码；创建对应打印扫描码解析数组，打印对应字符。

### 8.2 代码分析

#### 8.2.1 核心数据结构

表 8.1 keyboard.h 缓冲区 KB\_INPUT

|              |                  |               |
|--------------|------------------|---------------|
| 缓冲区 KB_INPUT | p_head           | 指向缓冲区中下一个空闲位置 |
|              | p_tail           | 指向键盘任务应处理的字节  |
|              | Count            | 缓冲区中字节数       |
|              | buf[KB_IN_BYTES] | 缓冲区           |

#### 8.2.2 关键代码分析

1. 键盘中断处理程序:

```
1     PUBLIC void keyboard_handler(int irq)
2     {
3         disp_str("str");
4     }
```

2. 打开键盘中断:

```
1     PUBLIC void init_keyboard()
2     {
3         put_irq_handler(KEYBOARD_IRQ, keyboard_handler); /* 设定键
                    盘中断处理程序 */
```



```

4     enable_irq(KEYBOARD_IRQ); /* 开启键盘中断 */
5 }

```

### 3. 调用 init\_keyboard:

```

1     PUBLIC int kernel_main()
2     {
3         // ...
4         init_keyboard();
5         // ...
6     }

```

### 4. 修改后的键盘中断:

```

1     PUBLIC void keyboard_handler(int irq)
2     {
3         /* disp_str("*"); */
4         u8 scan_code = in_byte(0x60);
5         disp_int(scan_code);
6     }

```

### 5. 扫描码解析数组:

```

1     u32 keymap[NR_SCAN_CODES * MAP_COLS] = {
2         /* 0x00 - none */ 0, 0, 0,
3         /* 0x01 - ESC */ ESC, ESC, 0,
4         /* 0x02 - '1' */ '1', '!', 0,
5         // ...
6     };

```

### 6. 键盘缓冲区:

```

1     typedef struct s_kb {
2         char* p_head; /* 指向缓冲区中下一个空闲位置 */
3         char* p_tail; /* 指向键盘任务应处理的字节 */
4         int count;    /* 缓冲区中共有多少字节 */
5         char buf[KB_IN_BYTES]; /* 缓冲区 */
6     } KB_INPUT;

```

### 7. 修改后的 keyboard\_handler:

```

1     PRIVATE KB_INPUT kb_in;
2
3     /* keyboard_handler */
4     PUBLIC void keyboard_handler(int irq)
5     {
6         u8 scan_code = in_byte(KB_DATA);
7         if (kb_in.count < KB_IN_BYTES) {
8             *(kb_in.p_head) = scan_code;
9             kb_in.p_head++;
10            if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {

```

```

11         kb_in.p_head = kb_in.buf;
12     }
13     kb_in.count++;
14 }
15 }

```

#### 8. 修改后的 init\_keyboard:

```

1     PUBLIC void init_keyboard()
2     {
3         kb_in.count = 0;
4         kb_in.p_head = kb_in.p_tail = kb_in.buf;
5         put_irq_handler(KEYBOARD_IRQ, keyboard_handler); /* 设定键
           盘中断处理程序 */
6         enable_irq(KEYBOARD_IRQ); /* 开键盘中断 */
7     }

```

#### 9. 时钟中断处理函数 init\_clock:

```

1     PUBLIC void init_clock()
2     {
3         /* 初始化 8253 PIT */
4         out_byte(TIMER_MODE, RATE_GENERATOR);
5         out_byte(TIMER0, (u8) (TIMER_FREQ/HZ));
6         out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));
7         put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断
           处理程序 */
8         enable_irq(CLOCK_IRQ); /* 让 8259A 可以接收时钟中断 */
9     }

```

#### 10. 键盘读取的 tty 任务:

```

1     PUBLIC void task_tty()
2     {
3         while (1) {
4             keyboard_read();
5         }
6     }

```

#### 11. 键盘读取函数 keyboard\_read():

```

1     PUBLIC void keyboard_read()
2     {
3         u8 scan_code;
4         if (kb_in.count > 0) {
5             disable_int();
6             scan_code = *(kb_in.p_tail);
7             kb_in.p_tail++;
8             if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
9                 kb_in.p_tail = kb_in.buf;

```

```

10         }
11         kb_in.count--;
12         enable_int();
13         disp_int(scan_code);
14     }
15 }

```

## 12. disable\_int 与 enable\_int:

```

1  /* void disable_int(); */
2  /* disable_int: */
3  /* cli */
4  /* ret */
5
6  /* void enable_int(); */
7  /* enable_int: */
8  /* sti */
9  /* ret */

```

## 13. 解析扫描码:

```

1  PUBLIC void keyboard_read()
2  {
3      u8 scan_code;
4      char output[2];
5      int make; /* TRUE: make; FALSE: break. */
6
7      memset(output, 0, 2);
8      if (kb_in.count > 0) {
9          disable_int();
10         scan_code = *(kb_in.p_tail);
11         kb_in.p_tail++;
12         if (kb_in.p_tail == kb_in.buf + KB_IN_BYTES) {
13             kb_in.p_tail = kb_in.buf;
14         }
15         kb_in.count--;
16         enable_int();
17         /* 下面开始解析扫描码 */
18         if (scan_code == 0xE1) {
19             /* 暂时不做任何操作 */
20         }
21         else if (scan_code == 0xE0) {
22             /* 暂时不做任何操作 */
23         }
24         else {
25             /* 下面处理可打印字符 */
26             /* 首先判断 Make Code 还是 Break Code */
27             make = (scan_code & FLAG_BREAK ? FALSE : TRUE);

```



```

28      /* 如果是 Make Code 就打印, 是 Break Code 则不做处
        理 */
29      if (make) {
30          output[0] = keymap[(scan_code & 0x7F) *
                               MAP_COLS];
31          disp_str(output);
32      }
33  }
34  }
35  }

```

### 8.3 调试过程及结果分析

使用 make 指令编译, bochs 指令运行, 可以看到, 当我们敲击键盘之后, 程序中出现了一个 “\*” 的字符, 但当我们再次敲击键盘后, 键盘却不会再次相应, 即屏幕上不会出现第二个 “\*” 字符, 而且在 terminal 中可以看到栈溢出的报错信息。结果如下图 8-1 所示:

```

Bochs x86 emulator, http://bochs.sourceforge.net/
Booting .....
Ready.
Loading .....
Ready.

BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
0009F000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h

RAM size:01FF0000h
----"cstart" begins----
----"cstart" finished----
----"kernel_main" begins----
*

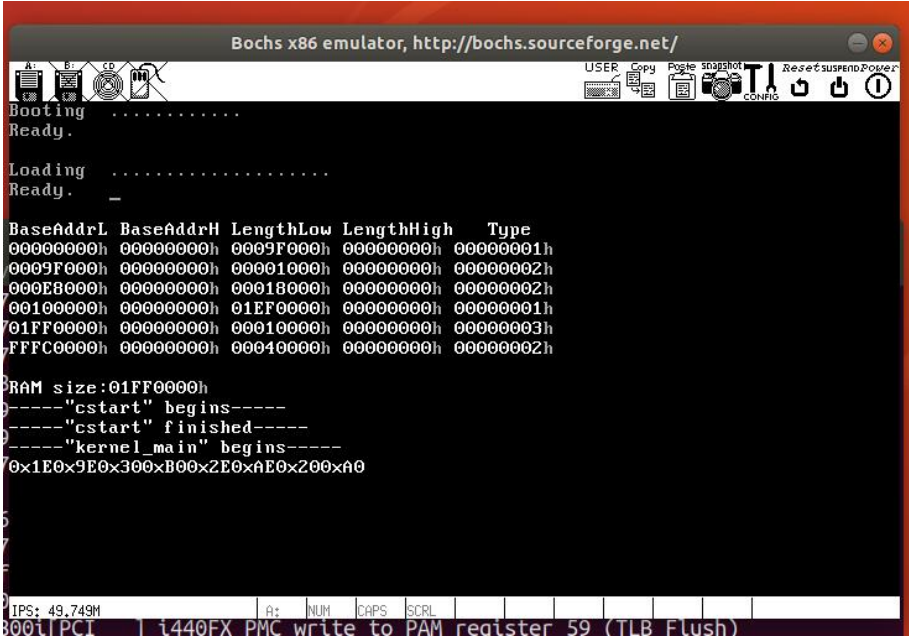
size=0xc
01ff0000

IPS: 51.146M  At: NUM CAPS SCRL
00001399002i[BIOS ] Firmware waking vector 0x1ff00cc
00001400797i[PCI ] i440FX PMC write to PAM register 59 (TLB Flush)
00001401520i[BIOS ] bios_table_cur_addr: 0x000fa6c4
00001529137i[VBIOS ] VGABios $Id: vgabios.c,v 1.76 2013/02/10 08:07:03 vruppert
Exp $
00001529208i[BXVGA ] VBE known Display Interface b0c0
00001529240i[BXVGA ] VBE known Display Interface b0c5
00001532165i[VBIOS ] VBE Bios $Id: vbe.c,v 1.65 2014/07/08 18:02:25 vruppert Exp
$
00004609976i[XGUI ] charmap update. Font Height is 16
00014040193i[BIOS ] Booting from 0000:7c00
00631424000i[KBD ] internal keyboard buffer full, ignoring scancode.(9e)

```

图 8.1 键盘中断处理程序打印星号

在此处，我们调用键盘中断，在敲击键盘后出现了一个“\*”字符，代表我们的键盘中断是准确无误的，但它目前只能相应一次键盘敲击。结合栈溢出的报错信息，我们可以每次输出缓冲区中字符对应的解析码。运行结果如下图 8-2 所示：



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Booting ....., Ready.
Loading ....., Ready.
BaseAddrL BaseAddrH LengthLow LengthHigh Type
00000000h 00000000h 0009F000h 00000000h 00000001h
0009F000h 00000000h 00001000h 00000000h 00000002h
000E8000h 00000000h 00018000h 00000000h 00000002h
00100000h 00000000h 01EF0000h 00000000h 00000001h
01FF0000h 00000000h 00010000h 00000000h 00000003h
FFFC0000h 00000000h 00040000h 00000000h 00000002h
RAM size: 01FF0000h
cstart begins
cstart finished
kernel_main begins
0x1E0x9E0x300x800x2E0xA0x200xA0
IPS: 49.749M
A: NUM CAPS SCRL
00011PCI i440FX PMC write to PAM register 59 (TLB Flush)
```

图 8.2 读取缓冲区并打印解析码

在上图所示调试过程中，我们连续连续敲击了四个键：字符“a”、字符“b”、字符“c”和字符“d”，而实验结果则一共出现了 8 组数字，分别对应于字符“a”、“b”、“c”和“d”的 Make Code 和 Break Code。可以看到，我们目前的程序已经不会再卡死，而是可以相应多次键盘敲击过程，并且打印出扫描码。对于这一点，则涉及到键盘控制器 8042 芯片和键盘编码器 8048 芯片，它们会监视键盘的输入，并把适当的数据传给计算机。而对于敲击键盘的动作，则会产生扫描码，分别为按下一个按键或保持一个按键的 Make Code 和键盘弹起时的 Break Code，这些扫描码可以通过 in al,60h 读取，并且只有我们将扫描码从缓冲区中读出来后，8042 才能继续响应新的按键，这也解释了我们在之前的实验中只打印一个“\*”的原因。接下来我们通过扫描码解析数组 keymap[]，建立字符与解析码之间的映射关系，再次调试结果如下图 8-3 所示：

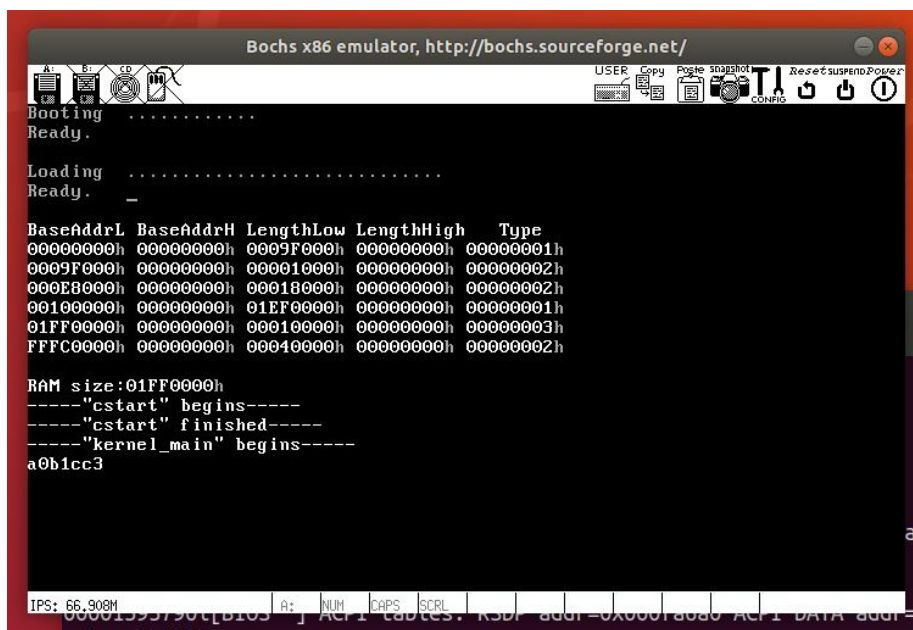


图 8.3 输出正确的小写字母和数字

可以看到，目前我们的程序已经可以对小写 a-z 以及 0-9 产生正常响应，但仍然无法打印出大写字母，对 shift、alt、ctrl 等按键则会输出意义不明的字符。由于我们已经知晓键盘敲击产生字符的原理，根据组合键的 Make Code 和 Break Code 专门处理，这一点不难实现。实现后运行结果如下图 8-4 所示：

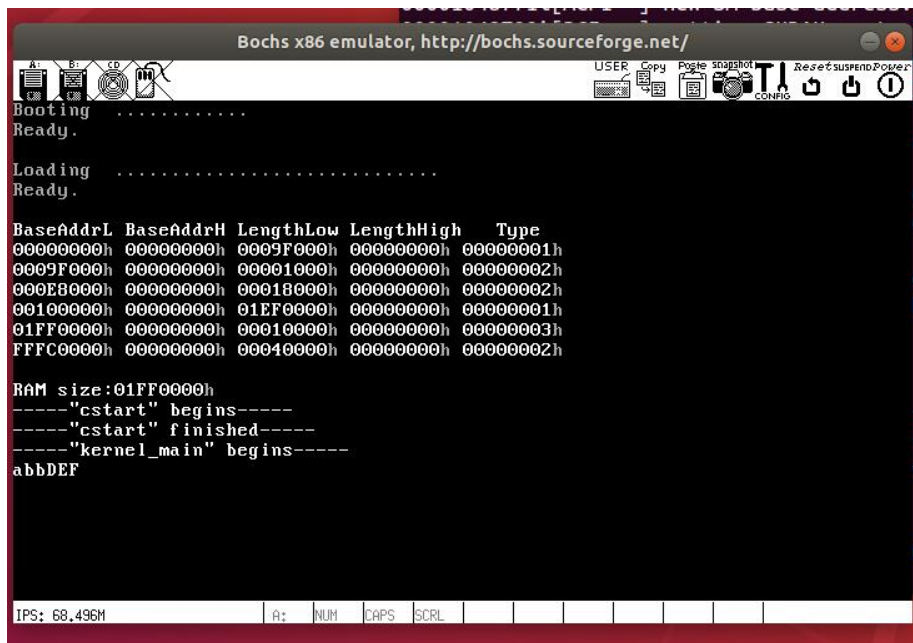


图 8.4 输出组合键

## 8.4 实验总结

本次实验研究的是输入与输出。经过这次实验，我对电脑如何将外界输入处理成对应显示有了更深的了解。一次按键对应一个输出字符很好理解，但对于各种复杂组合键的实现方法，了解到敲击键盘按下和松开分别会有两个扫描码后，这才从原理上完全理解。

总而言之，第七个实验的任务是实现操作系统的输入输出系统。最初的时候，我们的键盘只能输入一个星号，因为这个程序并没有从缓冲区读取扫描码，相应的芯片不能及时响应按键。然后我们添加 `in_byte(0x60)` 指令，这样就可以多次输入了。接着，我们再次修改代码，使得屏幕能够显示按键的扫描码 `MakeCode` 和 `BreakCode`。接下来，我们引入扫描码解析数组，再使用键盘缓冲区存放暂未处理的输入。我们首先忽略 `0xE0` 和 `0xE1`。这样所有的小写字母就可以正常输出了。然后增加处理 `Shift` 的代码，这样就能够识别大写字母和特殊字符了。整理的实验七实验思维导图如下图 8-5 所示：



图 8.5 实验七思维导图

## 9 操作系统进阶

### 9.1 实验内容

1. 自定义一个系统调用，能够统计一个进程在运行的过程中被调度的次数。编写一个简单的用户程序，调用该自定义的系统调用，从而将进程及其调度的次数输出在屏幕上。
2. 在实现了三个进程的优先级调度的基础上，将三个进程的循环次数从无限循环修改为有限次数，当三个进程执行完成时，计算三个进程在优先级调度算法下的周转时间、等待时间以及该系统的平均周转时间、平均等待时间和吞吐量。也可以添加新的进程，然后计算该系统的平均周转时间、平均等待时间和吞吐量。

### 9.2 代码分析

#### 9.2.1 系统调用部分

1. 系统调用过程：（以书中 `write` 系统调用为例），如下图 9-1 所示：

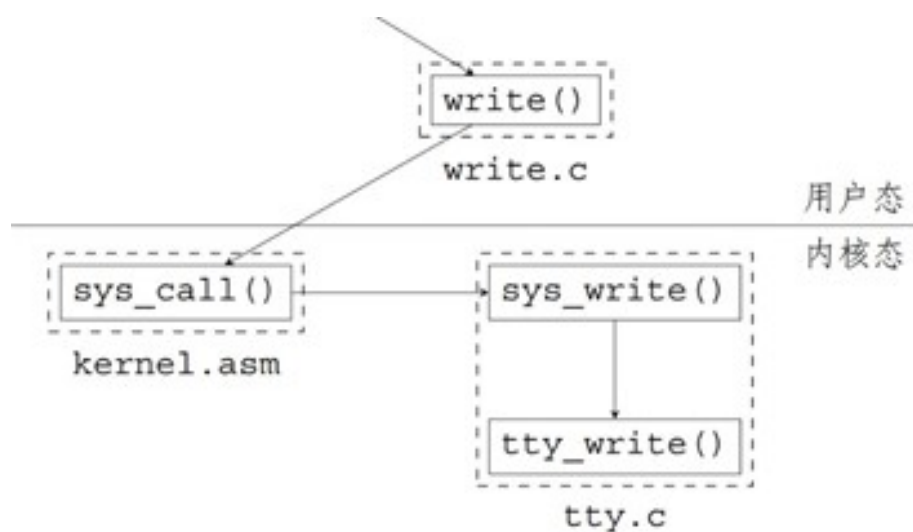


图 9.1 系统调用流程图

2. 实现一个新的系统调用（这里为 `get_foo`）主要需要完成两部分：`get_foo` 函数本身及其内核部分 `sys_get_foo`，如图 9-2 所示：

| 步骤 | 内容  | 文件                       |
|----|---|--------------------------|
| 1  | <code>NR_SYS_CALL</code> 加一                                     | <code>const.h</code>     |
| 2  | 为 <code>sys_call_table[]</code> 增加一个成员，假设是 <code>sys_foo</code> | <code>global.c</code>    |
| 3  | <code>sys_foo</code> 的函数体                                       | 因具体情况而异                  |
| 4  | <code>sys_foo</code> 的函数声明                                      | <code>proto.h</code>     |
| 5  | <code>foo</code> 的函数声明  | <code>proto.h</code>     |
| 6  | <code>_NR_foo</code> 的定义  | <code>syscall.asm</code> |
| 7  | <code>foo</code> 的函数体   | <code>syscall.asm</code> |
| 8  | 添加 <code>global foo</code>                                      | <code>syscall.asm</code> |
| 9  | 如果参数个数与以前的系统调用比有所增加，则需要修改 <code>sys_call</code>                 | <code>kernel.asm</code>  |

图 9.2 系统调用函数

3. 添加系统调用的步骤：

- (a) 将 `const.h` 文件中的宏 `NR_SYS_CALL` 加 1。

`NR_SYS_CALL` 表示的是操作系统中定义的系统调用的个数，会用到其值来定义系统调用数组。

- (b) 在 `global.c` 文件为 `sys_call_table[]` 增加一个成员，命名为 `sys_get_foo`，此处模仿前两个系统调用名并结合功能设为 `sys_get_num`。此处用到了步骤一中设置的 `NR_SYS_CALL` 来定义数组。

- (c) 完成 `get_foo` 的声明及 `sys_get_foo` 的声明和函数体。声明位于头文件 `proto.h` 中，函数体的位置较灵活，此处选择将新增的系统调用放在第一个系统调用 `sys_get_ticks` 后面，即 `proc.c` 文件最下方。

- (d) 定义 `_NR_get_foo` 及添加 `global get_foo`。

`_NR_get_time` equ 后的值设为 1，和已有系统调用的调用号不一致即可。

- (e) 设计用户程序：在 `main.c` 文件中编写了测试进程：`TestA`、`TestB` 与 `TestC`，如图所示。其中，`TestA` 调用了 `get_foo` 与 `get_ticks` 系统调用函数，进程 B 与进程 C 没有调用系统函数，目的为与 `TestA` 形成对照，使实验结果清晰，如下所示：



```

1  void TestA(){
2      for(int k = 0;k < 10;k++){
3          disp_str("A");
4          disp_int(get_foo() + 1);
5          disp_int(get_ticks());
6          milli_delay(100);
7      }
8  }

```

```

1  void TestB(){
2      for(int i = 0;i < 10;i++){
3          disp_str("B");
4          milli_delay(100);
5      }
6  }

```

```

1  void TestC(){
2      for(int i = 0;i < 10;i++){
3          disp_str("C");
4          milli_delay(100);
5      }
6  }

```

### 9.2.2 优先级调度部分

1. 此处添加用户进程的主要步骤与系统调用部分完全一致，不重复阐述。
2. 在 main.c 文件中给三个进程添加 ticks 属性，定义为有限值，每次调用 ticks 值减一，为 0 后则不再调用。并对 proc.c 文件中的 schedule 函数进行修改，删掉 ticks 归零后重新赋值的部分，从而将三个进程的循环次数从无限循环修改为有限次数。如下所示：

```

1  proc_table[0].ticks = 100;
2  proc_table[1].ticks = 40;
3  proc_table[2].ticks = 30;

```

```

1  /*if (!greatest_ticks){
2      for (p = proc_table; p < proc_table + NR_TASKS; p++){
3          p->ticks = p->priority;
4      }
5  }*/

```

3. 在进程结构体中添加相应成员。周转时间命名为 TurnaroundTick，等待时间命名为 WaitingTick。此处，我对优先级调度算法进行了修改，使进程调度完全按照自行设定的优先级顺序（priority 值的大小）来执行，保证高优先级

进程一定在低优先级进程之前执行完毕。因此，周转时间直接读取进程结束时的 ticks 中，而等待时间可直接读取进程开始时的 ticks。调度算法如下：

```
1  PROCESS*   p;
2  int        greatest_priority = 0;
3  for (p=proc_table; p<proc_table+NR_TASKS; p++) {
4      if (p->priority > greatest_priority && p->ticks >
5          0) {
6          greatest_priority = p->priority;
7          p_proc_ready = p;
8      }
9  }
```

4. 三个进程结束后计算并打印出相应的数据，需要注意的是单位为 ms，要进行转换。如下所示：

```
1  if (greatest_priority == 0) {
2      // turnaround time
3      disp_str("\n");
4      disp_str("turnaround time of a:");
5      disp_int(proc_table[0].TurnaroundTick / 10);
6      disp_str("ms\n");
7
8      disp_str("turnaround time of b:");
9      disp_int(proc_table[1].TurnaroundTick / 10);
10     disp_str("ms\n");
11
12     disp_str("turnaround time of c:");
13     disp_int(proc_table[2].TurnaroundTick / 10);
14     disp_str("ms\n");
15
16     // waiting time
17     disp_str("waiting time of a:");
18     disp_int(proc_table[0].WaitingTick / 10);
19     disp_str("ms\n");
20
21     disp_str("waiting time of b:");
22     disp_int(proc_table[1].WaitingTick / 10);
23     disp_str("ms\n");
24
25     disp_str("waiting time of c:");
26     disp_int(proc_table[2].WaitingTick / 10);
27     disp_str("ms\n");
28
29     // average turnaround time
30     disp_str("average turnaround time:");
31     disp_int((proc_table[0].TurnaroundTick + proc_table[1].
32         TurnaroundTick + proc_table[2].TurnaroundTick / 30) / 30);
33 }
```



```

32     disp_str("ms\n");
33     disp_str("average waiting time:");
34     disp_int((proc_table[0].WaitingTick + proc_table[1].
        WaitingTick + proc_table[2].WaitingTick) / 30);
35     disp_str("ms\n");
36
37     // throughput
38     disp_str("throughput:");
39     int timeOfThree = proc_table[0].TurnaroundTick > proc_table
        [1].TurnaroundTick ? proc_table[0].TurnaroundTick :
        proc_table[1].TurnaroundTick;
40     timeOfThree = timeOfThree > proc_table[2].TurnaroundTick ?
        timeOfThree : proc_table[2].TurnaroundTick;
41     disp_int((proc_table[0].ticks + proc_table[1].ticks +
        proc_table[2].ticks) / timeOfThree);
42     disp_str("/s\n");
43 }

```

### 9.3 调试过程及结果分析

沿用之前的 Makefile，以同样指令编译运行。系统调用部分运行结果如下图 9-3 所示：

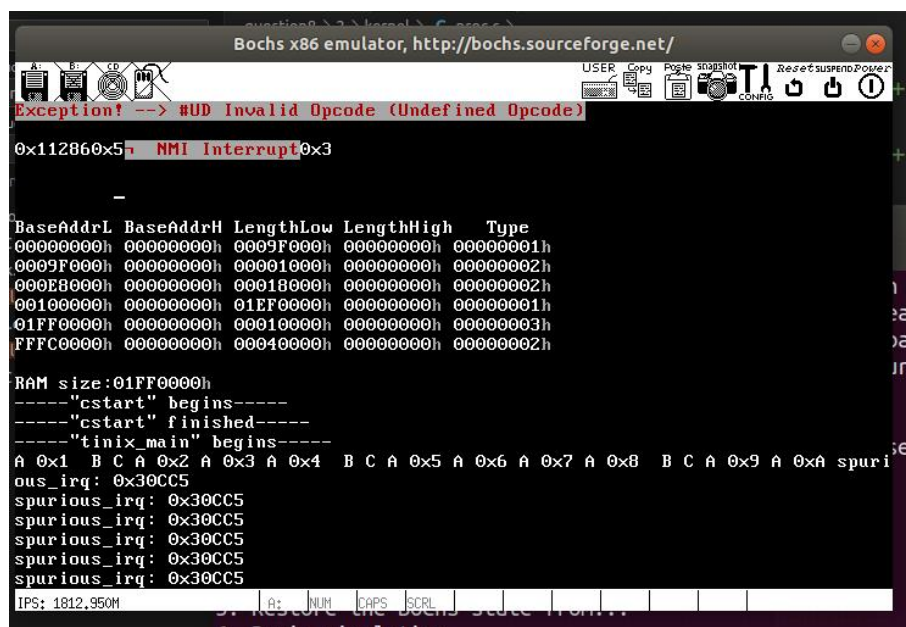
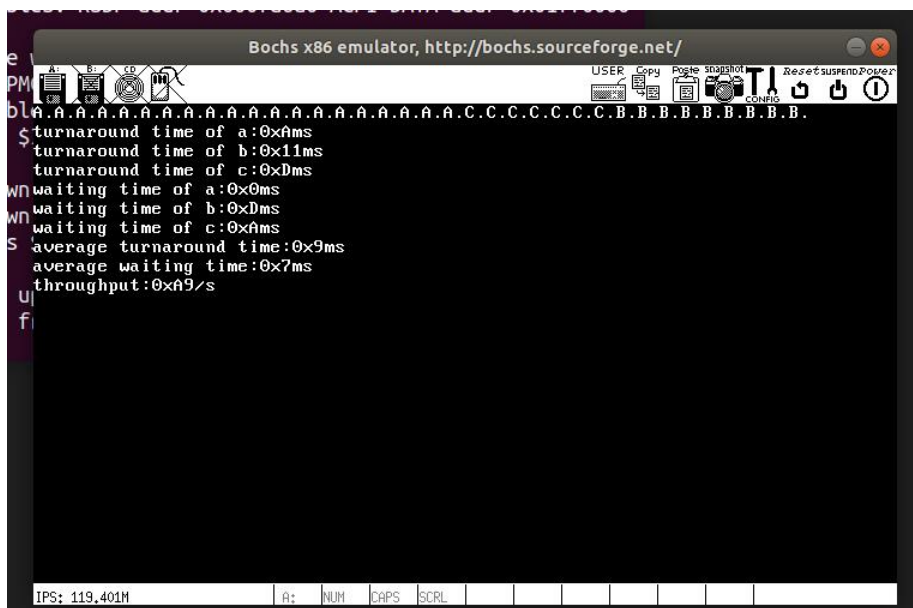


图 9.3 系统调用运行结果

优先级调度部分运行结果如下图 9-4 所示：



```
Bochs x86 emulator, http://bochs.sourceforge.net/
USER Copy Paste Snapshot Reset suspend power
A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.A.C.C.C.C.C.C.C.C.B.B.B.B.B.B.B.B.B.B.
$ turnaround time of a:0x0ms
$ turnaround time of b:0x11ms
$ turnaround time of c:0x0ms
waiting time of a:0x0ms
waiting time of b:0x0ms
waiting time of c:0x0ms
average turnaround time:0x9ms
average waiting time:0x7ms
throughput:0x9/s
IPS: 119,401M A: NUM CAPS SCRL
```

图 9.4 优先级调度运行结果

## 9.4 实验总结

最后的操作系统进阶实验中，首先在内核中添加了一个新的系统调用，该系统调用功能是统计一个进程在执行过程中被调度的次数，然后在用户程序中调用该系统调用。在实现了自定义的系统调用的基础上，我设计了一个简单的进程优先级调度算法。原先的三个进程是无限循环执行的，我将它们的循环次数修改为有限次数。三个进程执行完成后，我计算了它们在优先级调度算法下的周转时间、等待时间以及整个系统的平均周转时间、平均等待时间和吞吐量。通过这次实验，我学会了如何在操作系统中添加自定义的系统调用，熟悉了整个系统调用的流程，并且了解了系统调用的原理和使用方法，同时更深入地理解了进程调度算法的原理和实际应用。

## 10 实验心得体会

本次操作系统复现实验中，我参考《一个操作系统的实现》[4] 这本指导书，将理论知识与实践相结合，复现了一个基本的操作系统。从搭建虚拟机工作环境开始，一步步地实现保护模式和实模式的转换，切换到保护模式之后实加载 Loader 进入内存并将控制权交给 loader，随后将 loader 替换成 kernel 内核，实现进程切换、保存与恢复、调度，并解决可能会出现的中断重入问题。最后实现 I/O 系统，一个由我们自己设计的简单的操作系统就此完成。在这次实验过程中，我通过每一次实验之间代码的不同以及最后呈现的效果的区别，对不同代码对应的操作系统的基本结构、原理和功能有了更清晰明确的理解，也对 NASM 汇编语言有了初步的认识。

这次实验首先深化了我对操作系统的理解。我对操作系统的多进程管理、输入输出处理等方面有了更加深刻和直观的认识。通过亲自实践，我认识到了操作系统是如何协调和管理多个进程的执行，如何处理输入输出请求等。除此之外，我还增加了对 CPU 硬件的了解。实现一个操作系统不可避免地需要与硬件进行交互，而这次实验让我对一些关键芯片有了初步的了解。其中，我对中断控制芯片 8259A、时钟控制芯片 8253 以及键盘控制芯片 8042 有了初步的认识。了解这些芯片的功能和工作原理，让我意识到操作系统是如何与硬件进行通信和控制的。最后，在这次实验中，我还初步接触了 NASM 汇编语言，发现与此前学过的 x86 汇编语言较为类似，在此基础上通过阅读源码对比并实践运行，我逐渐掌握了 NASM 汇编语言的基本使用方式，以及编写汇编指令来实现特定的功能，与 C 语言进行混合编译。

由于之前的一些科研工作涉及到过 Linux 系统，这次实验的环境配置部分没有给我带来过多困扰，但在此前我更多地是了解学习 Linux 系统的相关指令。在这次实验的操作过程中，我开始从原理上深化理解这些指令的意义，这对我的实操能力带来了很大提升。

同时，我非常感谢老师的指导和帮助以及提供的参考资料 [4][3]，没有这些内容作为参考，我无法完成这个实验。通过这次实验，我不仅掌握了操作系统的基

本原理和实践技能，还培养了解决问题的能力和动手实践的精神。这对我的职业发展和学术研究都具有重要意义。我将继续深入学习操作系统的知识，并将其应用到未来的学习和研究中。

## 参考文献

- [1] Sinaean Dean. 80286 与保护模式, 2017.
- [2] michaelcao1980. romimage, 2012.
- [3] mit. Xv6, a simple unix-like teaching operating system, 2017.
- [4] 于渊. *Orange'S*: 一个操作系统的实现. 电子工业出版社, 2009.

## 教师评语评分

评语:

评分:

评阅人:

年 月 日

(备注:对该实验报告给予优点和不足的评价,并给出百分制评分。)