

# Linux 操作系统分析实验文档

2019 年 10 月

## 1 实验介绍

本实验包含以下内容。

- 实验一 初识实验环境
- 实验二 内核引导启动
- 实验三 VGA 与串行端口
- 实验四 内存管理与分页
- 实验五 缺页异常
- 实验六 中断与系统调用
- 实验七 进程控制
- 实验八 文件系统

### 1.1 NEUOS 简介

本实验代码由 NEUOS 与 Linux v0.11 组成。NEUOS 是一个基于 Linux v0.11 内核的教学用操作系统。通过学习早期 Linux 内核，排除当前内核中复杂而庞大的实现细节，我们能较为完整地了解内核实现原理。

由于 Linux 下的 C 语言开发通常使用 AT&T 内联汇编，NEUOS 中的汇编代码为 AT&T 格式。本实验需要一些简单的汇编语言知识。

## 1.2 源码

运行 NEUOS 需要在 Linux 环境下。我们提供了一个虚拟机镜像“NEUOS Lab Environment”用作实验环境，详见实验一关于实验环境的描述。实验环境的 Documents 目录中内置了下列三份源码，本实验在实验材料 neuos-material 中进行。

- NEUOS : <https://github.com/VOID001/neu-os>
- Linux 0.11 : <https://github.com/lzw429/Linux-0.11>
- 实验材料: <https://github.com/lzw429/neuos-material>

## 1.3 运行 (README)

NEUOS 源码使用 make 构建。打开终端，使用 cd 切换到 neu-os 目录，键入

```
make
```

make 将生成软盘镜像作为启动盘，这是启动 NEUOS 所必需的。

生成镜像并使用有调试功能的 Bochs 运行 NEUOS，在命令行键入：

```
make run_bochs
```

或者

```
make  
bochs -q
```

生成镜像并使用 QEMU 运行 NEUOS，在命令行键入：

```
make run
```

每次修改代码后，命令行键入：

```
make clean  
make
```

即可重新构建。

关于 QEMU 与 Bochs 的运行参数，请查看 makefile.

## 1.4 预备

完成实验所需的必备知识包括对计算机组成原理、数据结构与算法的掌握，如果了解 AT&T 汇编语言或 GNU 工具链，完成本实验会相对轻松。

## 1.5 学习建议

本实验文档中的“实验资料”，并非只为有限的实验内容而编写。实验资料对读者理解 Linux 的早期内核大有裨益。在开始每一项实验前，我们建议首先浏览写在“实验内容”后的“实验资料”。

在内核引导启动部分，建议着重关注操作系统如何与 BIOS 协调将计算机启动；在进程调度与文件系统部分，建议着重关注数据结构与算法的设计。

## 1.6 参考资料

- 《Linux 内核完全注释》赵炯（对于理解 Linux 0.11 十分必要）
- 《Linux 内核设计与实现》Robert Love
- 《深入理解 Linux 内核》Daniel Bovet, Marco Cesati
- 英特尔® 64 位和 IA-32 架构开发人员手册<sup>1</sup>

---

<sup>1</sup><https://software.intel.com/en-us/articles/intel-sdm>

## 1.7 实验反馈

请将您在本实验中遇到的疑惑或任何意见与建议反馈给我们<sup>2</sup>。

# 2 实验一 初识实验环境

## 2.1 实验目的

1. 安装好实验环境，熟悉基本操作
2. 掌握包括 objdump、makefile、QEMU、Bochs 等实验工具的基本用法

## 2.2 NEUOS 实验环境

本实验环境是安装了内核开发工具的 Manjaro GNOME Edition (17.1.11) 虚拟机镜像。在 Windows 或 Linux 上安装 VirtualBox，然后下载该虚拟机镜像（约 2.5 GB）<sup>3</sup>。

在 VirtualBox 菜单栏中点击“管理 - 导入虚拟电脑”以导入该镜像。推荐使用该镜像以避免繁琐的安装以及可能存在的依赖问题。另外也可自行在 Linux 环境中安装以下工具作为实验环境：git、gcc、gdb、make、qemu、bochs。

实验环境中的 oshacker 账户的默认密码是“oshacker”。若在 Ubuntu 下使用 VirtualBox 启动虚拟机镜像时，出现“kernel driver not installed”的错误提示，访问此处<sup>4</sup>。若虚拟机运行不流畅，可在 VirtualBox 设置中调整内存大小、显存大小及处理器数量。进入实验环境后，虚拟机的显示比例或许不正常，尝试在 VirtualBox 的“显示”设置中修改缩放率，或在 Manjaro 中修改显示设置。在桌面上右击，选择 Display Settings 即可。

不建议将系统语言设置为中文，某些软件可能出现乱码，并且在调试程序时使用英文调试信息在搜索引擎中检索是必要的。

---

<sup>2</sup><https://github.com/lzw429/neuos-guide/issues>

<sup>3</sup>百度网盘链接：<https://pan.baidu.com/s/1agqKofK4zD-vMlw4uykplg>，密码：7375

<sup>4</sup><https://askubuntu.com/questions/760671/could-not-load-vboxdrv-after-upgrade-to-ubuntu-16-04-and-i-want-to-keep-secure>

启动虚拟机需要开启 VT-x 功能，可在 BIOS 中设置<sup>5</sup>。如果不支持 VT-x，您的电脑无法使用虚拟机。

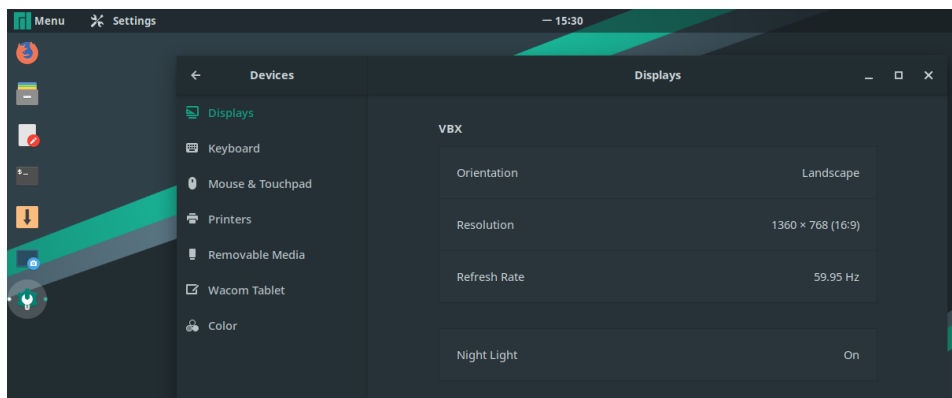


图 1: Manjaro Display Settings

NEUOS Lab Environment 的桌面是空白的，但它已包含实验所需的必要工具 GCC, GNU Toolchain, dd, Makefile, Bochs, QEMU 等。左侧 Dock 栏是常用软件。点击左下角按钮可查看应用列表。

打开 Files，**实验材料** “neuos-material” 位于 Documents 目录中。为确保实验材料是最新版本，进行所有实验前请打开终端，使用 cd 切换到 neuos-material 目录内，保证虚拟机联网状态，键入”git pull” 拉取最新版。

GCC (GNU Compiler Collection) 是由 GNU 开发的编程语言编译器，支持的语言包括 C 语言。它是以 GPL 许可证所发行的自由软件，也是 GNU 计划的关键部分。GCC 原本作为 GNU 操作系统的官方编译器，现已被大多数包括 Linux 的类 Unix 操作系统采纳为标准的编译器。

Objdump 是显示二进制文件信息的工具。dd <sup>6</sup>的用途为转换和复制文件。make 是一个工具程序，用于简化重复编译和重复链接进程。Makefile<sup>7</sup>是由 make 程序引用的文本文件，它描述了目标的构建方式，并包含诸如源文

<sup>5</sup><https://jingyan.baidu.com/article/8ebacdf0df465b49f65cd5d5.html>

<sup>6</sup>了解 dd 命令: [https://www.ibm.com/support/knowledgecenter/zh/ssw\\_aix\\_71/com.ibm.aix.cmds2/dd.htm](https://www.ibm.com/support/knowledgecenter/zh/ssw_aix_71/com.ibm.aix.cmds2/dd.htm)

<sup>7</sup>学习 Makefile: <https://seisman.github.io/how-to-write-makefile/overview.htm>

件级依赖关系以及构建顺序依赖关系之类的信息。

**练习** 尝试纠正 Makefile 文件 exp1/Makefile 中的错误，找到错误并解释其错误原因，其中至少有 3 处参数错误或语法错误，其中一个错误与 gcc 优化参数有关。

在改正错误后，你可能需要实际验证该 Makefile 是否正确。任意编写一段可正常运行的 C 程序，命名为 hello.c，作为被编译的源代码。在 exp1 目录下使用终端，键入 make，测试可执行文件能否正常地通过该 Makefile 生成、生成的可执行文件能否正常运行。

**练习** exp1/binary 程序可以在屏幕上打印一串字符，但在此之前用户需要输入正确的密码。请通过查阅文档了解 objdump 的用法，使用 objdump 程序查看二进制文件 exp1/binary 的 section 信息。其中有一个 section 为 “The password is ...”，请找到该密码，并运行 binary 输入密码，使程序打印字符串。

请解释你如何通过 objdump 查看二进制文件的 section 信息，并展示其中的密码与 binary 打印的字符串。

## 2.3 Bochs

Bochs 是一种 x86 PC 仿真器和调试器。它提供对整个 PC 平台的仿真<sup>8</sup>，包括一个或多个处理器和各种不同的 PC 外围设备。Bochs 提供的并不是现代意义上的虚拟化，而是模拟。

Bochs 模拟器的启动需要配置文件，如 neu-os 目录下已写好的 bochsrc。要在 Bochs 中启动 NEUOS，启动命令行，打开 neu-os 目录，键入

```
make
bochs -q
```

---

<sup>8</sup>使用 Bochs 进行平台仿真: <https://www.ibm.com/developerworks/cn/linux/l-bochs/index.html>

```

X _ 终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
914: c3          retq
915: 90          nop
916: 66 2e 0f 1f 84 00 00  nopw  %cs:0x0(%rax,%rax,1)
91d: 00 00 00

0000000000000920 <__libc_csu_fini>:
920: f3 c3      repz retq

Disassembly of section The_passcode_is_xianameng:

0000000000000922 <ans>:
922: 55          push  %rbp
923: 48 89 e5    mov   %rsp,%rbp
926: 90          nop
927: 5d          pop   %rbp
928: c3          retq

Disassembly of section .fini:

000000000000092c <.fini>:
92c: 48 83 ec 08  sub  $0x8,%rsp
930: 48 83 c4 08  add   $0x8,%rsp
934: c3          retq

```

图 2: objdump 查看 section 示例

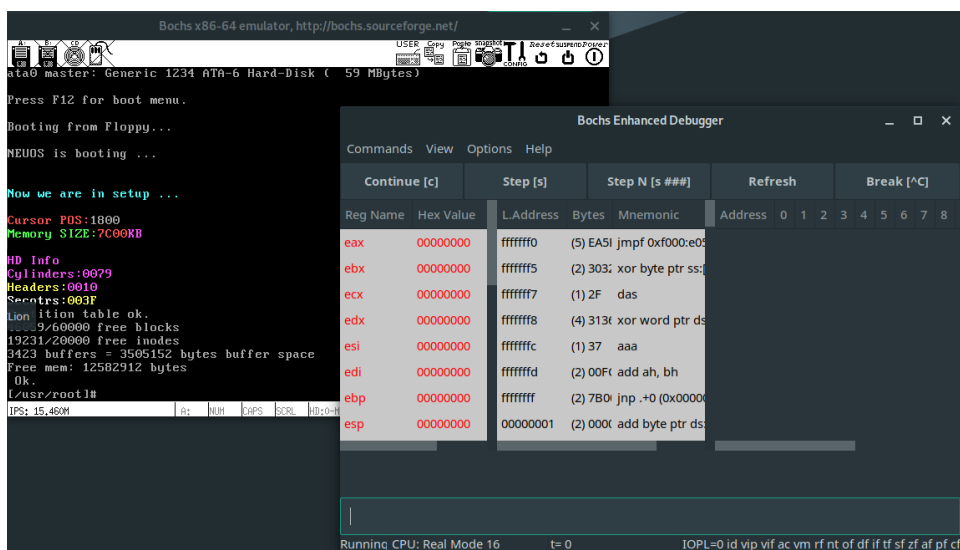


图 3: Bochs 调试界面

或者直接键入

```
make run_bochs
```

`run_bochs` 命令定义在 `Makefile` 中。Bochs 会读取 `bochsrc` 的配置信息，启动模拟器。执行该命令后，在虚拟机中，终端和 bochs 界面可能重叠在显示屏幕上，你可能需要留意 bochs 界面是否已经启动。

如图 3 中的 Bochs Enhanced Debugger 窗口是 Bochs 的调试器，左侧是寄存器信息，中间是将执行的汇编代码，右侧用于显示地址信息。顶部的 Continue 按钮会使模拟器一直执行代码，直到发生错误、遇到断点或等待用户操作；Step 按钮会向下执行一步，Step N 按钮会向下执行指定步数。调试器的底部是 Bochs 的命令行。点击顶部菜单栏中的 View，可查看 GDT、IDT 等信息。图 3 中的 Bochs x86-64 emulator 是模拟器的运行窗口。

**练习** 尝试在 Bochs 中启动 NEUOS，NEUOS 源码位于 NEUOS Lab Environment 的 Documents 目录下。

在 Bochs 中启动 NEUOS 时，在线性地址 `0x000f7b14` 处设置断点，查看调试信息。请阐述你如何在 Bochs 中设置与删除断点，并展示 Bochs 运行至断点时的截图。运行到断点后，Bochs 又如何继续执行指令？

## 2.4 QEMU

与 Bochs 类似，QEMU<sup>9</sup>是一个面向完整 PC 系统的开源仿真器，它允许实现高级概念上的仿真，如对称多处理系统和其他处理器架构。

与 Bochs 不同，启动 QEMU 模拟器需要以命令指定参数。要在 QEMU 中启动 NEUOS，打开命令行，进入 `neu-os` 目录，键入

```
make
```

---

<sup>9</sup>使用 QEMU 进行系统仿真：<https://www.ibm.com/developerworks/cn/linux/l-qemu/>



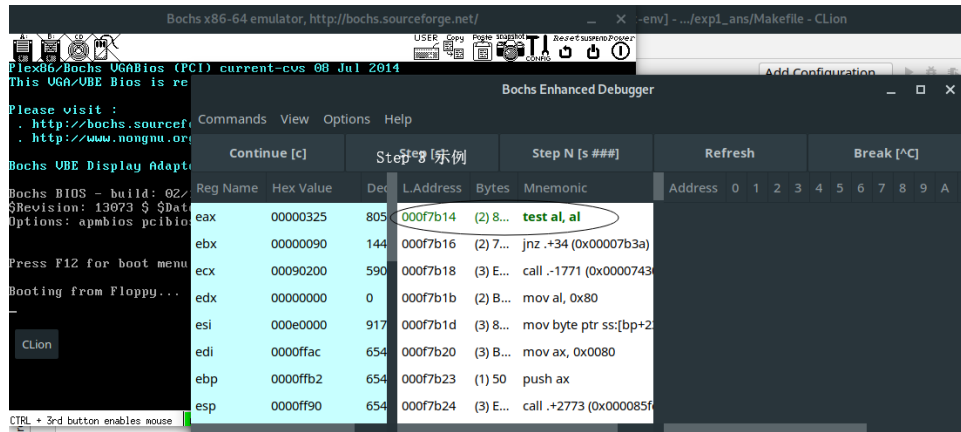


图 4: Bochs 运行至断点

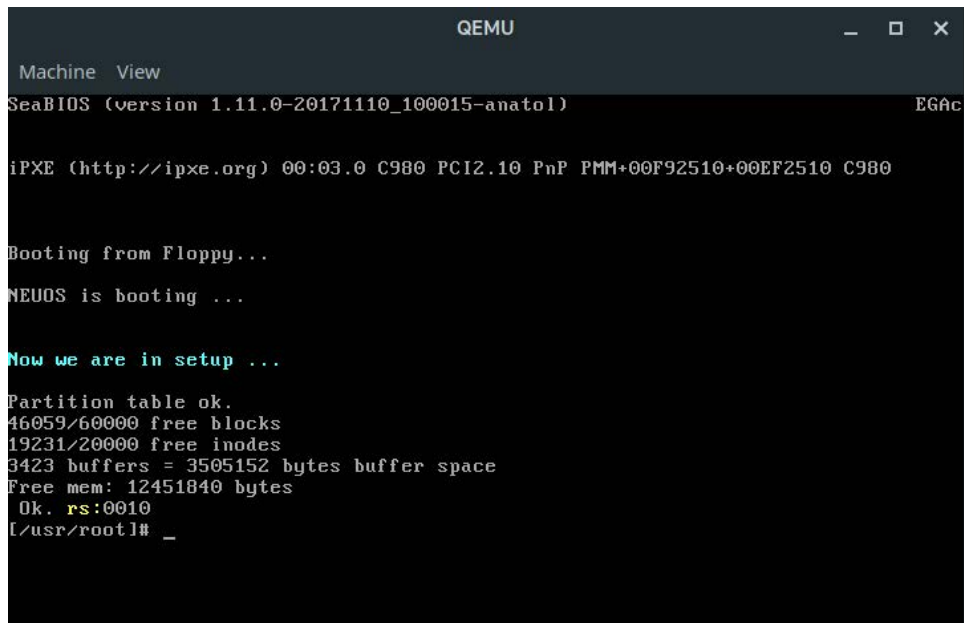


图 5: QEMU 运行界面

```
qemu-system-i386 -m 16M -boot a -fda bootimg -serial stdio
```

Make 会生成 NEUOS 中的 bootimg 镜像，然后运行 QEMU。或者也可直接键入：

```
make run
```

run 命令定义在 Makefile 中。

运行 QEMU 的各参数的意义是：

- i386：指定模拟的处理器
- -m 16M：指定内存大小
- -boot a：从软盘驱动器 A：启动
- -fda bootimg：指定系统镜像
- -serial stdio：将虚拟串口重定向到 stdio；按下 Ctrl + c，QEMU 将立即终止。

**练习** 按照上述流程，尝试使用 QEMU 启动 NEUOS。

## 2.5 文本编辑器

实验环境中包含系统预装的文本编辑器与 Vim。Dock 栏中的 Add/Remove Software 是包管理器，可使用它直接安装 Vim、Emacs、Atom 或 CodeBlocks 等工具。NEUOS 使用 make 构建，但也可使用 IDE 编辑代码。如 CLion 集成了全局搜索、终端、图形化版本控制、查看代码结构等功能，能提升开发效率。

**拓展学习** 观看视频教程 <https://www.bilibili.com/video/av12169693/>

**拓展学习** 尝试在 Virtual Box 虚拟机与其宿主机之间互传文件。

## 3 实验二 内核引导启动

### 3.1 实验目的

1. 掌握系统中断的调用。
2. 熟悉 system 加载的过程。
3. 熟悉 GDTR 与 IDTR 的设置。
4. 了解全局描述符格式。
5. 了解由实模式到保护模式的跳转过程。

### 3.2 BIOS 启动过程

计算机加电后，并非直接执行操作系统，而是执行系统初始化软件，完成基本 I/O 初始化和引导加载功能。

以 Intel 80386 为例，该初始化软件就是基本输入输出系统 (Basic Input Output System, BIOS)，它存储在一个只读存储器 (ROM) 中。计算机加电后，CPU 从物理地址 0xFFFFFFF0 (4G 地址的顶端) 开始执行。在 0xFFFFFFF0 处只存放了一条跳转指令，通过该指令跳到 BIOS 例行程序起始点。

BIOS 在执行硬件自检和初始化后，会选择一个启动设备 (如软盘、硬盘、光盘等)，并读取该设备的第一扇区 (即主引导扇区或启动扇区) 到内存的特定地址 0x7c00 处，然后 CPU 控制权会转移到该地址继续执行。

### 3.3 引导扇区

引导扇区 (boot sector) 可分为主引导扇区 (MBR) 和分区引导扇区 (DBR)。主引导扇区<sup>10</sup>是磁盘的 0 柱面 0 磁头 1 扇区，大小为 512 字节。

---

<sup>10</sup>[https://en.wikipedia.org/wiki/Master\\_boot\\_record](https://en.wikipedia.org/wiki/Master_boot_record)

该扇区可被 BIOS 识别为引导扇区的条件是其第 510 和第 511 字节分别为 0x55 和 0xAA。在启动时，BIOS 会将该扇区装入内存的特定位置，并使 CS:IP（代码段寄存器和指令指针寄存器）指向此处，执行引导扇区的代码。

我们可以利用引导扇区，在没有操作系统的情况下，开机后直接执行我们编写的代码。而操作系统中最先被执行的代码，就是下文将提到的 bootsect.s。

### 3.4 内核引导启动程序

NEUOS 的内核引导启动程序位于 exp2/boot 目录下，主要包含以下文件。

- bootsect.s 磁盘引导块程序，驻留在引导扇区中。在计算机接通电源，ROM BIOS 自检后，引导扇区中的内容由 BIOS 加载到内存 0x7C00 处。
- setup.s 主要作用是利用 ROM BIOS 中断读取机器系统数据，并将这些程序保存到 0x90000 开始的位置，即覆盖了 bootsect 程序所在的内存。由于 bootsect.s 已执行完毕，数据的保存不会影响程序正常运行。读取的数据如表 1 所示。另外，setup 程序将 system 模块从以 0x1000:0000 开始的位置整块向下移动到 0x0000:0000 处，并加载中断描述符表寄存器 IDTR 和全局描述符表寄存器 GDTR，设置 2 个 GDT 描述符，开启 A20 地址线，并重写中断控制芯片 8259A（本次实验未涉及），将控制寄存器 CR0 第 0 比特位置为 1，从而进入 32 位保护模式。
- binary.s 进入保护模式后运行的程序，它将打印一行字符串来显示工作状态。

**练习** read\_it 子程序在 exp2/bootsect.s 中，用于快速读取软盘中内容。认真阅读此段代码和注释，包括其中的 read\_track 子程序。

表 1: setup 程序读取并保留的参数			
内存地址	长度	名称	描述
0x90000	2B	光标位置	列号 (0x00-最左端), 行号 (0x00-最顶端)
0x90002	2B	扩展内存数	系统从 1 MB 开始的扩展内存数值 (KB)
0x90004	2B	显示页面	当前显示页面
0x90006	1B	显示模式	
0x90007	1B	字符列数	
0x90008	2B	??	
0x9000A	1B	显示内存	显示内存 (0x00-64k, 0x01-128, 0x02-192k, 0x03-256k)
0x9000B	1B	显示状态	0x00-彩色, I/O=0x3dX; 0x01-单色, I/O=0x3bX
0x9000C	2B	特性参数	显示卡特性参数
....			
0x90080	16B	硬盘参数表	第 1 个硬盘的参数表
0x90090	16B	硬盘参数表	第 2 个硬盘的参数表 (如果没有, 则清零)
....			
0x901FC	2B	根设备号	根文件系统所在的设备号 (bootsect.s 中设置)

表 2: 数据段 TYPE

十进制值	TYPE	说明			数据段
		E	W	A	
0	0	0	0	0	只读
1	0	0	0	1	只读、已访问
2	0	0	1	0	读写
3	0	0	1	1	读写、已访问
4	0	1	0	0	只读、向下扩展
5	0	1	0	1	只读、向下扩展、已访问
6	0	1	1	0	读写、向下扩展
7	0	1	1	1	读写、向下扩展、已访问

尝试在报告中以流程图或伪代码等形式描述程序。请注意格式规范，逻辑正确。

**练习** 尝试在 `exp2/setup.s` 文件中分别找到开启保护模式、设置 GDTR 的相关代码，并简要解释相关代码。

**练习** 全局描述符表（GDT）在 `exp2/setup.s` 的 `gdt` 标号后定义。请阅读 2 个 GDT 的定义，给出其基地址、段限长、`type` 类别，并阐述得出答案的具体依据。详细信息可参考 Intel® 64 and IA-32 Architectures Software Developer’s Manual “第 2756 页。

GDT 格式如图 6 所示，由图可见 1 个 GDT 长度为 64 比特位，需要一个 16 位的十六进制数表示。其中，TYPE 类型如表 2、表 3 所示。请思考，Intel x86 架构采用的是大端模式还是小端模式？

<sup>a</sup><https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>

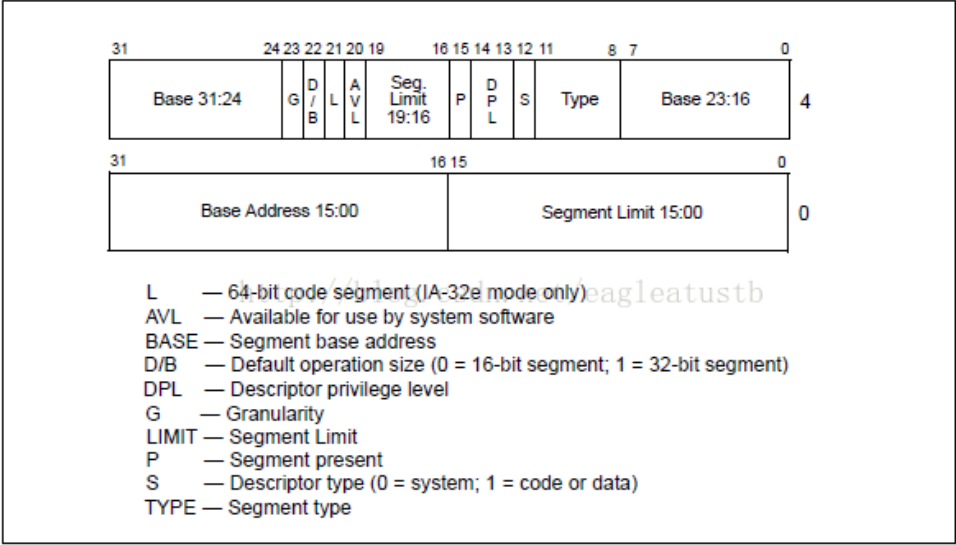


Figure 3-8. Segment Descriptor

图 6: GDT 格式

表 3: 代码段 TYPE

		TYPE			说明
十进制值		C	R	A	代码段
8	1	0	0	0	只执行
9	1	0	0	1	只执行、已访问
10	1	0	1	0	执行、可读
11	1	0	1	1	执行、可读、已访问
12	1	1	0	0	只执行、一致
13	1	1	0	1	只执行、一致、已访问
14	1	1	1	0	执行、可读、一致
15	1	1	1	1	执行、可读、一致、已访问

### 3.5 BIOS 写字符中断

BIOS 中断是 BIOS 提供的中断服务，注意与操作系统提供的系统中断相区别。系统中断调用的操作系统定义的中断服务程序，与 BIOS 没有直接联系。

写字符中断的信息如下。<sup>1112</sup>

int 10h (AH = 13h) Write String  
AL = Write mode  
BH = Page Number  
BL = Color  
CX = String length  
DH = Row  
DL = Column  
ES:BP = Offset of string

**练习** 下面是 exp2/bootsect.s 中一段调用 BIOS 写字符中断 int 10h (AH = 13h) 的 AT&T 汇编代码。请阅读这段代码，尝试具体解释这段代码如何实现了 int 10h。

```
print_msg:
    mov     $0x03, %ah
    xor     %bh, %bh
    int     $0x10

    mov     $20, %cx
    mov     $0x0007, %bx
    mov     $msg1, %bp
```

---

<sup>11</sup>[https://en.wikipedia.org/wiki/INT\\_10H](https://en.wikipedia.org/wiki/INT_10H)

<sup>12</sup>Color: [https://en.wikipedia.org/wiki/BIOS\\_Color\\_Attributes](https://en.wikipedia.org/wiki/BIOS_Color_Attributes)



```
mov    $0x1301, %ax
int     $0x10
```

### 3.6 bootsect 源码汇编、链接过程

使编写好的汇编源码 bootsect.s 在模拟器上运行，需要用到 as、ld、objcopy 三个程序。其基本过程如下，bootsect.s 经 as 汇编后生成目标文件 bootsect.o；ld 将 bootsect.o 链接后生成 ELF 格式文件，链接时，ld 会默认给代码加上 0x08048000 的偏移值，我们需要让这个偏移值变为 0，可通过 -Ttext 参数设置偏移值；最后 objcopy 将链接生成的 ELF 格式文件处理为 QEMU/bochs 可读取的 BIN 格式。

在此我们需要了解 BIN 格式与 ELF 格式文件的不同。

BIN 格式即 raw binary，这种文件只包含机器码；ELF 格式除了机器码外，还包含其他信息，诸如段 (section) 的加载地址，运行地址，重定位表，符号表等。

ELF 格式即可执行可链接格式 (Executable and Linkable Format)，这种格式文件的体积比对应的 BIN 格式要大。

前述的汇编出的.o 文件是 ELF 格式的文件，而此处我们需要让 QEMU 使用 BIN 格式的镜像。

关于上述操作使用的 as、ld、objcopy 详细参数，可在 neu-os 目录下的 Makefile 中查看。

### 3.7 实模式

在引导程序接替 BIOS 后，计算机处于实模式 (16 位) 运行状态，在这种状态下软件可访问的物理内存空间在 1MB 以内，且不支持分页机制。

实模式是不安全的。实模式下数据和代码位于内存不同区域，操作系统并没有区别对待二者，且每一个指针都指向实际的物理地址。用户程序的指针有能力指向操作系统区域，甚至修改其内容。

实模式下逻辑地址转换到物理地址的方式是：将逻辑地址的段(segment)左移 4 位，然后加上逻辑地址中的偏移 (offset)，即得到物理地址。实模式

下，多个逻辑地址可能被映射到同一物理地址。

为什么 16 位实模式仍然存在？为了向下兼容，而向下兼容也是有利有弊的。

### 3.8 保护模式

在保护模式下，80386 的全部 32 根地址线才有效，可寻址 4GB 的线性地址空间和物理地址空间，可访问 64TB 的逻辑地址空间，可采用分段存储管理机制和分页存储管理机制；保护模式下的特权级机制，在实现资源共享的同时保证代码数据的安全及任务的隔离。

### 3.9 段式存储管理机制

在保护模式下才能使用分段机制。分段机制将内存划分为段 (segment)，每个段有起始地址和长度限制。代码段、数据段所指的段与这个段是同一含义。

分段机制所需的数据结构是**段描述符**和**段描述符表**；需要的参数是逻辑地址，它由**段选择子** (selector) 和**段偏移** (offset) 组成。

分段机制能将逻辑地址通过查表转换到线性地址，过程是：CPU 将逻辑地址中的段选择子的内容作为段描述符表的索引，找到表中对应的段描述符，然后将段描述符中保存的段基址加上逻辑地址中的段偏移值，形成线性地址。若不启动分页机制，线性地址等同于物理地址。

这一转换过程对于应用程序员来说是不可见的。线性地址长度为 32 位，线性地址空间容量为 4GB。

**全局描述符表**是一个保存多个段描述符的“数组”，其起始地址保存在全局描述符表寄存器 GDTR 中。GDTR 长 48 位，其中高 32 位为基地址，低 16 位为段界限。GDT 不能由 GDT 自身以内的描述符进行描述，因此处理器将 GDTR 寄存器作为 GDT 的特殊系统段。注意，全局描述符表第一个段描述符是空的。

**练习** 阅读本实验已编写好的 Makefile 文件，在读懂其中指令的基础上，结合实验一所学，通过 Bochs 运行编写好的程序。

使用 Bochs 顶部的 View 菜单中查看 GDT 的功能，保存运行截图  
如图 7 所示。

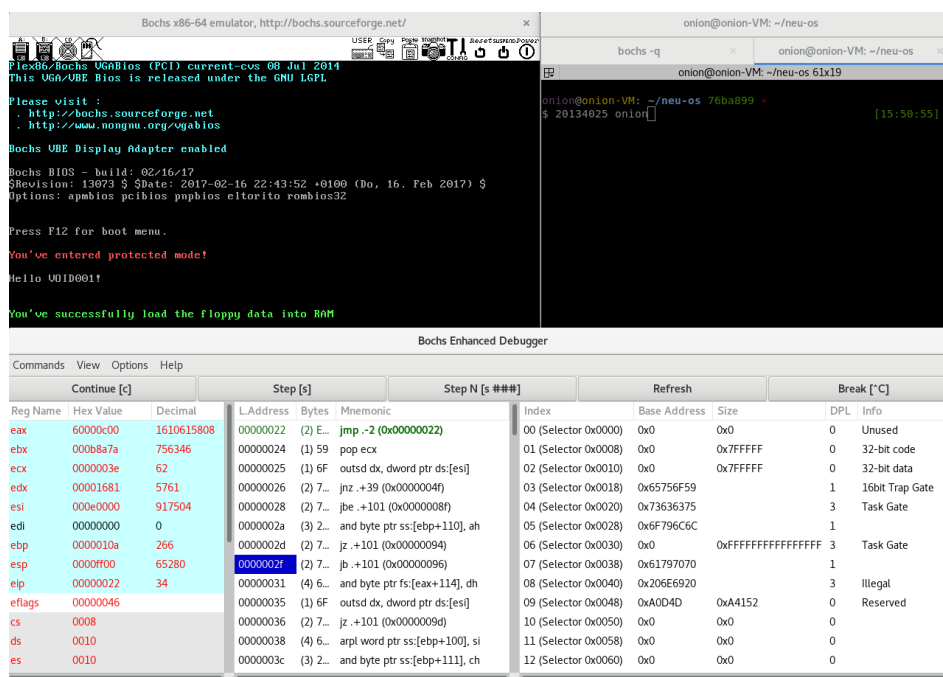


图 7: Bochs 查看 GDT

### 3.10 保护模式下的特权级

保护模式下特权级有 4 种，权限由高到低编号为 0-3。Linux 仅使用了 0 号和 3 号，它们分别是内核态和用户态。

- INT table: [http://stanislavs.org/helppc/int\\_table.html](http://stanislavs.org/helppc/int_table.html) 用于查询 int 中断用法。
- 本次实验视频教程: <https://www.bilibili.com/video/av13053659/>
- Intel® 64 and IA-32 Architectures Software Developer's Manual: <http://www.intel.com/content/www/us/en/processors/ia-32-ia-64/architectures-software-developer-manual-325462.html>

s://software.intel.com/sites/default/files/managed/39/c5/3  
25462-sdm-vol-1-2abcd-3abcd.pdf

**拓展学习** [从零开始的操作系统编写 Lesson 0x01] <https://www.bilibili.com/video/av12367780/?t=1945>

**拓展学习** 编写 Linker Script: 打开 exp2/ld-bootsect.ld, 尝试为 bootsect.s 编写链接脚本。建议参考上述视频。

## 4 实验三 VGA 与串行端口

### 4.1 实验目的

1. 了解 Linux 中 VGA 的实现, 掌握 printk 等字符打印函数的实现方法。
2. 了解 Linux 通过串行端口与终端交换信息的函数实现。

**视频图形阵列** (Video Graphics Array, 简称 VGA) 是 IBM 于 1987 年提出的一种电脑显示标准。运用该标准的接口被称为 VGA 端子, 通常用于在电脑的显示卡、显示器及其他设备发送模拟信号。本次实验关注于内核对彩色字符模式显示缓冲区的控制。

### 4.2 80 \* 25 彩色字符模式显示缓冲区的结构

内存地址中, B8000h \ BFFFFh 共 32KB 的空间, 是 80\*25 彩色字符模式的显示缓冲区。在这个地址空间中写入数据, 写入的内容会立即出现在显示器上。

在 80\*25 彩色字符模式下, 显示器可显示 25 行, 每行 80 个字符, 每个字符可以有 256 种属性, 包括背景色、前景色 (即字体色)、闪烁、高亮等组合而成的属性。

一个字符在显示缓冲区中占用 2 个字节, 分别存放其 ASCII 码和属性。一屏的内容在显示缓冲区中共占用占  $2 * 25 * 80 = 4000$  字节。

表 4: 属性字节中每位的具体含义

7	6	5	4	3	2	1	0
BL	R	G	B	I	R	G	B
闪烁	背景色	背景色	背景色	高亮	前景色	前景色	前景色

显示缓冲区分 8 页，每页 4 KB，显示器可显示任意一页的内容。一般地，显示第 0 页内容，即显示 B8000H B8F9FH 中的共 4000 字节的内容。

在一页显示缓冲区中，一行的 80 个字符占用 160 字节，偏移 000 09f 对应显示器上的第一行，偏移 0A0 13f 对应显示器上的第二行，以此类推。

在属性字节中，闪烁、背景色、高亮与前景色是按位设置的，如表 4。

下面介绍内核中用于 VGA 的一些函数。

#### 4.2.1 函数 `video_putchar_at(char ch, int x, int y, char attr)`

本函数用于在屏幕指定位置 (x,y) 处输出指定字符串 ch，并且指定字符串 ch 后光标的颜色 attr。其中，x 是行数，y 是列数。

ch 与 attr 变量不需要额外的处理，仅需要将这两个变量赋值到合适的内存地址处。

#### 4.2.2 不定参数函数

在内核调用函数 printk 时，需要被格式化输出的内容长度是不定的，因此 printk 是一个不定参数函数。函数 printk 的实现中，将用到 <stdarg.h> 库中的数据结构或函数：va\_list、va\_start、va\_arg。<sup>13</sup>

#### 4.2.3 函数 `printf(char *fmt, ...)`

printf 与 printk 在实现的功能上几乎一致，根据使用 C 语言函数 printf 的经验，可知这个格式化输出函数的首个参数 char \*fmt 应当表示一段字符串的首位，以 \*fmt 为首的字符串描述了格式化输出的格式。

<sup>13</sup><https://msdn.microsoft.com/zh-cn/library/kb57fad8.aspx>

这个字符串中可能含有以“%”为首位的子串，表示一个数字、一个 char 类字符或一段字符串，它具体的值即 printk 的某个在 \*fmt 之后的参数。

**练习** 打开 exp3/kernel/printk.c，尝试依次实现 video\_putchar\_at 和 printk 函数。

首先，思考一个屏幕坐标 (x, y) 对应的显存地址是如何计算的？其中，显存地址的首位在 exp3/kernel/printk.c 中由 video\_buffer 指针表示。

请通过调用函数 printnum、video\_putchar、va\_start、va\_arg，尝试为函数 printk 实现如下功能<sup>a</sup>：

1. 字符串无“%”时能直接在屏幕上输出字符串的内容
2. 读取到“%d”，根据某个参数输出一个有符号十进制整数
3. 读取到“%u”，根据某个参数输出一个无符号十进制整数
4. 读取到“%x”，根据某个参数输出一个无符号十六进制整数
5. 读取到“%c”，根据某个参数输出一个字符
6. 读取到“%s”，根据某个参数输出一段字符串
7. 读取到“%%”，输出百分号“%”

注意，你需要仔细考虑各种状态转移。例如，你可能认为读取到“%”时是解析一项参数的开始，但它同样可能是该字符串的最后一个字符，即一个百分号可能没有后继的字符。

若函数 video\_putchar\_at 与函数 printk 已成功实现，使用 QEMU 运行 neuos，将见到如图 8 的界面。printk 函数是由 exp3/kernel/main.c 调用的。

---

<sup>a</sup>参考网站：[https://wiki.osdev.org/Printing\\_To\\_Screen](https://wiki.osdev.org/Printing_To_Screen)

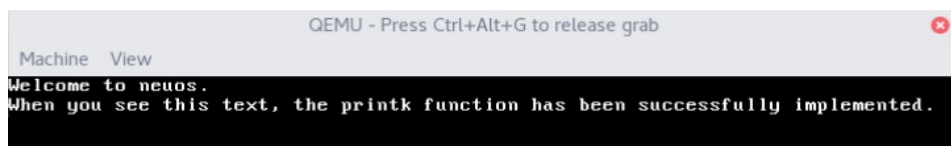


图 8: 函数 printk 成功实现

**练习** 尝试通过调用 `printk.c` 中的两个函数 `memcpy` 与 `video_putchar_at` 实现滚屏函数 `roll_screen`. 滚屏函数被调用的条件应当是, 需要输出字符到当前屏幕的最后一行之后。因此需要将当前屏幕的内容整体上移, 使得屏幕最后一行能够输出新的内容。

要验证函数 `roll_screen` 是否成功实现, 可打开目录 `exp3/kernel/main.c`, 使用函数 `printk` 输出足够多的字符, 再运行 `neuos`, 测试能否成功滚屏。

### 4.3 串行端口

**串行端口** (Serial port) 主要用于串行式逐位数据传输。可用于连接外置调制解调器、打印机、路由器等设备。在消费电子领域已被 USB 替代, 在网络设备中仍是主要的传输控制方式。

通过串行接口, Linux 设备, 即使并非计算机, 也可实现与其他设备的通信; 本实验中的串行端口用于将 `neuos` 中的信息打印到 NEU-OS Lab Environment 的终端中。

**练习** 进入 `exp3/kernel` 目录, 打开 `serial.c` 源文件。

函数 `is_transit_empty` 用于判断是否能够传输, 若允许传输, 函数返回 0。

结合函数 `is_transit_empty` 与端口输出函数 `outb`, 实现函数 `s_putchar`. 参考 Serial Ports - OSDev Wiki 4.3 节<sup>a</sup>。

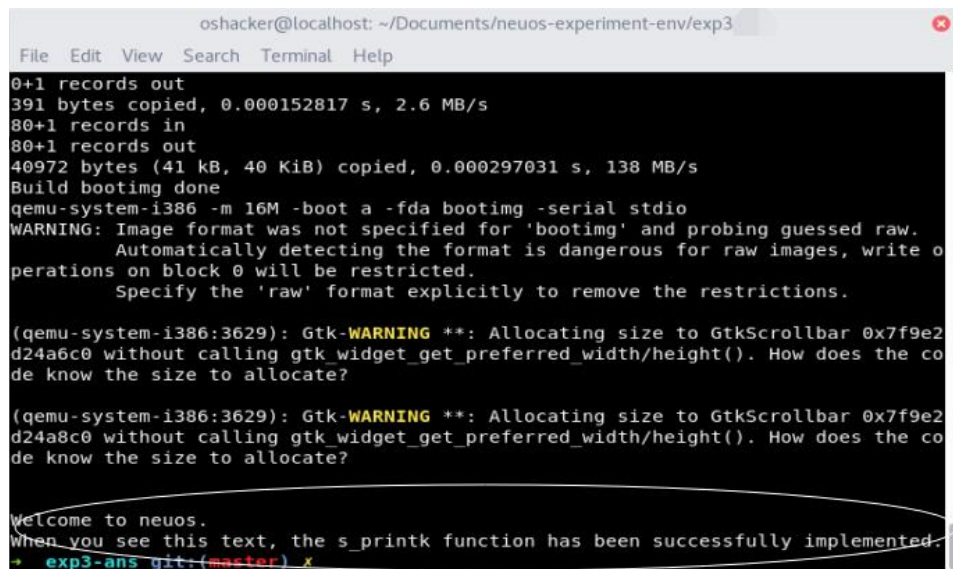
然后, 类似于 `printk` 函数的实现, 请通过调用函数 `s_printnum`、`s_putchar`、`va_start`、`va_arg`, 尝试为函数 `s_printk` 实现如下功能:

1. 字符串无 “%” 时能直接在屏幕上输出字符串的内容

2. 读取到 “%d”，根据某个参数输出一个有符号十进制整数
3. 读取到 “%u”，根据某个参数输出一个无符号十进制整数
4. 读取到 “%x”，根据某个参数输出一个无符号十六进制整数
5. 读取到 “%c”，根据某个参数输出一个字符
6. 读取到 “%s”，根据某个参数输出一段字符串
7. 读取到 “%%”，输出百分号 “%”

若函数 `s_putchar` 与函数 `s_printk` 已成功实现，使用 QEMU 运行 neuos，运行结果如图 9 所示，字符被输出到 os 的终端而不是 QEMU 界面中。

[http://wiki.osdev.org/Serial\\_Ports](http://wiki.osdev.org/Serial_Ports)



```
oshacker@localhost: ~/Documents/neuos-experiment-env/exp3
File Edit View Search Terminal Help
0+1 records out
391 bytes copied, 0.000152817 s, 2.6 MB/s
80+1 records in
80+1 records out
40972 bytes (41 kB, 40 KiB) copied, 0.000297031 s, 138 MB/s
Build booting done
qemu-system-i386 -m 16M -boot a -fda bootimg -serial stdio
WARNING: Image format was not specified for 'bootimg' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.

(qemu-system-i386:3629): Gtk-WARNING **: Allocating size to GtkScrollbar 0x7f9e2
d24a6c0 without calling gtk_widget_get_preferred_width/height(). How does the co
de know the size to allocate?

(qemu-system-i386:3629): Gtk-WARNING **: Allocating size to GtkScrollbar 0x7f9e2
d24a8c0 without calling gtk_widget_get_preferred_width/height(). How does the co
de know the size to allocate?

Welcome to neuos.
When you see this text, the s_printk function has been successfully implemented.
→ exp3-ans git:(master) x
```

图 9: 函数 `s_printk` 成功实现



**练习** 在 `exp3/kernel/serial.c` 源文件的末尾，尝试实现通过串行端口读取信息的代码。仅尝试完成代码<sup>a</sup>，不必实现最终效果。

<sup>a</sup>参考 [http://wiki.osdev.org/Serial\\_Ports](http://wiki.osdev.org/Serial_Ports) 读取数据部分。

**拓展学习** 找到 `printk.c` 源文件 208 行以后的“拓展学习”部分，实现具有指定 8 种颜色功能的函数 `echo`；`echo` 需要调用 `video_putchar_color` 等函数。

实现效果参考 [http://misc.flogisoft.com/bash/tip\\_colors\\_and\\_formatting](http://misc.flogisoft.com/bash/tip_colors_and_formatting)，考虑到实现的难度等因素，以下描述与该参考网站的描述并不完全一致。

例如，`echo(“\033[x;ymneuos”)`；可将“neuos”以 `x` 为前景色，`y` 为背景色输出。其中 `x` 和 `y` 是两个数字，均是 0~7 的整数，其表示的颜色见表 5。`m` 作为颜色的唯一后缀。

如果没有“\033[”指定颜色，函数 `echo` 应直接输出文本。这个格式是严格的，不符合这个格式的字符串应按普通文本直接输出。函数应支持输出多个指定颜色的字符串。

表 5 中的 RGB 指的是表 4 中属性字节的第 0~2 位及第 4~6 位；为了更好的显示效果，此处指定属性字节的第 3 位置为 1，第 7 位置为 0。根据这些信息，并结合表 4，可计算出应填充到属性字节的十六进制数。

如要测试该函数是否正确实现，在 `exp3/kernel/main.c` 中调用函数 `echo` 并填入参数即可。实现效果如图 10 所示。

注意，C 语言中“\\”是斜杠的转义字符，“\0”是 NULL 的转义字符。

## 5 实验四 内存管理与分页

### 5.1 实验目的

1. 了解 Linux 中的页式存储管理，并理解页目录、页表的概念。

表 5: 前景色与背景色代码表示的颜色

x: 字体色	y: 背景色	颜色	RGB 二进制
0	0	BLACK	000
1	1	RED	100
2	2	GREEN	010
3	3	YELLOW	110
4	4	BLUE	001
5	5	MAGENTA	101
6	6	CYAN	011
7	7	WHITE	111

```

1 // head.s后默认运行的程序 //
2
3 #include <linux/kernel.h>
4
5 extern int video_x;
6 extern int video_y;
7
8 void main()
9 {
10     int i;
11     video_init();
12     printk("Welcome to neuos.\n");
13     printk("When you see this text, the printk function has been successfully implemented\n");
14     s_printk("\n\nWelcome to neuos. \n"); s_experiment 4.
15     s_printk("When you see this text, the s_printk function has been successfully implemented\n");
16     echo("\033[4;5mtest1\033[0;7mtest2\033");
17
18 }
19
20

```

QEMU

Welcome to neuos.

When you see this text, the printk function has been successfully implemented.

\033[4;5mtest1\033[0;7mtest2\033

图 10: 函数 echo 实现效果

2. 掌握线性地址到物理地址的转换。
3. 了解 C 语言中的内联汇编。

## 5.2 内存地址空间

逻辑地址（相对地址，虚拟地址）：用户程序经过编译、汇编后形成目标代码，目标代码通常采用相对地址的形式，其首地址为 0，其余地址都相对于首地址而编址。不能用逻辑地址在内存中读取信息。

物理地址（绝对地址，实地址）：内存中存储单元的地址，可直接寻址。

为保证 CPU 执行指令时可正确访问内存单元，需要将用户程序中的逻辑地址转换为运行时可由机器直接寻址的物理地址，这一过程被称为地址重定位。

关于逻辑地址、线性地址与物理地址：Intel 实模式下，逻辑地址就是物理地址；没有启用分页机制时，线性地址就是物理地址。线性地址是逻辑地址到物理地址变换之间的中间层，段中的偏移地址（逻辑地址）加上相应段的基地址构成了线性地址。

- 启动分段机制，未启动分页机制：逻辑地址  $\rightarrow$  (分段地址转换)  $\rightarrow$  线性地址  $\rightarrow$  物理地址
- 启动分段和分页机制：逻辑地址  $\rightarrow$  (分段地址转换)  $\rightarrow$  线性地址  $\rightarrow$  (分页地址转换)  $\rightarrow$  物理地址

## 5.3 分页式存储管理

分页是操作系统中的一种内存管理技术，可使电脑的主存能够使用存储在辅助内存中的数据。操作系统会将辅助内存中的数据分区成固定大小的区块，称为“页”（pages）。当数据不需要在主存中时，操作系统将分页由主存移到辅助内存；当数据被需要时，再将数据取回，加载在主存中。相对于分段，分页允许内存存储于不连续的区块以维持文件系统的整齐。分页是磁盘和内存间传输数据块的最小单位。

## 5.4 页表结构

分页转换功能由内存中的表来描述，该表即是页表 (page table)，存放在物理地址空间中。页表可看做是简单的  $2^{20}$  物理地址数组。线性到物理地址的映射功能可简单地看做是进行数组查找。NEUOS 保护模式下，内存地址长度 32 位。线性地址的高 20 位构成这个数组的索引值，用于选择对应页面的物理（基）地址。线性地址的低 12 位给出了页面中的偏移量，加上页面的基地址最终形成对应的物理地址。

### 5.4.1 NEUOS 的两级页表结构

页表含有  $2^{20}$  (1M) 个表项，每项占用 4 字节。如果作为一个表来存放，最多将占用 4MB 的内存，而 NEUOS 仅支持 16MB 内存。为了减少内存占用量，需要使用两级表。高 20 位线性地址到物理地址的转换也被分为两步，每步转换其中 10 个比特。

第一级表称为页目录 (page directory)。页目录本身被存放在 1 页 4K 页面中，具有  $2^{10}$  (1K) 个 4 字节长度的表项。这些表项指向对应的二级表。线性地址的最高 10 位（位 22-31）用作一级表（页目录）中的索引值来选择  $2^{10}$  个二级表之一。

第二级表称为页表 (page table)，每个页表的长度也是 1 页，最多含有 1K 个 4 字节的表项。每个 4 字节表项含有相关页面的 20 位物理基地址。二级页表使用线性地址中间 10 位（位 12-21）作为表项索引值，以获取含有页面 20 位物理基地址的表项。该 20 位页面物理基地址和线性地址中的低 12 位（页内偏移）组合在一起就得到分页转换过程的输出值，即对应的最终物理地址。

二级表结构允许页表被分散在内存各处，而不需要连续保存，而且不需要为不存在的或线性地址空间未使用部分分配二级页表。

页目录中每个表项都有一个存在 (present) 属性，类似于页表中的表项。页目录表项中的存在属性指明对应的二级页表是否存在。如果存在，直接访问；如果不存在，处理器会产生缺页异常来通知操作系统。页目录表项中的存在属性使操作系统可根据实际使用的线性地址范围来分配二级页表页面。存在位还可用于在虚拟内存中存放二级页表。

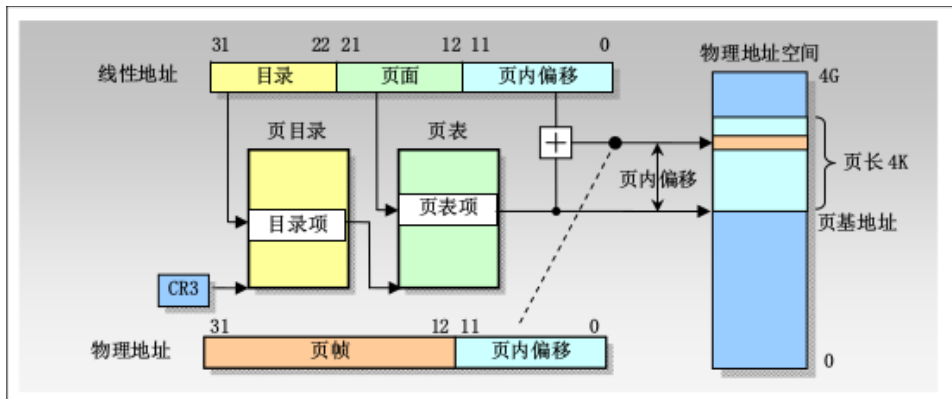


图 11: 线性地址通过两级页表转换为物理地址

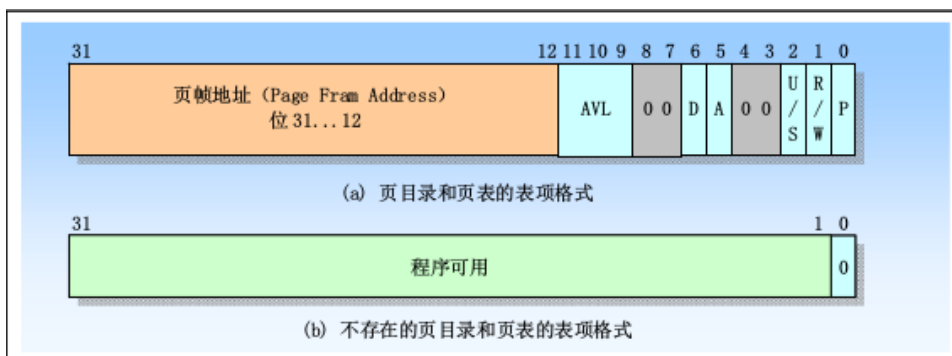


图 12: 页目录和页表的表项格式

表 6: 页表项 0-11 位

位	功能	位为 1	位为 0
S	页面大小	页面大小是 4M (要求开启 PSE)	页面大小是 4K
A	访问	该页被读或写过	该页未被读或写过
D	缓存禁用	该页不会被缓存	该页可被缓存
W	直写/写回	write through	write back
U	权限	用户/Supervisor 均可访问	仅 Supervisor 可访问
R	读写权限	可读可写	只读
P	存在	该页在物理内存中	该页不在物理内存

关于表项 0-11 位的意义, 见表 6 的简介。<sup>14</sup>

**练习** PDE (page directory entry) 指页目录项, PTE (page table entry) 指页表项。

打开 exp4/mm/mm\_test.c, 找到函数 linear\_to\_pte。该函数的功能是将线性地址通过两级页表转化为物理地址。实现该函数要求使用 C 语言位运算。

请尝试实现该函数, 并描述程序通过线性地址获得 pte 的过程。

若转换成功, 返回指向页表项的指针; 若失败, 返回 NULL(0)。

在 linear\_to\_pte 函数中, 首先使用无符号长整型 (32 位) 指针 \*pde 获取页目录地址; 注意, 页目录地址不是页目录的索引值, 而是内存地址, 页目录地址范围是 0x00000000 0x00000fff, 每个 PDE 占用内存 4 字节, 即查找 PDE 需要将线性地址的前 10 位加上 2 位 0。

然后判断, 若 pde 指向的页表不存在 (根据 present 位判断) 或者该地址不在页目录地址范围内 (页目录地址小于 4 KB), 返回 0。

由取内容运算 \*pde 取得页目录项存储的地址, 由该地址的前 20 位获得页表地址, 而后 12 位置为 0, 即使得页内偏移为 0; 通过线性地址 addr 的 12 - 21 位得到页表索引, 页表索引的范围是 0x00000000 0x003ff000。

<sup>14</sup>[https://wiki.osdev.org/Page\\_table#Page\\_Table](https://wiki.osdev.org/Page_table#Page_Table)

最后计算 PTE，由 \*pde 取前 20 位得到的页表地址加页表索引，得到指向页表项的指针。页表地址和页表索引的后 12 位均为 0，所得指针存储的地址后 12 位也为 0。

**练习** 打开 exp4/mm/mm\_test.c，找到函数 mm\_print\_pageinfo。该函数的功能是通过读取页表项显示页的信息。请尝试实现该函数，并描述程序从页表项中获取页的信息的过程。

给定线性地址，尝试读取页表项第 0 位、第 1 位、第 2 位。该函数框架已给出，尝试在该框架内完善；若该页存在于物理内存，**接着**输出 “P”，否则不输出；若该页可读可写，**接着**输出 “R/W”，否则**接着**输出 “RO”；若该页允许 user/supervisor 操作，**接着**输出 “U/S”，否则若仅允许 supervisor 操作，**接着**输出 “U”。

例如，某页存在于物理内存中，可读可写且允许 user/supervisor 操作，则其 Flag 应输出为 “P R/W U/S”。

## 5.5 内联汇编

内联汇编指的是将汇编语言内嵌在高阶语言的源代码中。这可以提升程序的执行效率，汇编语言代替高阶语言可使代码编写不受编译器的限制；可使程序使用处理器的特有指令；汇编语言能更方便直接地进行系统调用。

GCC 提供了两种内联汇编：基本内联汇编语句和扩展内联汇编语句。

### 5.5.1 基本内联汇编

基本内联汇编的基本格式：

```
asm("statements");
```

# 例如，

```
asm("nop"); asm("cli");
```

# “asm” 与 “\_\_asm\_\_” 的含义是完全相同的。

换行处理：如果有多行汇编，每行末尾要加上 “\n\t”，即换行符和 tab 符；目的是使 GCC 将内联汇编代码翻译成一般的汇编代码时能保证换行和留有一定空格。

基本内联汇编的局限：对于基本 asm 语句，双引号内的内容将直接被 GCC 编译成汇编代码，再交给汇编器。如果在内联汇编语句中改变寄存器的值，会影响寄存器原有状态，可能导致程序错误；基本内联汇编也不能实现 C 语言变量与寄存器内容的交换。

### 5.5.2 扩展内联汇编

一个基本但重要的区别是，简单内联汇编只包括指令，而扩展内联汇编包括操作数。

扩展内联汇编的基本格式：

```
asm [volatile] (  
  Assembler Template  
  [ : Output Operands ]  
  [ : Input Operands ]  
  [ : Clobbers ]  
);  
# []中表示参数是可选的。
```

asm 表示汇编代码的开始，其后所跟的 volatile 是可选项，其含义是避免 asm 指令被删除、移动或组合。在执行代码时，若不希望汇编语句被 GCC 优化而改变位置，就需要添加 volatile 关键字。

volatile 与 \_\_volatile\_\_ 同义。

括号中的语句被 “:” 分隔为四个部分，第一部分是汇编代码本身，称为指令部，这是必填语句，其他部分均可选。

指令部：在指令部中，加上前缀 ‘%’ 的数字（如%0，%1）表示的是需要使用寄存器的**样板操作数**。在涉及具体寄存器时，寄存器名前应加上两个百分号 ‘%%’，以免产生混淆。



输出部：输出部规定的是输出变量与样板操作数结合的条件，每个条件称为一个**约束**，每个约束以等号起始，可包含多个约束，相互之间用逗号分隔。例如

`:"=r"(b), "=r"(c)`

其中，`b` 和 `c` 是 C 语言变量，`r` 是限定符，将在下文提到。

输入部：输入约束的格式和输出约束相似，但不带等号。例如，`:"r"(b), "r"(c)`

声明产生副作用的寄存器：在进行某些操作时，除了进行数据输入和输出的寄存器外，还要使用多个寄存器保存中间的计算结果，这样难免修改原有寄存器的内容。在最后这一部分，对产生副作用的寄存器进行说明，使 GCC 采取相应措施保证寄存器原有状态不受内联汇编语句的影响。例如，`:"%ecx", "%eax"`

操作数编号：在内联汇编中用到的操作数从**输出部**的第一个约束开始编号，序号从 0 开始，每个约束记数一次，指令部要引用这些操作数时，只需在序号前加上 `'%'` 作为前缀即可。

限定符：内联汇编语句的指令部在引用一个操作数时总是将其作为 32 位的长字使用，但实际情况可能需要的是字或字节，因此应该在约束中指明正确的限定符，如表 7 所示。

一段内联汇编的示例：<sup>15</sup> <sup>16</sup> <sup>17</sup>

```
/* inline.c */
int main()
{
```

---

<sup>15</sup>Linux 汇编语言开发指南：<https://www.ibm.com/developerworks/cn/linux/l-assembly/index.html>

<sup>16</sup>GCC 扩展内联汇编 · ucore\_os\_docs：[https://chyyuu.gitbooks.io/ucore\\_os\\_docs/content/lab0/lab0\\_2\\_3\\_1\\_4\\_extend\\_gcc\\_asm.html](https://chyyuu.gitbooks.io/ucore_os_docs/content/lab0/lab0_2_3_1_4_extend_gcc_asm.html)

<sup>17</sup>Using the GNU Compiler Collection (GCC): Extended Asm：<https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>

表 7: 限定符

限定符	含义
"m", "v", "o"	内存单元
"r"	任何寄存器
"q"	寄存器 eax, ebx, ecx, edx 之一
"i", "h"	直接操作数
"E" 和 "F"	浮点数
"g"	任意
"a", "b", "c", "d"	分别表示寄存器 eax、ebx、ecx 和 edx
"S" 和 "D"	寄存器 esi、edi
"I"	常数 (0 至 31)

```
int a = 10, b = 0;
__asm__ __volatile__ ("movl %1, %%eax;\n\t"
                      "movl %%eax, %0;"
                      : "=r"(b)      /* 输出 */
                      : "r"(a)       /* 输入 */
                      : "%eax");     /* 不受影响的寄存器 */

printf("Result: %d, %d\n", a, b);
}

// 该程序的功能是将 C 语言变量 a 的值赋给 b.
```

**练习** 打开 exp4/mm/mm\_test.c, 找到函数 mm\_test\_main. 这个函数用于演示内存管理的各项操作。

参考 4.4 节中对扩展内联汇编的介绍, 尝试在这个函数中编写一段内联汇编, 修改线性地址 0xdad233 处的内容。

在 mm\_test\_main 函数中, 该页被设置为只读且 cr0 寄存器中 WP 位被置位, 因此测试这段内联汇编的方法是观察其是否触发了页的写入

保护。

**练习** 阅读《Understanding the Linux Virtual Memory Manager》Chapter 3 Page Table Management<sup>a</sup>，简要论述现代 Linux 与 NEUOS 在页表管理上的不同之处。

<sup>a</sup><https://www.kernel.org/doc/gorman/html/understand/understand006.html>

**拓展学习** 打开 exp4/mm/memory.c，找到函数 get\_free\_page。

该函数是内联汇编编写的，请尝试将它改写为 C 语言函数，写在 exp4/mm/get\_free\_page.c 源文件中。

## 6 实验五 缺页异常

### 6.1 实验目的

1. 了解发生缺页异常的原因。
2. 掌握缺页异常发生后的处理过程。
3. 复习串口的使用方式，掌握使用串口打印调试信息的方法。

### 6.2 页面出错异常处理

运行于开启分页机制的状态下时 ( $PG = 1$ )，若 CPU 在执行线性地址变换到物理地址的过程中时检测到以下条件后，就会引起页面出错异常中断 int 14:

1. 当前地址变换中用到的页目录项或页表项中存在位 P 为 0。
2. 当前执行程序没有足够的特权访问指定的页面。

此时，CPU 会向出错异常处理程序提供两方面信息来诊断及纠正错误。

第一，栈中的一个出错码 (error code)。出错码格式为一个 32 位的长字，但只有最低 3 个比特位有用，它们的名称与页表项中最后三位相同 (U/S、W/R、P)，它们的含义与作用分别为：

- 位 0 (P)，异常是由于页面不存在或违反访问特权而引发。P=0，表示页面不存在；P=1 表示违反页级保护权限。
- 位 1 (W/R)，异常是由于内存读或写操作引起。W/R=0，表示由读操作引起；W/R=1，表示由写操作引起。
- 位 2 (U/S)，发生异常时 CPU 执行的代码级别。U/S=0，表示 CPU 正在执行超级用户代码；U/S=1，表示 CPU 正在执行一般用户代码。

第二，在控制寄存器 CR2 中的线性地址。CPU 会把引起异常的访问使用的线性地址存在 CR2 中，页面出错异常处理程序便可以用这个地址来定位相关的页目录和页表项。

### 6.3 写时复制 (Copy on Write) 机制

写时复制是一种推迟或免除复制数据的一种方法。此时内核并不复制进程整个地址空间中的数据，而是让父进程和子进程共享同一个拷贝。当进程 A 使用系统调用 fork 创建出一个子进程 B 时，由于子进程 B 实际上是父进程 A 的一个拷贝，因此会拥有与父进程相同的物理页。

为了达到节约内存和加快进程创建速度的目的，fork 函数会让子进程 B 以只读方式共享父进程 A 的物理页面，同时将父进程 A 对这些物理页面的访问权限也设为只读（该操作详见 memory.c 中 copy\_page\_tables 函数）。当父进程 A 或子进程 B 任何一方对这些共享物理页面执行写操作时，都会产生页面出错异常 (page\_fault int 14) 中断，此时 CPU 会执行系统提供的异常处理函数 do\_wp\_page 来试图解决该异常。这就是写时复制机制。它把对内存页面的复制操作推迟到实际要进行写操作的时刻，免除了页面不会被写的情况下的页面复制操作。

do\_wp\_page 与 un\_wp\_page 函数在写时复制机制中发挥了重要作用，详见下一节。

## 6.4 内存管理的重要函数

以下函数参数 `error_code` 表示错误码, `address` 表示线性地址。

### 6.4.1 缺页异常处理函数

```
void do_no_page (unsigned long error_code, unsigned long address);
```

该函数是访问不存在页面的处理函数, 页异常中断处理过程中调用的函数, 在 `mm/page.s` 中被调用。

### 6.4.2 写时复制处理函数

```
void do_wp_page (unsigned long error_code, unsigned long address);
```

在 `mm/page.s` 中被调用, 对写时复制机制中导致写入异常中断的物理页面进行取消共享操作 (使用 `un_wp_page` 函数), 并为写进程复制一新的物理页面, 使父进程 A 和子进程 B 各自拥有一块内容相同的物理页面。该函数还将要执行写入操作的这块物理页面标记为可访问的。最后, 从异常处理函数中返回时, CPU 会重新执行刚才导致异常的写入操作指令, 使进程能继续执行下去。

### 6.4.3 解除页面的写入保护

```
void un_wp_page (unsigned long * table_entry);
```

如果该页面存在并且在 1MB 以上 (内核代码地址空间以外), 直接在 `FLAG` 上添加 `W` 并刷新 `TLB`; 否则申请一个新页面, 并复制原页面的内容到新页面 (Copy On Write)。

**练习** `exp5/mm/page.s` 中包括页异常中断处理过程 (注意与 BIOS 中断调用区分), 主要分两种情况处理。

一. 由缺页引起的页异常中断, 通过调用 `memory.c` 中的 `do_no_page (error_code, address)` 函数来处理;

二. 由页写保护引起的页异常, 此时调用页写保护处理函数 `do_wp_page (error_code, address)` 进行处理。`error_code` 和 `address` 是函数参数, `er-`

ror\_code（出错码）是由 CPU 自动产生并压入堆栈的，出现异常时访问的线性地址 address 是从控制寄存器 CR2 中取得的。

了解缺页异常的触发方式，在 mm/mm\_test.c 中的 mmtest\_main 函数中编写触发缺页异常的代码，简述你引发缺页异常的原理并展示关键代码。

代码编写完毕后运行 NEUOS，便会看到界面中提示的 page\_fault，如图 13 所示。由于未处理缺页，这里会无限次触发中断。

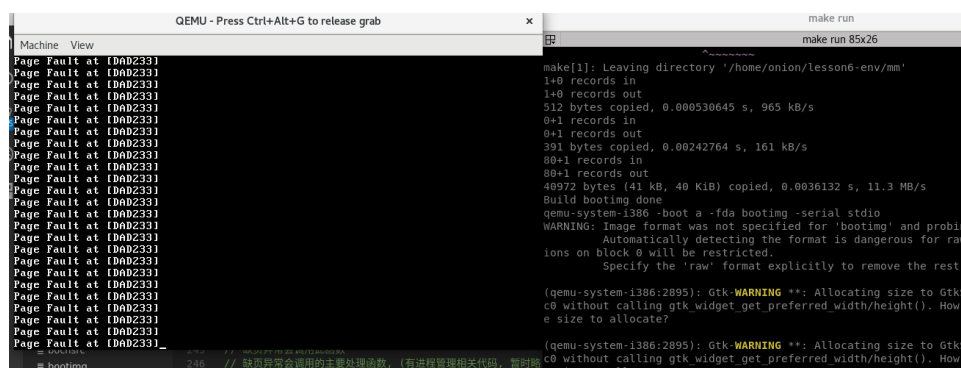
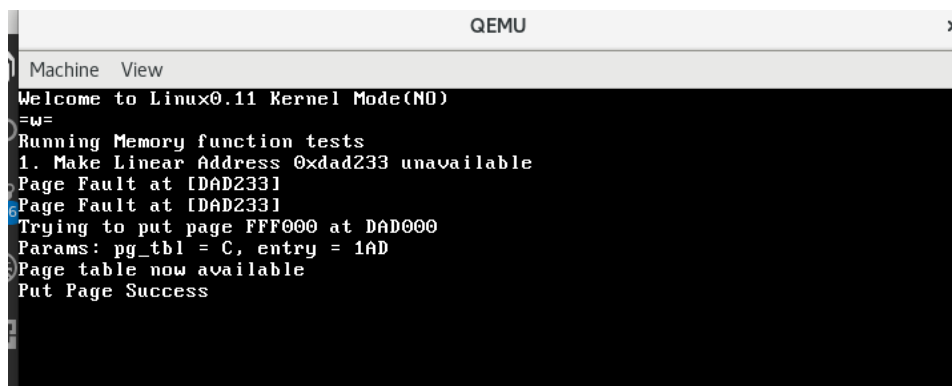


图 13: 缺页异常时的无限中断

**练习** 进入实验环境目录下 mm/memory.c 文件，找到 do\_no\_page, do\_wp\_page, un\_wp\_page 函数，其中，do\_no\_page 为缺页处理函数，该函数在发生缺页异常时会调用 get\_free\_page 函数申请一页物理内存，接下来将该 page 映射到指定的线性地址 address 处，若操作成功则返回，否则释放内存页，并调用 oom 函数报错。（此处暂不考虑进程管理以及页面共享，故代码与 Linux 0.11 内核有一定区别）

请尝试完成 do\_no\_page 函数，之后运行 NEUOS，便可以看到缺页异常提示以及申请的新的页信息，如图 14 所示。



```
QEMU
Machine View
Welcome to Linux0.11 Kernel Mode(NO)
=w=
Running Memory function tests
1. Make Linear Address 0xdad233 unavailable
Page Fault at [DAD2331]
Page Fault at [DAD2331]
Trying to put page FFF000 at DAD000
Params: pg_tbl = C, entry = 1AD
Page table now available
Put Page Success
```

图 14: 内存函数测试

**练习** 读懂 exp5/mm/memory.c 中的 do\_wp\_page 函数调用 un\_wp\_page 时传递的 table\_entry 表达式，并描述出具体含义。其中，table\_entry 是 un\_wp\_page 函数所需的参数。

**练习** 根据实验四所学串口知识及相关代码，在 exp5/kernel/serial.c 实现空缺的函数。在 exp5/include/linux/kernel.h 中这些函数已有声明，Makefile 中的 OBJS 也已补充。

将 exp5/mm/memory.c 源文件中 do\_no\_page 函数中的 printk(“Page Fault at...” 修改为 s\_printk(“Page Fault at...”

编译后使用 QEMU 运行 NEUOS 后即可在终端中看到串口输出的缺页异常信息，如图 15 所示。

接下来，在 QEMU 的 log 中找到中断发生时的寄存器信息。

提示：qemu-system-i386 -d 命令用于输出 log 信息，要输出中断的 log 信息，尝试使用 qemu-system-i386 -d help 命令找到输出中断信息的命令。然后，补充 exp5/Makefile 中的 run 命令，使用 make run 运行 NEUOS，如图 16 所示。请描述你是如何使用 QEMU 查看中断信息的，并展示运行截图。

```
Machine View
Welcome to Linux0.11 Kernel Mode(M0)
Running Memory function tests
1. Make Linear Address 0xdad233 unavailable
Page Fault at [0ad233]
Trying to put page 77f000 at 0ad000
Params: pg_tbl = C, entry = 1ad
Page table now available
Put Page Success

make run
make run 85x26
make[1]: Leaving directory '/home/onion/lesson6-env/mm'
140 records in
140 records out
512 bytes copied, 6.7795e-05 s, 7.6 MB/s
0+1 records in
0+1 records out
321 bytes copied, 0.000649252 s, 602 KB/s
00+1 records in
00+1 records out
40972 bytes (41 KB, 40 KiB) copied, 0.00202393 s, 14.0 MB/s
Build booting done.
qemu-system-i386 -boot a -fda bootimg -serial stdio
WARNING: Image format was not specified for 'bootimg' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
[DEBUG]Page Fault at [0ad233]
(qemu-system-i386:1627): Gtk-WARNING **: Allocating size to GtkScrollbar 0x7fecfad5c0 without calling gtk_widget_get_preferred_width/height(). How does the code know
```

图 15: 缺页异常信息

```
onion@onion-VM: ~/lesson6-env 124x2
onion@onion-VM: ~/lesson6-env master *
$ 20134025 vim log 137x31
check_exception old: 0xffffffff new 0xe
0: v=0e e=0000 i=0 cpl=0 IP=0008:00007d6d pc=00007d6d SP=0010:0000bef0 CR2=00dad233
EAX=00dad233 EBX=00000003 ECX=000b8000 EDX=000006b4
ESI=00000000 EDI=00000ffc EBP=0000bf18 ESP=0000bef0
EIP=00007d6d EFL=00000406 [D---P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 00ffffff 00c09300 DPL=0 DS [-WA]
CS =0008 00000000 007ffffff 00c09a00 DPL=0 CS32 [-R-]
SS =0010 00000000 00ffffff 00c09300 DPL=0 DS [-WA]
DS =0010 00000000 00ffffff 00c09300 DPL=0 DS [-WA]
FS =0010 00000000 00ffffff 00c09300 DPL=0 DS [-WA]
GS =0010 00000000 00ffffff 00c09300 DPL=0 DS [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT= 00005cb8 000007ff
IDT= 000054b8 000007ff
CR0=80000013 CR2=00dad233 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0fff DR7=00000400
CCS=000006b4 CCD=000046b4 CCO=ADDL
EFER=0000000000000000
```

图 16: QEMU 中断日志



## 7 实验六 中断与系统调用

### 7.1 实验目的

1. 加深对设备管理基本原理的认识，了解键盘中断、扫描码等概念。
2. 掌握 Linux 使用系统调用的基本原理。

本实验基于 Linux 0.11 源码。

### 7.2 中断处理

中断处理主要涉及两个代码文件：asm.s 和 traps.c 文件。

中断信号通常可以分为两类：**硬件中断**和**软件中断**（异常）。每个中断由 0-255 之间的一个数字来表示。软件中断是由 CPU 执行指令时探测到异常情况而引起的，通常可分为**故障**（fault）和**陷阱**（trap）两类。

在进程将控制权交给中断处理程序之前，CPU 会首先将至少 12 字节的信息压入中断处理程序的堆栈中。

由于有些异常引起中断时，CPU 内部会产生一个出错代码压入堆栈（异常中断 int 8 和 int 10-14），而其他中断并不带有这个出错代码（例如除零出错和边界检查出错）。因此对中断的处理，需要根据是否携带出错码分别处理，但处理流程是相同的。

1. 所有寄存器入栈。
2. 出错代码入栈。无出错代码时，使用 0。
3. 中断返回地址入栈。
4. 所有段寄存器置为内核代码段的选择符值。
5. 调用相关 C 处理函数。
6. 弹出入栈的出错码和后来入栈的中断返回地址。
7. 弹出所有入栈寄存器。

## 8. 中断返回。

其中，调用的 C 函数在 kernel/traps.c 中实现。压入堆栈的出错代码和中断返回地址是用作 C 函数的参数。

## 7.3 系统调用

Linux 应用程序调用内核的功能是通过中断调用 int 0x80 进行的，寄存器 eax 中放调用号。因此该中断调用被称为系统调用。实现系统调用的文件包括 system\_call.s、fork.c、signal.c、sys.c 和 exit.c 文件，涉及时钟中断、出错停机、进程调度等系统调用。

通常名称以 'sys\_' 开头的系统调用函数都是相应系统调用需要调用的处理函数，以汇编语言实现或 C 语言实现。而名称以 'do\_' 开头的函数，可能是系统调用处理过程中通用的函数，也可能是某个系统调用专用的。

**练习** exp6/kernel/signal.c 文件的 do\_signal 函数是系统调用中断处理程序中的信号处理程序，它将信号的处理句柄插入到用户程序堆栈中，并在系统调用结束返回后立刻执行信号句柄程序，然后继续执行用户程序。

程序将用户调用系统调用的代码指针 eip 指向信号处理句柄时，使用的语句是 "\*( &eip ) = sa\_handler;"。

这条语句难道不等价于 "eip = sa\_handler;"? 请尝试解释原因，注意函数内的变量声明。

## 7.4 键盘驱动

exp6/kernel/chr\_drv/kb.S 是一个键盘驱动程序，主要包括键盘中断处理程序。kb.S 的 key\_table 标号后的代码是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码处理子程序。

例如按下 F1-F12 按键后，kb.S 的 func 标号后的汇编指令将被执行，如下所示。

```

pushl %eax
pushl %ecx
pushl %edx
call __show__stat
pop %edx
pop %ecx
pop %eax
...

```

可知，这段汇编调用了显示各任务状态的函数 (exp7/kernel/sched.c)。

**练习** 参照 Linux v0.11 中已有的系统调用，尝试添加一个系统调用 sys\_ver。结合键盘中断，使用户在按下 F12 时，系统屏幕打印出“NEUOS exp6”，或是你编写的具有其他功能的函数。

## 7.5 fork 系统调用

Linux 所有进程都是进程 0 的子进程。fork() 系统调用用于创建子进程。exp6/kernel/fork.c 是 exp6/kernel/system\_call.s 的辅助处理函数集，给出了 sys\_fork() 系统调用使用的两个 C 语言函数 find\_empty\_process() 和 copy\_process()。包括进程内存区域验证与内存分配函数 verify\_area()。

**练习** 在 exp6/kernel/fork.c 中，copy\_process 的参数有 17 个，其中一个参数并没有在函数体中被使用，它对应了**堆栈**中的什么内容？请简要说明原因。

你需要熟悉函数调用时堆栈是如何被使用的，该函数由 system\_call.s 中的 system\_call 标号后的汇编指令调用。

## 8 实验七 进程控制

### 8.1 实验目的

1. 掌握 Linux 0.11 的进程调度策略。

2. 了解进程控制块的概念与内容。
3. 了解 fork、wait、exit 等系统调用的实现。

## 8.2 任务数据结构

在阅读前，建议先打开 `exp7/include/linux/sched.h`，通过代码注释了解其大致构成。

内核程序通过进程表对进程进行管理，每个进程在进程表中占有一项。

Linux 中，进程表项是一个 `task_struct` 任务结构指针（定义在头文件 `exp7/include/linux/sched.h`）。它被称为**进程控制块**（Process Control Block）或进程描述符（Process Descriptor）。其中保存着用于控制和管理进程的所有信息。

当一个进程在执行时，CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的**上下文**（context）。当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能恢复到切换时的状态。Linux 中，当前进程上下文均保存在进程的任务数据结构中。发生中断时，内核就在被中断进程的上下文中，在内核态下执行中断服务例程。

**任务状态段**（Task State Segment）是一个系统段（x86 架构的一部分，不是完全依靠软件实现），是保存任务信息的数据结构，包括寄存器状态、I/O 端口权限、内层堆栈指针、之前的 TSS 链接等信息。它在 Linux 中以结构体 `tss_struct`（定义在 `exp7/include/linux/sched.h` 中）描述。任务状态段被内核用于进程切换。

## 8.3 进程运行状态

进程状态保存在 `task_struct` 的 `long` 型变量 `state` 中。进程状态的宏定义在 `exp7/include/linux/sched.h` 的头部。

**就绪状态/运行状态** `TASK_RUNNING`：进程可在内核态运行，也可以在用户态运行。图中的用户运行态、内核运行态和就绪态都处于 `TASK_RUNNING` 状态。

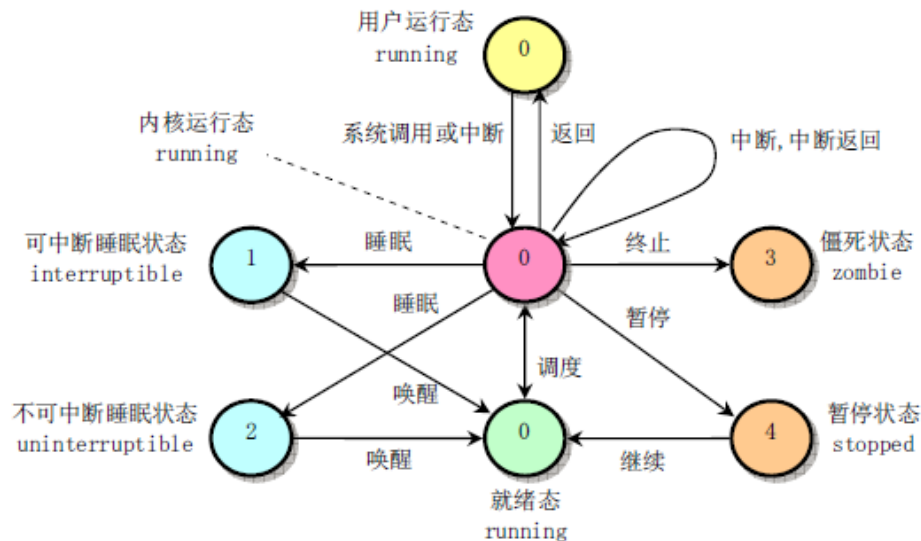


图 17: 进程状态及转换关系

**可中断睡眠状态** `TASK_INTERRUPTIBLE`: 系统不会调度该进程执行。当系统产生了中断或者释放了进程正在等待的资源, 或者进程收到一个信号, 都可以唤醒进程, 转换到就绪状态。

**不可中断睡眠状态** `TASK_UNINTERRUPTIBLE`: 与可中断睡眠状态类似。区别是处于该状态的进程只有被 `wake_up` 函数 (定义在 `exp7/kernel/sched.c`) 明确唤醒时才能转换到就绪状态。

**僵死状态** `TASK_ZOMBIE`: 进程已停止运行, 但其父进程还未询问其状态时, 进程僵死。

**暂停状态** `TASK_STOPPED`: 进程收到信号 `SIGSTOP`、`SIGTSTP`、`SIGTTIN` 或 `SIGTTOU` 时会进入暂停状态。

## 8.4 进程初始化

系统启动后, 当 `init/main.c` 将初始化操作完成后, 程序将自身移动到任务 0 (进程 0) 中运行, 并使用 `fork` 调用首次创建出进程 1。在进程 1 中程序将继续进行应用环境的初始化并执行 `shell` 程序。而原进程 0 则会在系

统空闲时被调度执行，此时任务 0 仅执行 pause 系统调用，并会再调用调度函数。

“移动到任务 0 中执行”由 exp7/include/asm/system.h 中的宏 move\_to\_user\_mode 完成。它将 main.c 程序执行流从内核态 (特权级 0) 移动到用户态 (特权级 3) 的任务 0 中继续执行。在移动到任务 0 的过程中，宏 move\_to\_user\_mode 使用了中断返回指令造成特权级改变，其详细过程此处省略。

## 8.5 创建新进程

Linux 系统中创建新进程使用 fork 系统调用 (exp7/kernel/fork.c)。所有进程都是复制进程 0 得到的，都是进程 0 的子进程。

系统创建新进程的步骤：

1. 在任务数组 (sched.h 中定义的 struct task\_struct \*task[NR\_TASKS]) 中找出一个尚未被任何进程使用的空项。如果系统已有 64 个进程在运行，则 fork 系统调用会因此出错返回。
2. 为新建进程在主内存区申请一页内存，存放其任务数据结构信息，并复制当前进程任务数据结构中的所有内容作为新进程任务数据结构的模版。为了防止尚未处理完成的新建进程被调度，新进程被置为不可中断的睡眠状态 TASK\_UNINTERRUPTIBLE。
3. 对复制的任务数据结构进行修改。将当前进程设置为新进程的父进程，清除信号位图并复位新进程各统计值。设置新进程初始运行时间片值为 15 个系统滴答数 (150ms)。根据当前进程设置 TSS 中各寄存器的值，若有协处理器还需要设置 TSS 中协处理器的信息。
4. 设置新任务的代码段和数据段的基址与限长，复制当前进程内存分页管理的页表。
5. 将父进程打开的文件的打开次数增 1。
6. 在 GDT 中设置新任务的 TSS 和 LDT 描述符项。
7. 将新任务设置为可运行状态 TASK\_RUNNING，返回新进程号。

## 8.6 进程切换

执行实际进程切换的任务由 `switch_to` 宏定义 (`exp7/include/linux/sched.h`) 的一段汇编代码完成。

进程切换的步骤：

1. `switch_to` 检查要切换到的进程是否为当前进程。如果是，直接退出；否则执行 2。
2. 把内核全局变量 `current` 置为新任务的指针，长跳转到新任务的任务状态段 TSS 组成的地址处，造成 CPU 执行任务切换操作。
3. CPU 把进程切换前所有寄存器的状态保存到当前任务寄存器 TR 中 TSS 段选择符所指向的当前进程任务数据结构的 `tss` 结构中。
4. 把新任务状态段选择符所指向的新任务数据结构的 `tss` 结构中的寄存器信息恢复到 CPU 中，系统正式开始运行新切换的任务。

## 8.7 终止进程

当一个进程结束或被终止运行，内核需要释放该进程所占用的资源，包括其打开的文件、申请的内存。

当一个用户程序调用 `exit` 系统调用时，会执行内核函数 `do_exit`（定义在 `exp7/kernel/exit.c` 中），其工作流程如下：

1. 释放进程代码段和数据段占用的内存页面，关闭进程打开着的所有文件，对进程使用的当前工作目录、根目录和运行程序的 `i` 节点（文件系统的相关实验中会介绍）进行同步操作。
2. 如果进程有子进程，则使 `init` 进程成为其所有子进程的父进程。
3. 如果进程是一个会话头进程并且有控制终端，则释放控制终端，并向属于该会话的所有进程发送挂断信号 `SIGHUP`，这通常会终止该会话的所有进程。
4. 把进程状态置为僵死状态 `TASK_ZOMBIE`。

5. 向其原父进程发送 SIGCHLD 信号，通知其某个子进程已终止。
6. 调用调度函数执行其他进程。

进程被终止时之初，其任务数据结构仍然保留，因为其父进程还需要使用其中的信息。

在子进程执行期间，父进程通常使用 `wait` 或 `waitpid` 函数（同样在 `exit.c` 中）等待其某个子进程终止。当等待的子进程被终止并处于僵死状态时，父进程会把子进程运行所使用的时间累加到自身中。最终释放已终止子进程任务数据结构所占用的内存页面，并置空子进程在任务数组中占用的指针项。

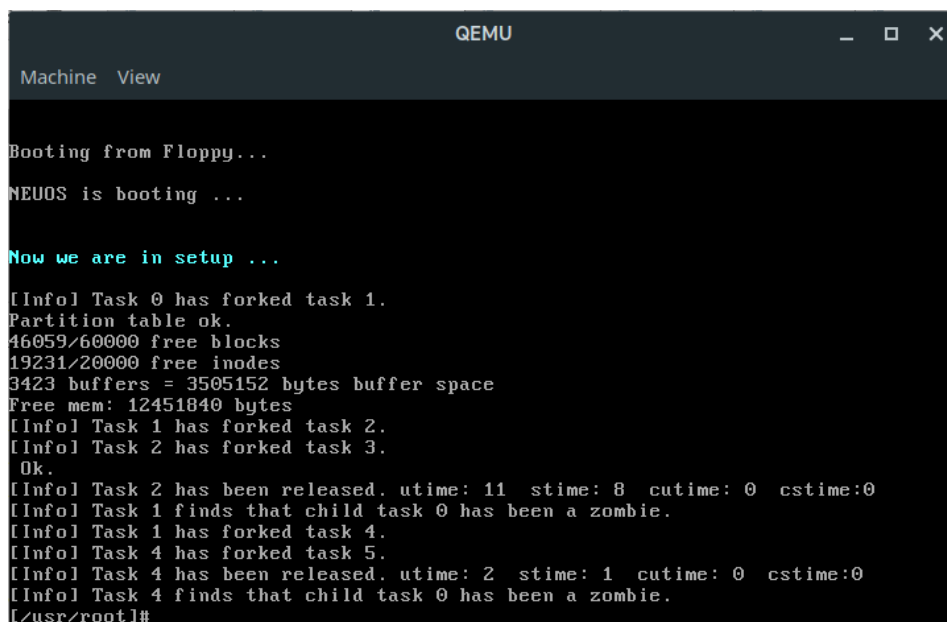
**练习** 阅读 `exp7/kernel` 目录中的 `fork.c` 与 `exit.c`，使用 `printk` 在终端上打印三类进程活动信息。实现效果如图 18 所示。请简述你如何实现进程活动信息的打印，并展示运行截图。

1. 当一个进程 `fork` 另一个进程时，输出两进程的进程号。
2. 当一个进程从任务数组 `task[]` 中被释放时，输出其进程号、用户态运行时间、系统态运行时间、子进程用户态运行时间、子进程系统态运行时间。
3. 一个进程通过 `wait` 等待其子进程，当子进程停止或僵死时输出两进程的进程号。

**练习** 打开 `exp7/kernel/sched.c`，`schedule(void)` 是进程调度函数。请描述该函数实现了哪种经典的进程调度策略，你认为使用这种策略有什么优点或不足？

**练习** 请将上述进程调度策略修改为随机调度，或你认为更优的调度算法，并描述这种新的调度策略。改写后，重新构建系统并运行，通过命令测试系统是否正常工作。



A screenshot of a QEMU terminal window. The window has a title bar with 'QEMU' and standard window controls. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal output shows the boot process of NEUOS, starting with 'Booting from Floppy...', followed by 'NEUOS is booting ...'. A green prompt 'Now we are in setup ...' appears. Subsequent lines show system statistics: 'Partition table ok.', '46059/60000 free blocks', '19231/20000 free inodes', and '3423 buffers = 3505152 bytes buffer space'. It then reports 'Free mem: 12451840 bytes'. The output continues with task management logs: '[Info] Task 0 has forked task 1.', '[Info] Task 1 has forked task 2.', '[Info] Task 2 has forked task 3.', '[Info] Task 2 has been released. utime: 11 stime: 8 cutime: 0 cstime: 0', '[Info] Task 1 finds that child task 0 has been a zombie.', '[Info] Task 1 has forked task 4.', '[Info] Task 4 has forked task 5.', '[Info] Task 4 has been released. utime: 2 stime: 1 cutime: 0 cstime: 0', and '[Info] Task 4 finds that child task 0 has been a zombie.'. The prompt at the bottom is '[usr/root]# \_'.

```
Machine View

Booting from Floppy...
NEUOS is booting ...

Now we are in setup ...

[Info] Task 0 has forked task 1.
Partition table ok.
46059/60000 free blocks
19231/20000 free inodes
3423 buffers = 3505152 bytes buffer space
Free mem: 12451840 bytes
[Info] Task 1 has forked task 2.
[Info] Task 2 has forked task 3.
Ok.
[Info] Task 2 has been released. utime: 11 stime: 8 cutime: 0 cstime: 0
[Info] Task 1 finds that child task 0 has been a zombie.
[Info] Task 1 has forked task 4.
[Info] Task 4 has forked task 5.
[Info] Task 4 has been released. utime: 2 stime: 1 cutime: 0 cstime: 0
[Info] Task 4 finds that child task 0 has been a zombie.
[usr/root]# _
```

图 18: 打印进程活动信息

## 9 实验八 文件系统

### 9.1 实验目的

1. 了解 MINIX 文件系统的发展。
2. 掌握虚拟文件系统的概念。
3. 掌握高速缓冲区相关的数据结构与算法。
4. 了解内核对竞争条件的处理。

### 9.2 文件系统布局概览

Linux 0.11 内核的文件系统参照了 MINIX 文件系统 1.0 版。

MINIX 文件系统与标准 UNIX 的文件系统基本相同，由 6 个部分组成。

**引导块**是计算机加电启动时可由 ROM BIOS 自动读入的执行代码和数据。但并非所有盘都用作引导设备，对于不用于引导的盘片，这一盘块中可

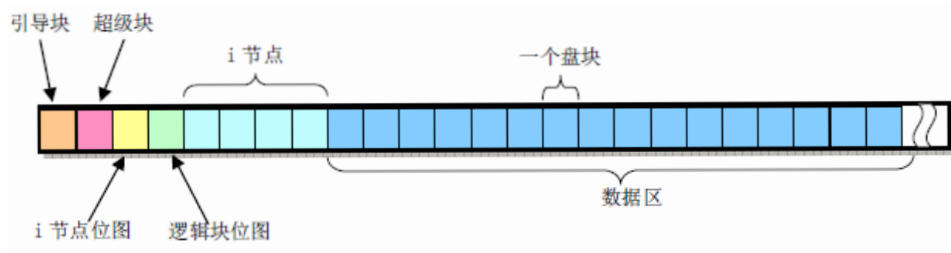


图 19: 建有 MINIX 文件系统的一个 360KB 软盘中文件系统各部分的布局示意图

以不含代码。但任何盘片必须含有引导块，以保持 MINIX 文件系统格式的统一。

**超级块**用于存放盘设备上文件系统结构的信息，并说明各部分的大小。

**i 节点位图**用于说明 i 节点是否被使用，每个比特位代表一个 i 节点。

**逻辑块位图**用于描述盘上的每个数据盘块的使用情况，每个比特位代表盘上数据区中的一个数据盘块。

盘上的 i 节点部分存放着文件系统中文件（或目录）的索引节点，每个文件（或目录）都有一个 i 节点。每个 i 节点结构中存放着对应文件的相关信息。

文件中的数据是放在磁盘块的数据区中的，而一个文件名则通过对应的 i 节点与这些数据磁盘块相联系。

### 9.3 文件的分类

类 UNIX 操作系统中的文件通常分类 6 类。在 Linux 的 Shell 下执行“ls -l”命令，可从文件状态信息中查看文件的类型。

1. **正规文件**是一类文件系统对其不作解释的文件，包含任意长度的字节流。源程序文件、二进制执行文件、文档以及脚本文件都是正规文件。
2. **目录**（‘d’）在 UNIX 文件系统中也是一种文件，但文件系统管理会对其内容进行解释，使人们看到有哪些文件包含在一个目录中，以及它们是如何组织在一起构成一个分层次的文件系统的。

3. **符号连接**（‘s’）用于使用一个不同的文件名来引用另一个文件。符号连接可以跨越文件系统，把一个文件名连接到另一个文件系统中的另一个文件上。
4. **命名管道**（‘p’）文件是系统创建有名管道时建立的文件。可用于无关进程之间的通信。
5. **字符设备**（‘c’）文件用于访问字符设备，例如 tty 终端、内存设备以及网络设备。
6. **块设备**（‘b’）文件用于访问像硬盘、软盘等设备。

## 9.4 高速缓存

CPU 中的高速缓存（cache）是用于减少处理器访问内存所需平均时间的部件，其容量远小于内存，但速度却可以接近处理器频率。

Linux 内核实现高速缓存的程序是 Linux-0.11/fs/buffer.c。文件系统中其他程序通过指定需要访问的设备号和数据逻辑块号来调用它的块读写函数。

## 9.5 数据结构定义与函数声明

neu-os 文件系统的数据结构定义与函数声明在 neu-os/include/linux/fs.h 中。

如表 8 是 fs.h 中定义的所有结构体。

以下是用于实现文件系统的重要函数，同样声明在 fs.h 中。

- `int bmap (struct m_inode *inode, int block);` 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号。
- `void iput (struct m_inode *inode);` 从设备读取指定节点号的一个 i 节点。
- `struct m_inode *iget (int dev, int nr);` 从 i 节点表中获取一个空闲 i 节点项。

结构体	表 8: fs.h 中的结构体 用途
struct buffer_head	缓冲区头数据结构
struct d_node	磁盘中的索引节点 (i 节点) 数据结构
struct m_node	内存中的 i 节点数据结构
struct file	文件结构, 用于在文件句柄与 i 节点之间建立关系
struct super_block	内存中的磁盘超级块结构
struct d_super_block	磁盘超级块结构
struct dir_entry	文件目录项结构

**练习** Linux 0.11 内核使用了 MINIX 文件系统 1.0, 请查阅相关资料, 指出 MINIX 文件系统的 1.0 和 2.0 版本之间的主要区别, 改进的主要作用是什么?

随着 Linux 的发展, 它是如何支持除 MINIX 外的多文件系统的? 并简述你对虚拟文件系统的认识。

**练习** 打开 exp8/fs/buffer.c, 在 hash 函数与 insert\_into\_queues 函数中有至少 3 处错误, 请解释你所找到的错误的原因以及修正方案, 保证系统能正常运行。

提示: 缓冲块 hash 解决冲突的方式是开放地址法还是链地址法? bh 和 free\_list 是何种类型的链表?

**练习** 分析内核是如何解决竞争条件问题的。内核中许多函数使用了 cti() 与 sti() 函数, 例如 exp8/fs 目录中的 buffer.c、inode.c、super.c, init/main 等源文件。内核是如何定义这两个函数的, 两函数的作用是什么? 如果缺少两函数, 可能有哪些问题?

exp8/fs/buffer.c 的 getblk 函数中, tmp 是空闲链表中首个空闲缓冲区, do\_while 循环中为何要判断其 b\_count 是否为 0?

注意, 可注释掉 cti() 或 sti() 函数, 运行程序观察内核是否报错。