



操作系统专题实验 实验指导书

基于 Linux 内核

实验内容: Linux 内核裁剪、系统调用设计与篡改、系统动态模块编程、字符设备驱动、类 Ext2 文件系统设计、最小内核操作系统设计、内核空间的内存映射

电信学院计算机系
2015.3

目 录

实验一 编译内核	- 1 -
1.1 安装 Linux 操作系统	- 1 -
1.1.1 实验目的	- 1 -
1.1.2 实验内容	- 1 -
1.1.3 实验步骤	- 1 -
1.2 编译 Linux 内核	- 6 -
1.2.1 实验目的	- 6 -
1.2.2 实验内容	- 6 -
1.2.3 实验原理	- 6 -
1.2.4 实验步骤	- 7 -
实验二 系统调用	- 14 -
2.1 实验目的	- 14 -
2.2 实验内容	- 14 -
2.3 实验原理	- 14 -
2.4 实验步骤	- 16 -
实验三 模块编程	- 19 -
3.1 模块编程	- 19 -
3.1.1 实验目的	- 19 -
3.1.2 实验内容	- 19 -
3.1.3 实验原理	- 19 -
3.1.4 实验步骤	- 20 -
3.2 proc 编程	- 23 -
3.2.1 实验目的	- 23 -
3.2.2 实验内容	- 23 -
3.2.3 实验原理	- 23 -
3.2.4 实验步骤	- 25 -
实验四 篡改系统调用	- 42 -
4.1 实验目的	- 42 -

4.2 实验内容	- 42 -
4.3 实验原理	- 42 -
4.4 实验步骤	- 46 -
实验五 文件系统	- 53 -
5.1 类 ext2 文件系统	- 53 -
5.1.1 实验目的	- 53 -
5.1.2 实验内容	- 53 -
5.1.3 Ext2 文件系统分析	- 53 -
5.1.4 设计原理	- 61 -
5.1.5 实验步骤	- 67 -
5.2 添加一个文件系统	- 99 -
5.2.1 实验目的	- 99 -
5.2.2 实验内容	- 99 -
5.2.3 实验原理	- 99 -
5.2.4 实验步骤	- 99 -
实验六 字符设备驱动程序	- 107 -
6.1 一个简单的字符驱动程序	- 107 -
6.1.1 实验目的	- 107 -
6.1.2 实验内容	- 107 -
6.1.3 实验原理	- 107 -
6.1.4 实验步骤	- 109 -
6.2 键盘指示灯驱动	- 119 -
6.2.1 实验目的	- 119 -
6.2.2 实验内容	- 119 -
6.2.3 实验原理	- 119 -
6.2.4 实验步骤	- 121 -
实验七 一个最小操作系统的实现	- 126 -
7.1 引导启动程序	- 126 -
7.1.1 实验目的	- 126 -
7.1.2 实验内容	- 126 -

7.1.3 实验原理	- 126 -
7.1.4 实验步骤	- 126 -
7.2 实现一个简单的多任务操作系统	- 129 -
7.2.1 实验目的	- 129 -
7.2.2 实验内容	- 129 -
7.2.3 实验原理	- 129 -
7.2.4 实验步骤	- 130 -
实验八 内核空间的内存映射实验	- 141 -
8.1 实验目的	- 141 -
8.2 实验内容	- 141 -
8.3 实验原理	- 141 -
8.4 实验步骤	- 151 -

实验一 编译内核

1.1 安装 Linux 操作系统

1.1.1 实验目的

熟悉 Linux 操作系统的安装步骤，建立实验环境。

1.1.2 实验内容

安装 VMwrae 虚拟机，并在其上安装 Linux 操作系统，或在 PC 上已有其它操作系统（如 Windows）的基础上安装 Linux，构成双系统。

1.1.3 实验步骤

目前的虚拟机软件主要有好多种，如 VMware Workstation 和 Virtual PC，其中 VMware Workstation 对宿主机的仿真性很高，尤其是在显示上比其他虚拟机软件要好。本实验选用 VMware Workstation 10.0，展示如何在虚拟机下安装 Linux。VMware 虚拟机的安装只要按向导提示操作即可。

Linux 有很多发行版，如 Red Hat 、RedFlag 、Ubuntu 、Fedora 等。本实验采用 Ubuntu12.10，其原始内核版本为 3.8。安装时，默认 VMware Workstation 10.0 已经装好。

Linux 虚拟机安装的主要步骤如下：

步骤一：新建虚拟机，选择典型（如图 1.1 所示）。



图 1.1 新建虚拟机

步骤二：稍后再将虚拟机光驱设为 Linux 安装光盘镜像文件（如图 1.2 所示）。

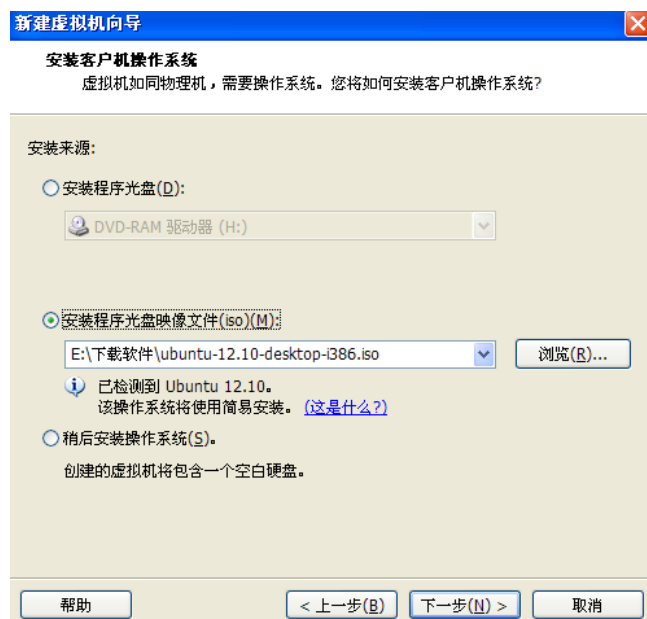


图 1.2 设置安装光盘镜像

步骤三：选择磁盘容量大小。建议如图所示进行设置（如图 1.3 所示）。

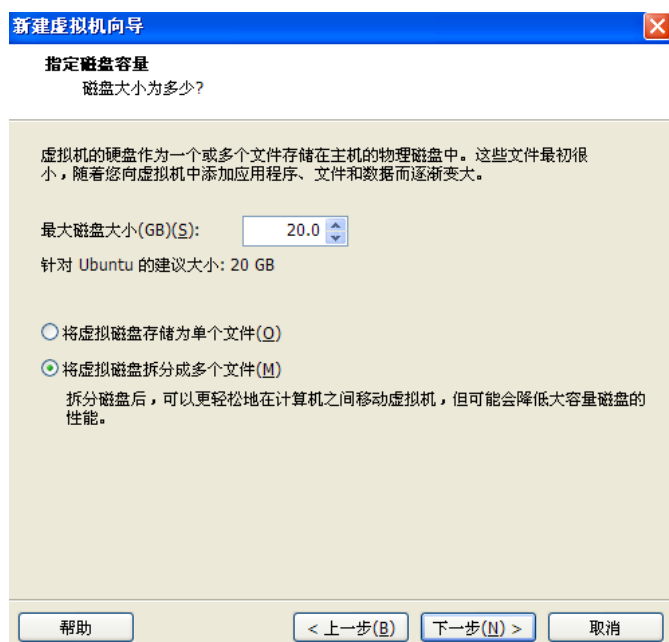


图 1.3 确定磁盘容量

步骤四：完成名称设置，准备好创建虚拟机（如图 1.4 和图 1.5 所示）。

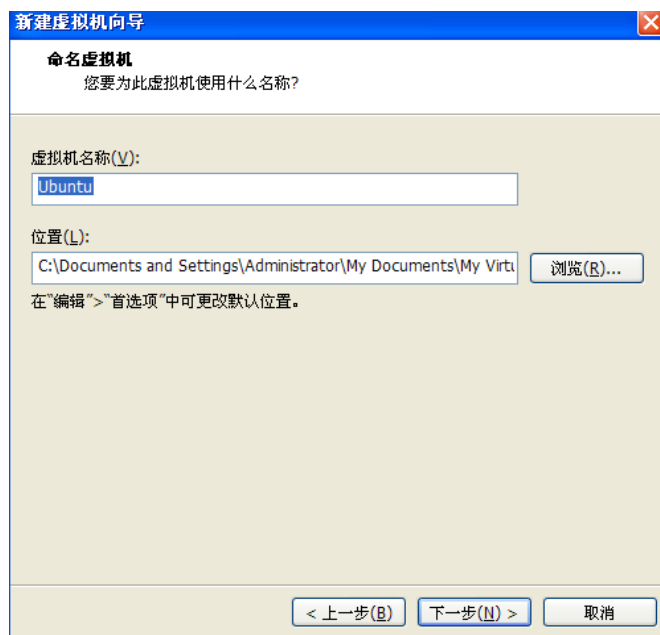


图 1.4 设置虚拟机名称



图 1.5 准备好创建

步骤五：启动虚拟机按照提示安装（如图 1.6 所示）。



图 1.6 安装选项

步骤六：按照提示完成后续安装。考虑诸多软件安装过程需联网进行，为加快安装进程，建议此过程断开互连网络（如图 1.7 所示）。



图 1.7 安装过程

安装完毕就可以从虚拟机启动新系统。

步骤七：设置密码。

- 1) 进入 Ubuntu Linux（桌面环境）。
- 2) 首次使用 Ubuntu 设置 root 密码。在终端输入 `sudo passwd root`，然后输入两遍新的 root 密码即可。
- 3) 获取 Linux root 权限，终端输入 `su root`，按照提示输入 root 密码后回车即可。

注意：为了实现主操作系统（以XP为例）与虚拟机Ubuntu Linux之间信息共享，共享文件夹设置步骤如下：

- 1) 点击虚拟机菜单下“虚拟机”，依次选择“设置”/“选项”/“共享文件夹”/“添加”。
(如图 1.8 所示)

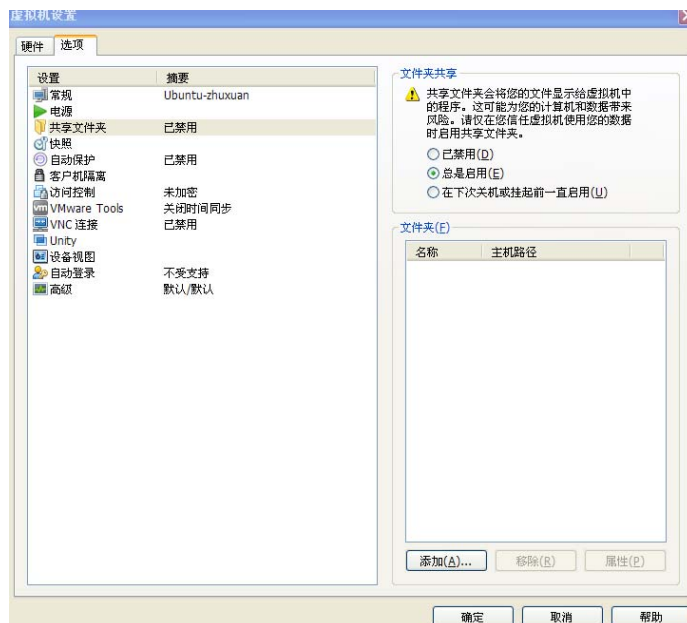


图 1.8 设置过程选择

- 2) 按向导进行设置 (如图 1.9 所示)。

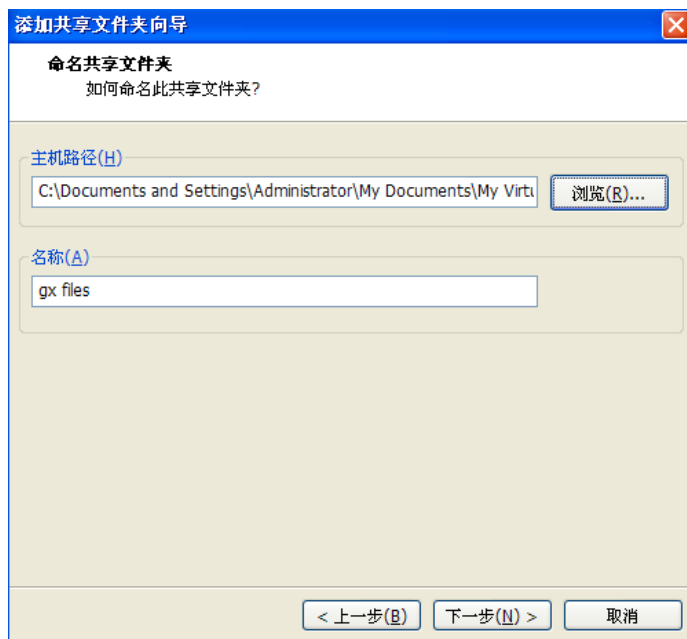


图 1.9 设置向导

- 3) 在终端中进入 “/mnt/hgfs”, 对共享文件设置进行验证。图中 “gx files”为自己创建共享文件夹 (如图1.10所示)。

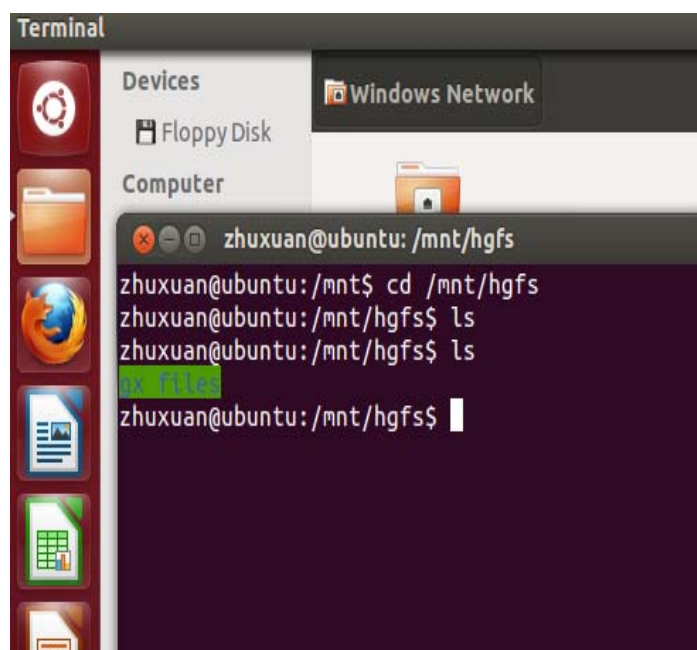


图 1.10 共享验证

1.2 编译 Linux 内核

1.2.1 实验目的

通过编译 Linux 内核，掌握对内核功能的裁剪，以构造满足自身环境需要的 Linux 内核。

1.2.2 实验内容

下载新的内核源代码，配置之后再行编译链接，形成自己所需的内核映像文件，并予以使用。

1.2.3 实验原理

内核是一个操作系统的核心。它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux作为一个自由软件，在广大爱好者的支持下，内核版本不断更新。新的内核修订了旧内核的bug，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统度身定制一个更高效、更稳定的内核，就需要重新编译内核。

为了正确、合理地设置内核编译配置选项，从而只编译系统需要的功能代码，一般主要有下面四个考虑：

- 1) 自己定制编译的内核运行更快（具有更少的代码）；
- 2) 系统将拥有更多的内存（内核部分将不会被交换到虚拟内存中）；
- 3) 不需要的功能编译进入内核可能会增加被系统攻击者利用的漏洞；

4) 将某种功能编译为模块方式会比编译到内核内的方式速度要慢一些。

1.2.4 实验步骤

目前内核版本主要分为 2.6.x 版本与 3.x 版本, 这两个版本的内核编译过程稍有不同。本实验在 Ubuntu12.04 上以内核 3.8.13 版本为例给出编译过程。

步骤一：从 <https://www.kernel.org/pub/linux/kernel/v3.x/> 中下载 3.8.13 版本的内核源代码解压至 /usr/src 目录。

```
#cd /usr/src  
  
#tar -zxvf linux-3.8.13.tar.bz2  
  
#cd linux-3.8.13
```

步骤二：配置前准备。

#make mrproper 该命令确保源代码目录下没有不正确的.o 文件以及文件的互相依赖, 如果要多次编译内核最好执行此步骤。

步骤三：启动内核配置程序。

在编译Linux内核之前, 可以根据需要选择配置选项, 但必须告诉编译程序Linux内核需要哪些功能。内核配置中, 某项选择Y表示把该项选择进内核, 选择M则表示把该项编译成模块, 选择N则表示不选择进内核。Linux提供了多种内核配置命令, 可以任选其一:

```
#make config    基于文本的最为传统的配置界面, 不推荐使用。  
  
#make menuconfig  基于文本选单的配置界面, 字符终端下推荐使用。  
  
#make xconfig    基于图形模式的配置界面, Xwindow 下推荐使用, 但需要安装QT。  
  
#make oldconfig  如果只想在原来内核配置基础上修改一些小地方, 会省去不少麻烦。
```

这里我们用 make menuconfig 命令。在 make menuconfig 下, 其中的[*]表示 Y, [M]表示 M, [空白]表示 N。该命令是基于 ncurses 的菜单选择模式, 需要 ncurses 库的支持, ubuntu 中默认没有安装, 可使用以下命令安装:

```
# apt-get install libncurses5-dev
```

如果提示未找到库, Ubuntu12.04 下可以将软件源更改为 Main server(System->Administration->Software Sources)。

步骤四：配置内核各选项。

下面是 Linux 内核编译配置选项的一些简介。图 1.11 是执行 make menuconfig 的显示结果, 此结果实际上是读取或者生成一个.config 文件, 该文件即为配置内核的结果。

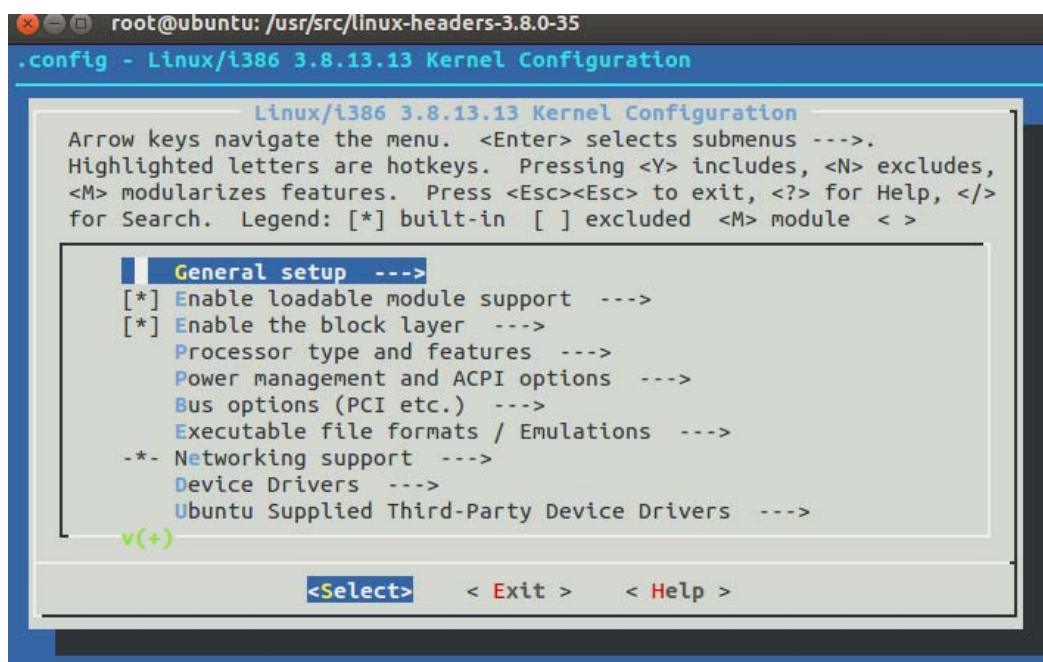


图 1.11 menuconfig 配置菜单

General setup 常规设置。

Enable loadable module support 可加载模块支持。

Enable the block layer 块设备支持，使用硬盘/USB/SCSI 设备者必选。

Processor type and features 中央处理器(CPU)类型及特性。

Power management options 电源管理选项。

Bus options (PCI etc.) 总线选项。

Executable file formats / Emulations 可执行文件格式。

Networking 网络支持选项。

Device Drivers 设备驱动选项。

Firmware Drivers 固件驱动选项。

File systems 各种类型的文件系统支持选项。

.....

第一次做内核编译实验，在不太清楚哪些需要选择，哪些不需要选择的情况下，最好保持默认状态，以后随着对内核编译选项的熟悉，可以根据需要去掉一些不必编译的部分，以精简内核。本实验中，以下四个部分最好选中编译进内核或动态模块。

1) 在 General setup -> Local version -append to kernel release 中为本地内核的版本号添加一个标志，本实验中添加-2，如图 1.12 所示，这样新内核的版本号为 3.8.13-

2. 在多次编译同一版本的内核源代码时，此选项方便查看新的本地内核版本。

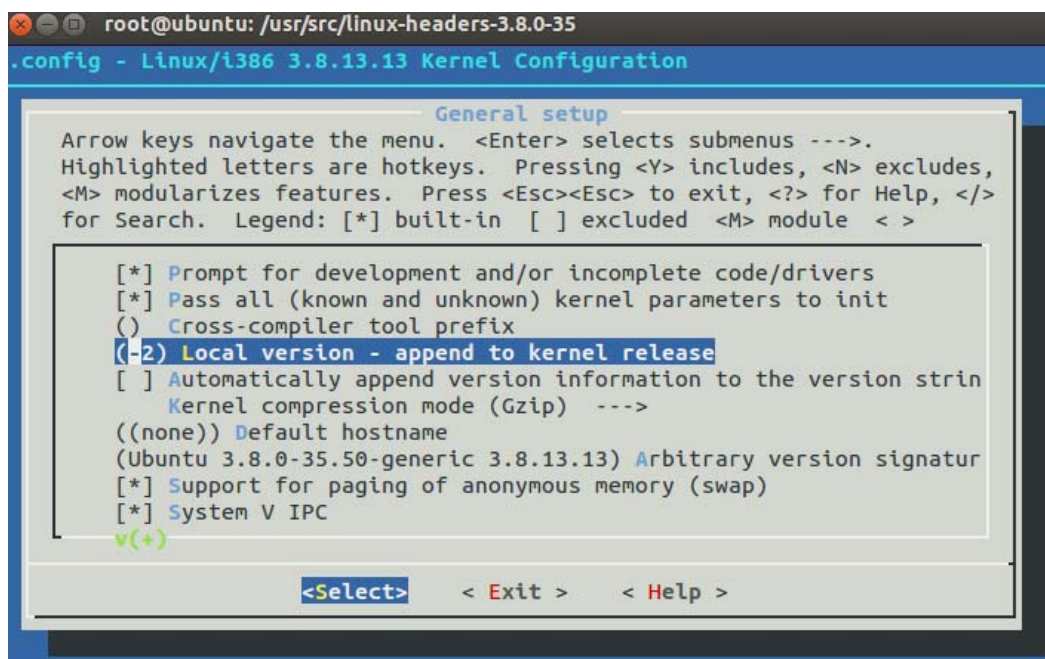


图 1.12 添加自定义版本号

2) 鉴于之后要进行模块加载的实验，确保在 Enable loadable module support 中模块加载支持的选项被选中，如图 1.13 所示。

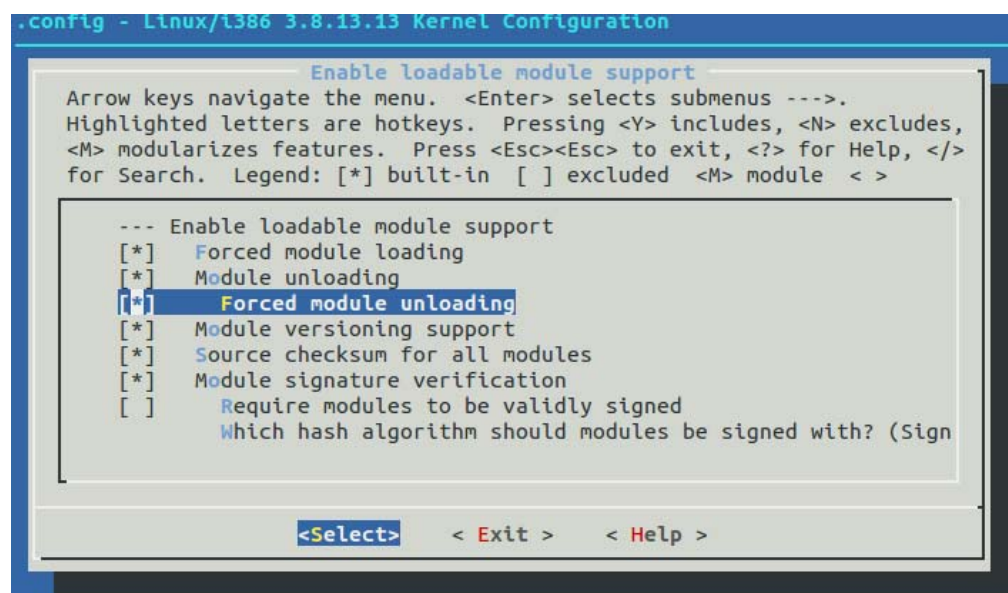


图 1.13 动态加载模块支持

3) 如果新建虚拟机时磁盘类型选择为 SCSI 类型，则在 Device Drivers -> SCSI device support 中选中 SCSI disk support, 如图 1.14 所示。IDE 类型的磁盘也可以选中此项，但不是必须的。

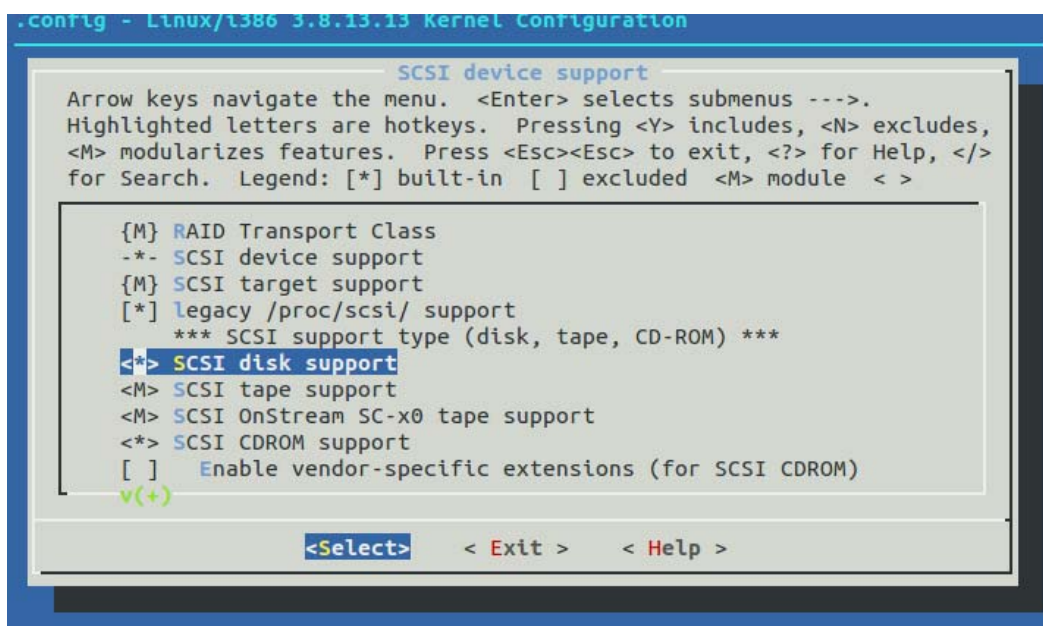


图 1.14 SCSI 磁盘类型支持

4) 在文件系统 File systems 中, 确保选中对 ext3 文件系统的支持, 如图 1.15 所示。
在 File systems -> DOS/FAT/NT Filesystems 中确保选中对 NTFS 文件系统的支持选项, 如图 1.16 所示。

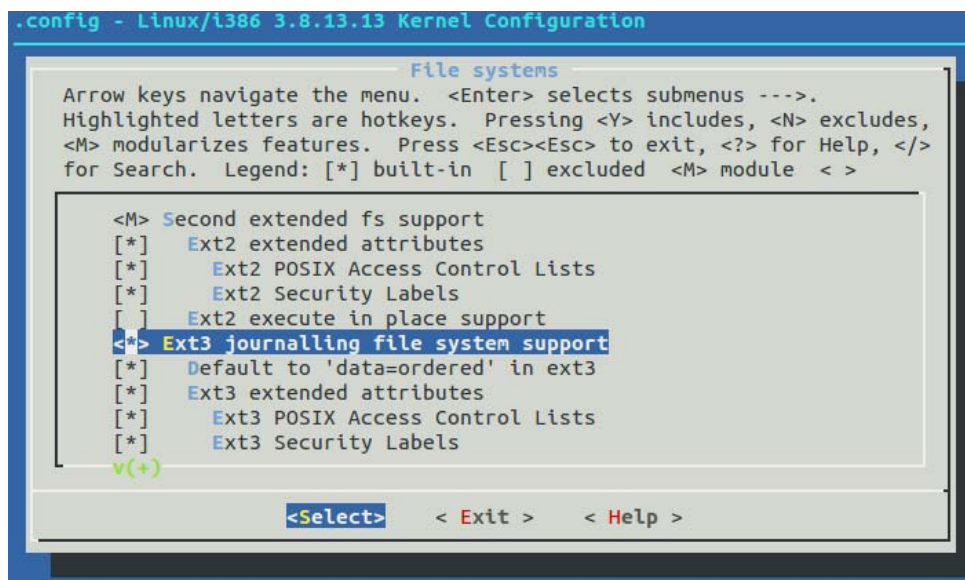


图 1.15 Ext3 文件系统支持

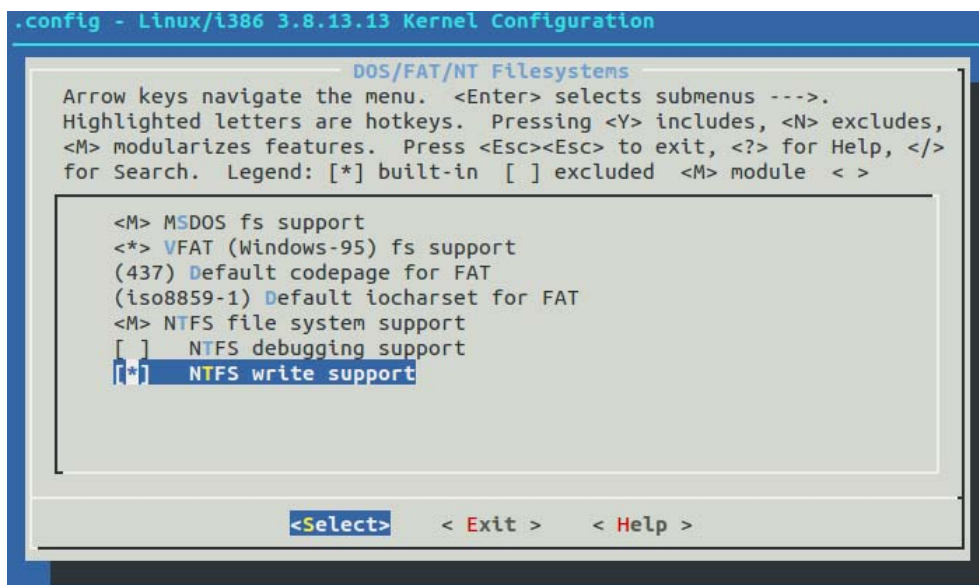


图 1.16 NTFS 文件系统支持

选择完成后，保存设置生成.config 文件。

步骤五：编译内核。

#make clean 完成删除前面步骤留下的文件，以避免出现一些错误。

#make bzImage 内核编译成功后，会在linux-3.8.13-2/arch/x86/boot 目录中生成一个新内核的映像文件bzImage。

步骤六：编译安装可加载模块。

#make modules 设置了可加载模块 所以用此命令生成内核模块。

#make modules_install 安装生成的内核模块。

编译成功后，系统会在/lib/modules 目录下生成一个3.8.13-2的子目录，其下存放着新内核的所有可加载模块。

（注：上面几个命令也可以合成一个make bzImage modules modules_install，经过2小时左右编译，如果不报错终止，那么就成功了；或者用make 命令来替换make bzImage与make modules。）

步骤七：安装编译生成的内核文件。

#make install 安装编译生成的内核文件。

步骤八：修改开机引导文件。

首先在/boot/目录下查看是否生成了 initrd.img-3.8.13-2 文件，如果没有的话，则使用mkinitramfs 命令生成镜像文件。

mkinitramfs -o initrd.img-3.8.13-2 3.8.13-2

最后一个参数 3.8.13-2 内核版本，即/lib/modules/目录下相应的文件夹名字。这个参数必须指定，否则使用正在运行的系统版本。

在当前目录下生成 initrd 镜像文件，然后把它移动到 boot 目录下。

```
#cp initrd.img-3.8.13-2 /boot/ initrd.img-3.8.13-2
```

接着更新 grub 引导菜单，使用如下命令：

```
#update-grub
```

最后修改/boot/grub/grub.cfg 文件，Ubuntu 下可以使用 gedit 或 vi 编辑器。在某些版本上，此步骤不是必须的，因为在执行上述 update-grub 命令后系统会更新 grub.cfg 文件，使其包含新的镜像文件，但此处最好还是打开一下 grub.cfg 文件，检查一下新内核的镜像文件设置是否正确。

```
#gedit /boot/grub/ grub.cfg
```

查看是否有如下内容，并修改 timeout 的值。其中 timeout 时间设置为 10 秒。

```
if [ "${recordfail}" = 1 ]; then
    set timeout=-1
else
    set timeout=10
fi
.....
menuentry 'Ubuntu, Linux 3.8.13-2' --class ubuntu --class gnu-linux --class gnu
--class os {
    set gfxpayload=$linux_gfx_mode
    insmod part_msdos
    insmod ntfs
    set root='(hd0,msdos6)'
    search --no-floppy --fs-uuid --set=root 247C75217C74EEC4
    loopback loop0 /ubuntu/disks/root.disk
    set root=(loop0)
    linux /boot/vmlinuz-3.8.13-2 root=UUID=247C75217C74EEC4
    loop=/ubuntu/disks/root.disk ro quiet splash vt.handoff=7
    initrd /boot/initrd.img-3.8.13-2
}
menuentry 'Ubuntu, Linux 3.8.13-2 (恢复模式)' --class ubuntu --class gnu-linux
--class gnu --class os {
    insmod part_msdos
    insmod ntfs
    set root='(hd0,msdos6)'
    search --no-floppy --fs-uuid --set=root 247C75217C74EEC4
    loopback loop0 /ubuntu/disks/root.disk
```



```
set root=(loop0)
echo '载入 Linux 3.8.13-2 ...'
linux /boot/vmlinuz-3.8.13-2 root=UUID=247C75217C74EEC4
loop=/ubuntu/disks/root.disk ro recovery nomodeset
echo '载入初始化内存盘...'
initrd /boot/initrd.img-3.8.13-2
}
.....
```

步骤九：重启系统，开机时按 Esc 键，选择新的内核，如图 1.17 所示。

#reboot

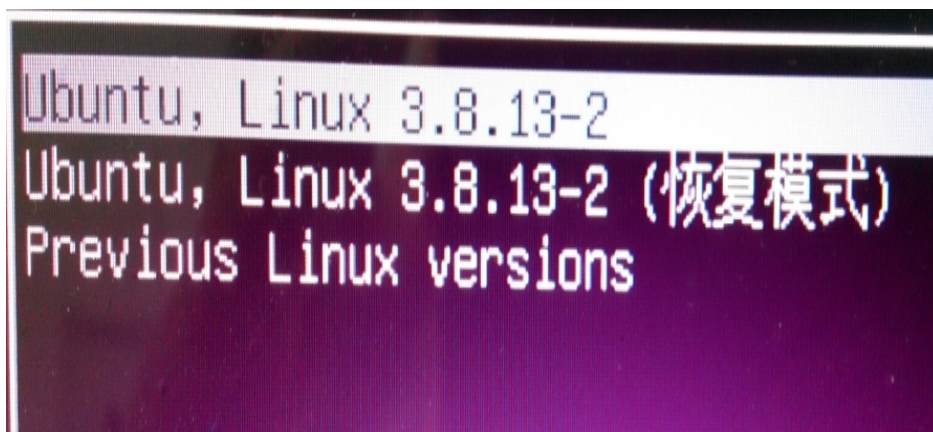


图 1.17 选择新内核

步骤十：验证内核版本。

在终端输入 `uname -r`，若显示 3.8.13-2 即证明新内核编译安装成功，且正在运行的系统使用的为新编译的内核。

实验二 系统调用

2.1 实验目的

掌握系统调用的添加过程，为以后添加更复杂的系统调用奠定基础。

2.2 实验内容

设计一个系统调用，功能为在内核日志中打印“Hello my world”（或其他功能，如输出系统所有进程的相关信息）。

2.3 实验原理

系统调用是应用程序和操作系统内核之间的功能接口，其主要目的是使用户可以使用操作系统提供的有关设备管理、输入/输出系统、文件系统和进程控制、通信以及存储管理等方面的功能，而不必了解系统程序的内部结构和有关硬件细节，从而起到减轻用户负担和保护系统以及提高资源利用率的作用。

在Linux系统中，系统调用是作为一种异常类型实现的，它将执行相应的机器代码指令来产生异常信号，产生中断或异常的重要效果是系统自动将用户态切换为核心态来对它进行处理。这就是说，执行系统调用异常指令时，自动地将系统由用户态切换为核心态，并安排异常处理程序的执行。Linux用来实现系统调用异常的实际指令是：

```
int $0x80
```

这一指令使用中断/异常向量号128（即16进制的80）将控制权转移给内核。为达到在使用系统调用时不必用机器指令编程，在标准的C语言库中为每一系统调用提供了一段短的子程序，完成机器代码的编程工作。事实上，机器代码段非常简短。它所要做的工作只是将送给系统调用的参数加载到CPU寄存器中，接着执行int \$0x80指令，然后运行系统调用。该返回值将送入CPU一个寄存器中，标准库的子程序取得这一返回值，并将它送回用户程序。为使系统调用的执行成为一项简单的任务，Linux提供了一组预处理宏，这些宏指令取一定的参数，然后扩展为调用指定的系统调用的函数，这样就可以在用户程序中执行这一系统调用。

为了更好地理解系统调用的细节过程，下面将以内核0.12版本中一个汇编的系统调用例子（在本实验环境下也可以正常运行）加以说明。其中第一段程序是系统调用原语句宏扩展后的write原函数，第二段程序是实际的一个调用write的汇编程序，对于汇编程序的知识，

读者可查阅相关书籍，这里不再作为重点介绍。

4号系统调用宏扩展函数write:

```
int write(int fd, char *buf, int n)
{
    long __res;
    __asm__ volatile(
        "int $0x80"
        : "=a" (__res)
        : "0" (__NR_write), "b" ((long) (fd)), "c" ((long) (buf)),
        "d" ((long) (n)));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

一个系统调用的实际例子:

```
.text
_entry:
    movl $4, %eax           #系统调用号，写操作
    movl $1, %ebx           #写操作调用的参数，是文件描述符。数值1对应stdout
    movl $message, %ecx     #参数，缓冲区指针
    movl $12, %ebx          #参数，写数据长度值
    int $0x80
    movl $1, %eax           #系统调用号，退出程序
    int $0x80
message:
    .ascii "Hello World\n"  #欲写的数据
```

其中使用了两个系统调用：4—写文件操作sys_write()和1—退出程序sys_exit()。写文件系统调用所执行的C函数声明为sys_write(int fd, char *buf, int len)。它带有三个

参数，在系统调用之前这3个参数分别被存放在寄存器ebx、ecx和edx中。该程序编译和执行的步骤如下：

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
Hello World
[/usr/root]#
```

2.4 实验步骤

Linux 2.6.x 版本的内核和 3.x 版本的模块机制有些不同，添加一个系统调用的方法也略有不同，本实验主要基于 3.8.13 版本的内核来展示如何设计一个自己的系统调用。

Linux 下设计实现一个系统调用的步骤如下：

步骤一：设计系统调用源代码并将其加入 Linux 内核源代码中。

在 /usr/src/Linux-3.8.13/kernel 目录下，打开文件 sys.c，在头文件中添加：`#include<linux/linkage.h>`(若无此头文件时);在文件的最后部位增加系统调用函数：

```
asmlinkage int sys_my_syscall(int n)
{
    printk("Hello my world"); //或者其他功能
    return n;
}
```

步骤二：修改与系统调用表相关的文件，以便根据系统调用号在系统调用表中确定其对应的系统调用名。

编辑系统调用入口文件。打开 /usr/src/linux-3.8.13/arch/x86/systemcalls 目录下 syscall_32.tbl 文件（如果系统为 64 位的，则需修改 syscall_64.tbl），在最后添加系统调用符号名：

sys_my_syscall 和调用号 350（格式参照上文件），以便以系统调用号来确定新的系统调用的入口地址。

步骤三：修改头文件 syscalls.h，定义新的系统调用所对应的系统调用号。

对于新设计的系统调用，还要求定义一个新的系统调用号，此时需要修改 syscalls.h。

文件路径为：/usr/src/linux-3.8.13/arch/x86/include/asm/syscalls.h

在末尾适当位置添加系统调用函数声明：

```
asmlinkage int sys_my_syscall(char *)
```

用户程序在调用某一系统调用时，使用了系统调用号，系统可根据该调用号在系统调用表中确定该系统调用代码的起始地址，进一步可根据该起始地址执行其相应的系统调用程序。其中 `asmlinkage` 表示强制从堆栈中获取参数，即设置系统调用函数的参数保存在堆栈中。

步骤四：编译内核。

编译过程同实验一，此处不再赘述。可以给内核版本号添加一个标志。

步骤五：重启系统，进入新编译的内核。

步骤六：利用用户程序测试新设计的系统调用。

编写用户程序 `syscall_test.c`。

```
#include<stdio.h>
#include<linux/unistd.h>
int main()
{
    int ret;
    ret=syscall(350,2); //350 号系统调用是新设计的系统调用
    printf("%d\n",ret);
    if(2==ret)
        printf("the first syscall is success!\n");
    return 0;
}
```

编译运行测试程序。

```
gcc -o syscall_test syscall_test.c
```

```
./ syscall_test
```

运行成功在终端应显示：

```
2
```

```
the first syscall is success!
```

而对于新系统调用执行的输出结果 “Hello my world”，可以通过内核日志查看命令 `dmesg` 进行观察。

输入命令：`#dmesg -c`

显示内核进程输出结果。（需在执行 `./syscall_test` 之后）若显示 “Hello my world” 则证明已经成功运行了新设计的系统调用。

实验三 模块编程

3.1 模块编程

3.1.1 实验目的

掌握 Linux 下内核模块的实现机制，熟练运用动态模块进行系统程序编程。

3.1.2 实验内容

为内核设计一个简单的动态模块，并利用用户程序进行测试，之后再予以卸载。

3.1.3 实验原理

模块是在内核空间运行的程序，实际上是一种目标文件，不能单独运行但其代码可在运行时链接到系统中作为内核的一部分运行或卸载。Linux内核模块是一个编译好的、**具有特定格式的独立目标文件**，用户可通过系统提供的一组与模块相关的命令将模块加载进内核，**当内核模块被加载后，它有如下特点：**

- 1) 与内核一起运行在相同的内核态和内核地址空间；
- 2) 运行时与内核具有同样的特权级；
- 3) 可方便地访问内核中的各种数据结构。

此外，内核模块还可以很容易地被移出内核，当用户不再需要某模块功能时，可以将它从内核卸载以节省系统主存开销。

一个典型的内核模块应包含如下几个部分：

(1) 头文件声明。其中module.h和init.h是必不可少的。module.h包含加载模块时需要的函数和符号定义；init.h中包含模块初始化和清理函数的定义。如果在加载时允许用户传递参数，模块中还应包含moduleparam.h头文件。

(2) 模块许可声明。从内核2.4.10版本开始，模块必须通过MODULE_LICENSE宏声明此模块的许可证，否则在加载此模块时会显示“kernel tainted”（内核被污染）的警告信息。从linux/module.h文件中可看到，被内核接受的许可证有GPL、GPL v2、GPL and additional rights、Dual BSD/GPL、Dual MPL/GPL、Dual MIT/GPL和Proprietary。

(3) 初始化和清理函数声明。内核模块必须调用宏module_init和module_exit去注册初始化和清理函数。初始化和清理函数必须在宏module_init和module_exit使用前定义，否则会出现编译错误。这两个函数配对使用，例如当初始化函数申请了一个资源，那么清理函数就应释放这个资源，使得模块不留下任何副作用。除了模块初始化函数和清理函数，还可

以根据需要设计编写其它函数。

本实验基于3.8版本的内核。设计一个简单的内核模块，该模块的功能是在被加载进内核时向系统日志中写入“hello,my module was loaded!”；在模块被卸载时向系统日志写入“goodbye,unloading my module.”。

3.1.4 实验步骤

设计一个内核模块，首先要设计模块代码，对其进行编译形成模块程序（.ko 文件，2.6 以前内核版本为.o 文件），然后就可以加载和卸载模块了。当模块加载后，就可以如同使用系统调用一样调用模块中的相关函数了。

内核模块是否加载成功可以通过lsmod命令观察。模块加载成功后，使用lsmod命令输出的模块列表中会有相应的模块项目（模块程序名）；如果模块初始化函数中通过printk函数向内核日志写入信息的话，此时还可以使用命令dmesg 查看内核日志，观察其变化情况；或者查看/proc/modules 文件，检查文件内容是否包含相应的模块项目。同理，在卸载时也同时观察三者的变化。

步骤一：设计内核模块程序。并将其放置于一个没有 Makefile 文件的目录，如/home/module 中。

内核模块代码如下（mymodules.c）：

```
#include <linux/init.h> /*必须要包含的头文件*/
#include <linux/kernel.h>
#include <linux/module.h> /*必须要包含的头文件*/
static int mymodule_init(void) //模块初始化函数
{
    printk("hello,my module was loaded! \n"); /*输出信息到内核日志*/
    return 0;
}
static void mymodule_exit(void) //模块清理函数
{
    printk("goodbye,unloading my module.\n"); /*输出信息到内核日志*/
}
module_init(mymodule_init); //注册初始化函数
module_exit(mymodule_exit); //注册清理函数
MODULE_LICENSE("GPL"); //模块许可声明
```

步骤二：在模块源程序的相同目录下创建 Makefile 文件。

为了便于对其上述设备驱动程序 `globalvar.c`（模块源程序）进行编译，建立 `Makefile` 文件，代码如下：

```
ifneq ($(KERNELRELEASE),)
obj-m := mymodules.o          #obj-m 指编译成外部模块

else
KERNELDIR := /lib/modules/$(shell uname -r)/build #定义一个变量，指向内核目录
PWD := $(shell pwd)

modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules #编译内核模块
endif
```

注：1) “modules:”下一行需用“tab”键作为开始。

2) `Makefile` 的详细编写规则参考 `linux-source/Documentation/kbuild/makefiles.txt`

步骤三：编译内核模块程序。

获取 root 权限后切换至内核动态模块程序所在的目录执行命令：

```
#make
```

执行 `make` 命令后可以看到在此目录下产生的文件列表如图 3.1 所示。其中 `.ko` 文件就是编译产生的模块文件（2.6 以前的内核版本则是 `.o` 文件）。

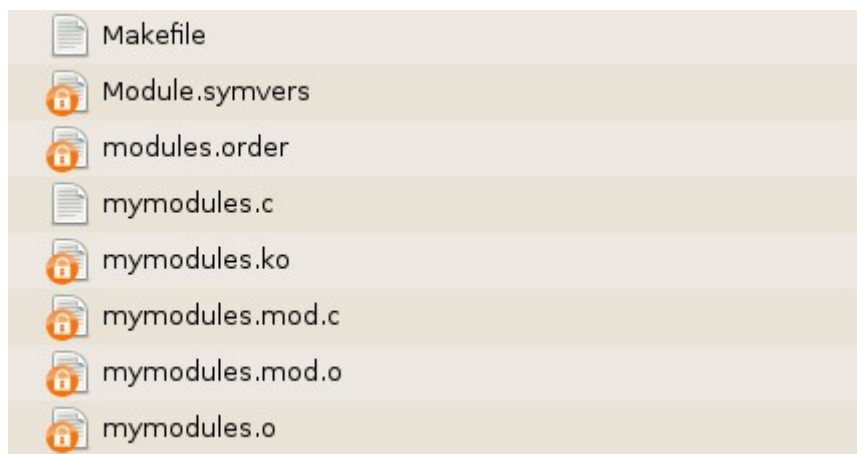


图 3.1 编译模块程序后产生的文件列表

步骤四：加载内核动态模块。

当需要使用内核模块中的相关函数时，可加载相应的模块，执行如下命令：

```
#insmod ./mymodules.ko
```

此时可以使用 `lsmod` 命令来查看系统内的模块列表，以观察新设计的模块加载情况(如图 3.2 所示)。

```
#lsmod
```

Module	Size	Used by
mymodules	2304	0
ipv6	246372	22
af_packet	17152	2
rfcomm	33936	2

```
.....
```

图 3.2 加载模块后系统模块列表

其中 mymodules 即为新加载的内核动态模块。

或者查看内核日志信息，执行：

```
#dmesg -c
```

此时可以看到内核日志信息：hello,my module was loaded!

步骤五：卸载内核动态模块。

模块使用完毕后，为了释放模块文件所占用的系统资源，可以对模块予以卸载。执行的命令是：

```
#rmmod mymodules
```

此时可以使用 lsmod 命令查看系统内的模块列表，以观察新设计的模块卸载情况(如图 3.3 所示)。

```
#lsmod
```

Module	Size	Used by
ipv6	246372	22
af_packet	17152	2
rfcomm	33936	2
l2cap	20352	13 rfcomm

```
.....
```

图 3.3 卸载模块后系统模块列表

此时可以看到系统中已经没有 mymodules 模块了。

或者查看内核日志信息，执行：

```
#dmesg -c
```

此时可以看到内核日志信息：goodbye,unloading my module.

3.2 proc 编程

3.2.1 实验目的

了解和熟悉 proc 文件系统的编程方法。

3.2.2 实验内容

编写一个简单的内核动态模块，该模块至少包括两个函数，一个模块初始化函数，当模块加载到内核时被调用；另一个函数是模块清理函数，当内核模块卸载时被调用，然后调用宏module_init和module_exit去注册初始化和清理函数。

可利用Linux的模块，在proc文件系统的目录下创建子目录及文件，并对其文件进行读写操作（文件格式与内容自拟，可以是学生学籍管理的基本信息，也可以是企业人员的基本情况，也可以是某班级的成绩册等）。

3.2.3 实验原理

proc 文件系统是一个虚拟的文件系统，实际上使用内存虚拟磁盘文件系统。它的虚拟化的实质是重新编写文件读写函数。也就是说将文件读写函数映像到另外的一个读写控制内存中数据的函数上面。这样，对于用户来说，它依旧使用的是文件系统的读写函数，但实际上完成的是对内存中单元的操作，从而起到虚拟化的作用。

所以对于 proc 文件系统编程创建有价值的应用程序时，需要做两方面的考虑：

首先，编写的模块需要做什么，其次，编写的应用程序需要做什么。

对于模块，需要完成的是基本的读写操作，也就是将文件系统的读写函数映像到的函数的编写。同时它需要完成的是初始化和清理功能，即初始化内存区域和释放内存区域的工作。对于应用程序应当完成的是实际的业务流程的描述。这时的应用程序应当将自己视为对普通的文件系统的读写。只是实际上这个文件系统是不可以随便读写的位于内存中的一块单元而已。最后加载模块，对应用程序进行操作，然后卸载模块，同时观察变化的发生。

实验中设计一个机房的上机登记系统。假设机房中共有 32 台计算机，学生上机来需要在前台登记，并且领取一台空闲机器的机器号；上机结束后到前台交回机器号，前台办理下机登记。前台可以查询所有的上机记录和当前的计算机使用情况。

假如设计的是实际文件系统，那么必然是由两个文件存放数据，一个是存放的当前 32 台机器的上机情况，另一个文件存放的线性记录下的当天学生上机记录。模块加载时初始化两个文件，卸载时销毁所有数据。

在 proc 文件系统的实验中，其基本模块结构如图 3-4 所示。

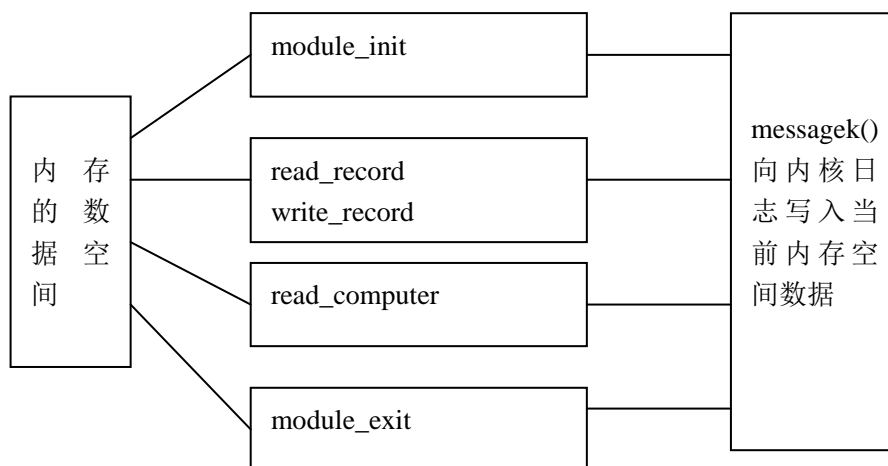


图 3-4 proc 实验的模块结构图

为了能够完成这样一个 proc 文件系统，需要介绍实验里用到的编程函数。

a) 创建 proc 新目录

```
struct proc_dir_entry* proc_mkdir(const char *name, struct proc_dir_entry
*parent);
```

b) 创建 proc 新文件

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct
proc_dir_entry *parent);
```

其中 name 为要创建的文件名，mode 为要创建的文件的属性，parent 为这个文件的父目录。

其中 name 为要创建的目录名，parent 为这个目录的父目录。

c) 读函数 read_func

```
int read_func(char *buffer, char **stat, off_t off, int count, int *peof, void
*data);
```

其中 buffer 是接收用户返回的信息，最大不超过 4K；stat 一般不使用；off 是 buffer 的偏移量；count 是用户要取的字节数；peof 的作用是读到文件尾时，把 peof 指向的位置置 1；data 是当被多个 proc 文件定义为读时，通过 data 传递参数。

d) 写函数 write_func

```
int write_func(struct file *file; const char *buffer, unsigned long count, void
*data);
```

其中 file 是该 proc 文件对应的 file 结构，一般忽略；buffer 是待写的数据所在位

置; count 是待写数据的大小; data 是当被多个 proc 文件定义为读时, 通过 data 传递参数。

以上这四个函数便是程序中主要用到的四个, 其中前两个还有相对应的移除函数 (在 module_exit 里有使用), 后两个一般要进行函数的重定向。

3.2.4 实验步骤

本实验主要编写两个部分的程序, 其一为内核态程序, 组织为动态模块的方式; 其二为用户态程序, 组织为各种用户接口。

步骤一: 编写内核态源程序 procfs.c, 示例如下:

```
/**
 *procfs.c
 *proc_module source file
 *you will make and install this module in linux
 *in xjtu
 *2014.1.13
 */
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#define SYSTEM "system"
#define RECORD "record"
#define COMPUTER "computer"
#define N 32 //total computers
#define MAX 100 // max record table size
#define LEN 11

int read_record(char *, char**, off_t, int, int *, void *);
int write_record(struct file *, const char*, unsigned long, void *);
int read_computer(char *, char**, off_t, int, int *, void *);
int write_computer(struct file *, const char*, unsigned long, void *);

struct proc_dir_entry *record, *system;
struct proc_dir_entry *computer;
```

```
int k;
int record_loop=0;
char bitmap[N]; // computer condition array
char table[MAX][LEN]; //record table

static int proc_module_init(void)
{
    int i;
    /*create file system*/
    system=proc_mkdir(SYSTEM, NULL);
    record=create_proc_entry(RECORD, 0644, system);
    computer=create_proc_entry(COMPUTER, 0644, system);
    record->read_proc=read_record;
    record->write_proc=write_record;
    computer->read_proc=read_computer;
    /*initialize bitmap and top implies no students here*/
    for(i=0;i<N;i++)
        bitmap[i]=-1;
    /*print message in kernel*/
    printk("proc module init\n");
    return 0;
}

/**
 * print message in the kernel for every action.
 * the action may be : write_record, read_record, read_computer for this module
 */
void messagek(char *action)
{
    int i;
    printk("\n action=%s \n", action); //sign for read or write action
    for(i=0;i<6;i++)
        if(table[i][0]=='c')
            printk("%c:%d:%s ", table[i][0], table[i][1], table[i]+2);
        else if(table[i][0]=='c')
```

```
        printk("%c:%d ", table[i][0], table[i][1]);
    printk("\ncomputer condition:");
    for(i=0; i<N; i++)
        printk("%d ", bitmap[i]);
    return;
}

/**
 * the new write function in our proc file system
 * write data to record proc file system
 * we map this function to pro_write in the module_init
 * we can call this function in the user space through fwrite(...)
 */
int write_record(struct file *file, const char *buffer, unsigned long count, void
*data)
{
    char str[LEN+1];
    int i;
    copy_from_user(str, buffer, count);
    str[count]='\0';
    if(count>LEN) return 0;
    k=str[1]; //computer number
    if(k>=N || k<0)
        return -1;
    if(record_loop>=MAX) record_loop=0;
    if(str[0]=='c')/*come*/
    {
        bitmap[k]=k;
        for(i=0; i<=count; i++)
            table[record_loop][i]=str[i];
        record_loop++;
    }
    }else if(str[0]=='l')/*leave*/
    {
```

```
        bitmap[str[1]]=-1;
        for(i=0;i<=count;i++)
            table[record_loop][i]=str[i];
        record_loop++;
    }
    messagek("write_record");
    return count;
}

/**
 * the new read function in our proc file system
 * read data from computer proc file system
 * we map this function to pro_read in the module_init
 * we can call this function in the user space through fread(...)
 */
int read_computer(char *page, char **start, off_t off, int count, int *eof, void
*data)
{
    int i;
    int length=0;
    for(i=0;i<count;i++)
        length+=sprintf(page+i,"%c",bitmap[off+i]);
    messagek("read_computer");
    return length;
}

/**
 * the new read function in our proc file system
 * read data from record proc file system
 * we map this function to pro_read in the module_init
 * we can call this function in the user space through fread(...)
 */
int read_record(char *page, char **start, off_t off, int count, int *eof, void
*data)
{

```



```

    int i;
    int len=0;
    char *pointer=table[0]+off;
    for(i=0;i<count;i++,pointer++)
        len+=sprintf(page+i,"%c",*pointer);
    messagek("read_record");
    return len;
}

```

```

static void proc_module_exit(void)
{
    /*remove the file system*/
    remove_proc_entry(RECORD, system);
    remove_proc_entry(COMPUTER, system);
    remove_proc_entry(SYSTEM, NULL);
    printk("proc module exit\n");
}

```

```

module_init(proc_module_init);
module_exit(proc_module_exit);

```

```

MODULE_DESCRIPTION("MY FIRST PROC TEST");
MODULE_AUTHOR("QINGWEI");
MODULE_LICENSE("GPL");

```

步骤二：编写用户态应用程序 check. c, 其示例代码如下：

```

/**
 *check. c
 *the test program fo module:procfs
 *in xjtu
 *2014. 1. 13
 */
#include<stdio.h>
#include<string.h>
#define COMPUTER "/proc/system/computer"
#define RECORD "/proc/system/record"

```

```
#define LEN 11
#define MAX 100
void come();
void leave();
void display();
void help();
int apply_computer(char *);
int release_computer(int);
int getrecord();
int getcomputer(int);
int max_count=0;

int main()
{
    char command[10];
    printf("*****\n");
    printf("** *\n");
    printf("** Hello! Welcome to our software! *\n");
    printf("** Athor: zhuxuan *\n");
    printf("** Code: 2014-1-13 *\n");
    printf("** *\n");
    printf("*****\n");
    /*-----Shell Loop-----*/
    while(1)
    {
        printf("command=># ");
        scanf("%s", command);
        if(!strcmp(command, "come"))
            come();
        else if(!strcmp(command, "leave"))
            leave();
        else if(!strcmp(command, "display"))
            display();
        else if(!strcmp(command, "exit"))
            break;
    }
}
```

```
        else if(!strcmp(command, "help"))
            help();
        else
            printf("Command not available\n");
    }
    printf("Thanks for using our software!\n");
    return 0;
}

/*****come and register a computer*****/
void come()
{
    char name[10];
    int computer_no;
    /*come loop*/
    while(1)
    {
        printf("come=>$Please input your name:");
        scanf("%s", name);
        computer_no=apply_computer(name);
        if(-1==computer_no)
            printf("come=>$ No computer available, please wait");
        else
            printf("come=>$Computer Number:%d\n", computer_no);
        printf("come=>$Press q to get out, other keys to continue:");
        scanf("%s", name);
        if(!strcmp(name, "q")) break;
    }
}

/*****leave and unregister*****/
void leave()
{
    int number;
    char buffer[32];
    int ret;
```

```
while(1)
{
    printf("leave=>$ Please input the number of your computer:");
    scanf("%d",&number);
    ret=release_computer(number);
    if(ret!=0)
        printf("wrong number or this computer not in use\n");
    else
        printf("\tcomputer No:%d is cheching out...\n",number);
    printf("leave=>$ Press q to quit,other keys to continue:");
    scanf("%s",buffer);
    if(!strcmp(buffer,"q")) break;
}
}
```

```
void display()
{
    /*display function*/
    char buffer[32];
    int number;
    int i=0;
    /**display loop***/
    while(1)
    {
        printf("[1] Records of today.\n");
        printf("[2] Records of all computers.\n");
        printf("[3] Records of computer in use.\n");
        printf("[4] Records of computer unused.\n");
        printf("display=>$");
        scanf("%d",&number);
        switch(number)
        {
            case 1:
                getrecord();
                break;
        }
    }
}
```

```
        case 2:
            for(i=0;i<32;i++)
            {
                if(getcomputer(i)==-1)
                    printf("%d:computer not use\n",i);
                else printf("%d:computer in use\n",i);
            }
            break;
        case 3:
            for(i=0;i<32;i++)
            {
                if(getcomputer(i)==-1) continue;
                else printf("%d:computer in use\n",i);
            }
            break;
        case 4:
            for(i=0;i<32;i++)
            {
                if(getcomputer(i)==-1)
                    printf("%d,UNUSED\n",i);
                else continue;
            }
            break;
        default:
            printf("Command not available\n");
    }

    printf("display=>$ Press q to quit,other keys to continue:");
    scanf("%s",buffer);
    if(!strcmp(buffer,"q")) break;
}

void help()
{
    /*help function*/
}
```

```

printf(" Thank you for using our software!!\n");
printf("-----\n");
printf("come: for student registred a computer.\n");
printf("leave: for student unregistred and checked out.\n");
printf("display: for administrator to get current information.\n");
printf("exit: for administrator to quit the program.\n");
printf("help: get information of the software.\n");
printf("-----\n");
printf(" in xjtu \n");
return;
}

/*****interaction with system module*****/
/**
 * apply a computer for student 'name'
 * return : computer No if have ;
 *          -1 if not have
 */
int apply_computer(char *name)
{
    FILE *fp;
    char buffer[32];
    int i, j;
    /*get a computer unused*/
    fp=fopen(COMPUTER, "r+w");
    fseek(fp, 0, 0);
    fread(buffer, 1, 32, fp);
    for(i=0; i<32; i++)
        if(buffer[i]==-1) break;
    fclose(fp);

    /*no computer unused, return -1*/
    if(i==32) return -1;

    /*fill in the record table*/

```

```
    buffer[0]='c';
    buffer[1]='c'; //fill in a char first
    buffer[2]='\0';
    strcat(buffer,name);
    buffer[1]=i; //replace the char
    if(max_count>=MAX)
    {
        printf("table out of buffer!\n");
        return -1;
    }
    else
    {
        fp=fopen(RECORD, "a+");
        fwrite(buffer, strlen(name)+2, 1, fp);
        fclose(fp);
        max_count++;
    }
    return i;
}

int release_computer(int number)
{
    if(number>=32) return -1;//out of bound
    FILE *fp;
    int i;
    char buffer[32];
    /**/
    fp=fopen(COMPUTER, "r+w");
    fseek(fp, 0, 0);
    fread(buffer, 1, 32, fp);
    fclose(fp);
    i=buffer[number];
    if(-1==i) return -1; // not in use

    buffer[0]='l';
```

```
    buffer[1]=i;
    buffer[2]='l' ;
    if(max_count>=MAX)
    {
        printf("table out of buffer!\n");
        return -1;
    }
    else
    {
        fp=fopen(RECORD, "a+");
        fwrite(buffer, LEN, 1, fp);
        fclose(fp);
        max_count++;
    }
    return 0; //nomal out
}

int getrecord()
{
    char str[32];
    FILE *fp;
    fp=fopen(RECORD, "r+w");
    fseek(fp, 0, 0);
    while(1)
    {
        fread(str, LEN, 1, fp);
        if(!strcmp(str+2, ""))
            break;
        else
        {
            if(str[0]=='c')
            {
                printf("%s:come in %d\n", str+2, str[1]);
            }
            if(str[0]=='l')
```



```
        {  
            printf("computer %d:leave\n",str[1]);  
        }  
    }  
}  
fclose(fp);  
return 0;  
}
```

```
int getcomputer(int number)  
{  
    int i;  
    char buffer[32];  
    FILE *fp;  
    fp=fopen(COMPUTER,"r+w");  
    fseek(fp,0,0);  
    fread(buffer,1,32,fp);  
    fclose(fp);  
    if(buffer[number]==-1) return -1;  
    return 0;  
}
```

步骤三：参考 3.1 节将内核源文件 procfs.c 编译成模块，并加载。加载后在 /proc/system/ 目录下可以看到两个文件 record 和 computer。

步骤四：将用户应用程序 check.c 编译成可执行文件，输入：

```
#gcc -o check check.c
```

步骤五：执行用户应用程序。

```
#. /check
```

步骤六：验证。

运行用户程序，观察运行效果如下图 3.5 至图 3.12 所示。

运行用户程序后，即出现了欢迎界面，如下图 3.5。

```

*****
** **
** Hello! Welcome to our software! **
** Author: zhuxuan **
** Code: 2014-1-13
** **
*****
Command=>#

```

图 3.5 程序开始运行界面

输入 display 命令, 可看到程序使用提示, come 为学生注册命令, leave 为学生离开命令, display 为查看当前上机情况, exit 为退出。如图 3.6。

```

command=># help
Thank you for using our software!!

Come: for student registred a computer.
Leave:for student unregistered and checked out.
Display: for administrator to get current information.
Exit: for administrator to quit the program.
Help:get information of the software.

Author by xjtu
command=>#

```

图 3.6 操作提示界面

输入 come 即可输入到来学生的名字, 随后机器会分配电脑给该学生。如图 3.7 所示, 共登记了三个上机的学生。

```

command=># come
come=>$Please input you name :zhuxuan
come=>$Computer Number:0
come=>$Press q to get out,other| keys to continue :a
come=>$Please input your name:chentao
come=>$ Computer Number:1
come=>$Press q to get out,other keys to continue :a
come=>$Please input your name:wangheng
come=>$ Computer Number:2
come=>$Press q to get out,other keys to continue :q
command=>#

```

图 3.7 有三个人上机操作界面

Display 命令可用于查看当前使用电脑的学生信息，如图 3.8 所示，已经有三个学生在使用电脑上机。

```
Command=># display
[1] Records of today.
[2] Records of all computers.
[3] Records of computer in use.
[4] Records of computer unused.
Display=>$1
Zhuxuan:come in 0
Chentao:come in 1
Wangheng:come in 2
Display=>$ Press q to quit ,other keys to continue:
```

图 3.8 三个人上机后查看记录表

同时，也可查看内核输出的信息，记录的为操作类型、用户信息以及当前各台电脑的使用情况。如图 3.9 所示。

```

root@zhuxuan-desktop:/home/zhuxuan/module/proc_new# dmesg -c
[2920.029224]
[2920.029228] action=read_computer
[2920.029285] computer condition: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[2920.029605] action=write_record
[2920.029608] c:0:zhuxuan
[2920.029610] computer condition: 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[2925.191561] action=read_computer
[2925.191567] c:0:zhuxuan
[2925.191569] computer condition: 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[2925.191649] action=write_record
[2925.191651] c:0:zhuxuan c:1:chentao
[2925.191654] computer condition: 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[2931.577635] action=write_record
[2931.577637] c:0:zhuxuan c:1:chentao c:2:wangheng
[2931.577641] computer condition: 0 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
root@zhuxuan-desktop:/home/zhuxuan/module/proc_new#

```

图 3.9 三人上机后通过 dmesg 查看内核日志信息

当学生下机时，就可使用 `leave` 命令完成注销，同时该离开动作也会被记录在内核中。如图 3.10 所示。

```

Command=># leave
Leave=>$ Please input the number of your computer:1
Computer No:1 is cheching out ...
Leave=>$ Press q to quit,other keys to continue;q
Command=.# display
[1] Records of today.
[2] Records of all computers.
[3] Records of computer in use.
[4] Records of computer unused.
Display=>$1
zhuxuan:come in 0
chentao:come in 1
wangheng:come in 2
computer 1 : leave
display=>$ Press q to quit,other keys to continue;q

```

图 3.10 一号机离开界面

查看学生离开后机器的使用情况，如图 3.11，可以看出，1 号机器已经不再被使用了。

```
display=>$ Press q to quit,other keys to continue:a↵
[1] Records of today.↵
[2] Records of all computers.↵
[3] Records of computer in use.↵
[4] Records of computer unused.↵
Display=>$3↵
0:computer in use↵
2:computer in use↵
display=>$ Press q to quit,other keys to continue:↵
```

图 3.11 一号机离开后查看机器使用情况

同时,在用户离开后,查看内核空间记录的信息。

```
root@zhuxuan-desktop:/home/zhuxuan/module/proc_new# dmesg -c↵  
[3349.098503] action=read_computer↵  
[3349.098536] c:0:zhuxuan c:1:chentao c:2:wangheng↵  
[3349.098566]] computer condition:0 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ↵  
[3349.098775] action=write_record↵  
[3349.098777] c:0:zhuxuan c:1:chentao c:2:wangheng l:1↵  
[3349.098804] computer condition:0 -1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 ↵  
root@zhuxuan-desktop:/home/zhuxuan/module/proc_new# ↵
```

图 3.12 一号机离开后内核日志信息

读者可以注意，即便退出程序，重新运行一次，被记录过的操作信息也依然存在，不会丢失，这是因为使用的存储数据的空间属于内核，只要模块未卸载，则操作记录会一直保留。

实验四 篡改系统调用

4.1 实验目的

掌握采用内核动态模块的方式篡改 Linux 内核中的系统调用。

4.2 实验内容

设计一个内核动态模块，在模块初始化时篡改一个已有的系统调用，使其执行一段自定义的功能，在模块退出时还原系统调用的功能。

4.3 实验原理

1) Linux系统调用概述

详见第二次实验部分。

2) 系统调用相关的数据结构

(1) 系统调用号

Linux操作系统利用系统调用号来标识系统调用，每一个系统调用对应唯一的一个系统调用号。本实验中 3.8.13 版本内核中系统调用号的定义位于：

`/usr/src/linux-3.8.13/arch/x86/include/generated/uapi/asm/unistd_32.h`

`/usr/include/i386-linux-gun/asm/unistd_32.h`

其中前者代表内核代码的头文件，后者代表标准C库的头文件（64 位版本位于 `unistd_64.h` 中），内容如下。

```
#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
#define __NR_open               5
#define __NR_close              6
.....
#define __NR_gettimeofday       78
.....
#define __NR_timerfd            322
```

```
#define __NR_eventfd      323
```

```
#define __NR_fallocate    324
```

(2) 系统调用表

Linux操作系统利用函数指针数组保存系统调用服务程序的地址，这个数组成为系统调用表（sys_call_table），该数组的首地址（也即sys_call_table的值）可以在/boot目录下的 System.map-'uname -r' 文件中找到。例如：

```
.....
c1630060 R sys_call_table
.....
```

则系统调用表的值为 0x c1630060, 不同版本的Linux内核，系统调用表的值会有不同。操作系统利用系统调用号为下标在系统调用表中找到相关的系统调用服务程序函数指针并跳转到相应的地址，实现系统调用服务程序的寻址与调用。

3.8.13 版本的内核系统调用表的定义位于：

/usr/src/linux-3.8/arch/x86/syscalls目录下的syscall_32.tbl文件中。

num	abi	name	entry point	compat entry point
0	i386	restart_syscall	sys_restart_syscall	
1	i386	exit	sys_exit	
2	i386	fork	sys_fork	stub32_fork
3	i386	read	sys_read	
4	i386	write	sys_write	
.....	
77	i386	getrusage	sys_getrusage	compat_sys_getrusage
78	i386	gettimeofday	sys_gettimeofday	compat_sys_gettimeofday
79	i386	settimeofday	sys_settimeofday	compat_sys_settimeofday
.....
349	i386	kcmp	sys_kcmp	
350	i386	finit_module	sys_finit_module	
351	i386	my_syscall	sys_my_syscall	

其中sys_之后的名字即为系统调用的服务程序函数名，在time.c文件中就有 78 号系统

调用的函数体：

```
asmlinkage long sys_gettimeofday(struct timeval __user *tv,
                                struct timezone __user *tz)
{
    if (likely(tv != NULL)) {
        struct timeval ktv;
        do_gettimeofday(&ktv);
        if (copy_to_user(tv, &ktv, sizeof(ktv)))
            return -EFAULT;
    }
    if (unlikely(tz != NULL)) {
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
    }
    return 0;
}
```

其中参数tv是保存获取时间结果的结构体，参数tz用于保存时区结果（若不使用则传入NULL即可）。

结构体timeval的定义为：

```
struct timeval {
    time_t      tv_sec;      /* seconds */
    suseconds_t tv_usec;     /* microseconds */
};
```

结构体timezone的定义为：

```
struct timezone {
    int tz_minuteswest; /* minutes west of Greenwich */
    int tz_dsttime; /* type of dst correction */
};
```

函数名前的asmlinkage，是提示编译器在获取参数时是从堆栈中提取而不是寄存器；函数do_gettimeofday的实现位于linux-3.8.13/kernel/time目录下的timekeeping.c文件中。

```
void do_gettimeofday(struct timeval *tv)
{
    struct timespec now;
```



```

getnstimeofday(&now);

tv->tv_sec = now.tv_sec;

tv->tv_usec = now.tv_nsec/1000;

}

```

(3) 控制寄存器的修改

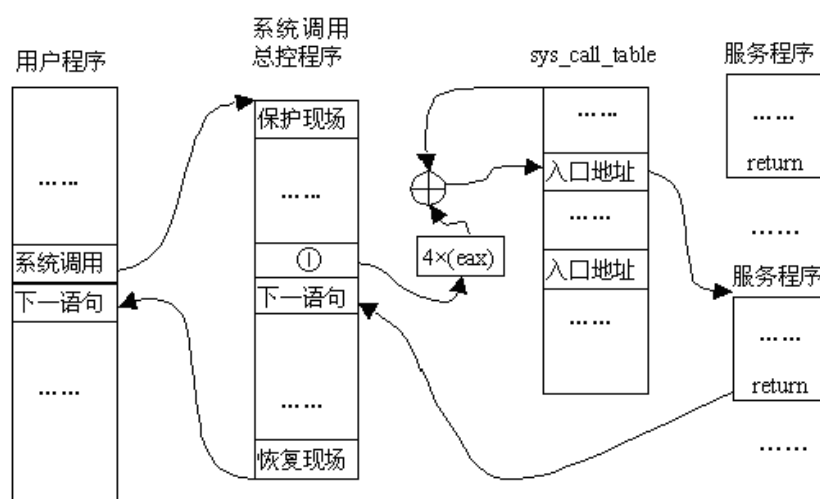
在上面符号表中查阅sys_call_table的地址时，c1630060 R sys_call_table里面的R代表的是可读属性，所以为了能顺利篡改系统调用的地址，需要提前对控制寄存器CR0 的第16位WP（写保护位）进行修改，在程序退出时则进行还原。（详细实现见 4.4 中代码）

(4) Linux系统调用的执行过程

软中断指令int 0x80 执行时，系统调用号会被放进eax寄存器，同时，sys_call_table每一项占用4个字节。这样，system_call函数可以读取eax寄存器获得当前系统调用的系统调用号，将其乘以4产生偏移地址，然后以sys_call_table为基址，基址加上偏移地址所指向的内容即是应该执行的系统调用服务程序的地址。另外，除了传递系统调用号到eax寄存器，假如需要，还会传递一些参数到内核，比如write系统调用的服务程序原型为：调用write系统调用时就需要传递文件描述符fd、要写进的内容buf以及写进字节数count等几个内容到内核。ebx、ecx、edx、esi以及edi寄存器可以用于传递这些额外的参数。

如之前所述，系统调用服务程序定义中的asmlinkage标记表示，编译器仅从堆栈中获取该函数的参数，而不需要从寄存器中获得任何参数。进入system_call函数前，用户应用将参数存放到对应寄存器中，system_call函数执行时会首先将这些寄存器压进堆栈。

具体的系统调用的执行过程如图 4-1 所示。



注①：此处语句为： `call *SYMBOL_NAME(sys_call_table)(,%eax,4);` eax 中为系统调用号

图 4-1 Linux系统调用执行过程

4.4 实验步骤

在学习完为 Linux 添加一个动态模块之后，可以编写一个模块，在模块初始化时修改某一系统调用的服务程序函数地址，模块退出时恢复原来的系统调用服务程序函数地址。本实验以修改 78 号系统调用为例来演示如何动态地修改一个系统调用。78 号系统调用的原始功能为获取当前时间(gettimeofday), Ubuntu 的桌面环境要使用到此系统调用，在修改这个系统调用前，可以按下 Ctrl+Alt+F1 组合键进入纯命令行模式，测试完成后再按下 Ctrl+Alt+F7 组合键返回桌面环境。

步骤一： 建立模块源程序 modify_syscall.c。

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

//original,syscall 78 function: gettimeofday
// new syscall 78 function: print "No 78 syscall has changed to hello" and return a+b+c
#define sys_No 78

unsigned long old_sys_call_func;
unsigned long p_sys_call_table=0xc0342860; // find in /boot/System.map-'uname -r'
unsigned long orig_cr0;

static int clear_cr0(void)
//modify the value of WP in CR0
{
    unsigned int cr0 = 0;
    unsigned int ret;
    asm volatile ("movl %%cr0, %%eax":"=a"(cr0));
    ret = cr0;
    cr0 &= 0xfffffff;
    asm volatile ("movl %%eax, %%cr0": : "a"(cr0));
    return ret;
}

static void setback_cr0(int val)
//recover the value of WP
{
```

```
    asm volatile ("movl  %%eax,  %%cr0": : "a"(val));
}

asmlinkage int hello(int a,int b,int c) //new function
{
    printk("No 78  syscall has changed to hello");
    return a+b+c;
}

void modify_syscall(void)
{
    unsigned long *sys_call_addr;
    orig_cr0 = clear_cr0();
    sys_call_addr=(unsigned long *)(p_sys_call_table+sys_No*4);
    old_sys_call_func=(sys_call_addr);
    *(sys_call_addr)=(unsigned long)&hello; // point to new function
}

void restore_syscall(void)
{
    unsigned long *sys_call_addr;
    sys_call_addr=(unsigned long *)(p_sys_call_table+sys_No*4);
    *(sys_call_addr)=old_sys_call_func; // point to original function
    setback_cr0(orig_cr0);
}

static int mymodule_init(void)
{
    modify_syscall();
    return 0;
}

static void mymodule_exit(void)
{
    restore_syscall();
}

module_init(mymodule_init);
```

```
module_exit(mymodule_exit);
```

```
MODULE_LICENSE("GPL");
```

步骤二：同目录下建立 Makefile 文件。

```
ifneq ($(KERNELRELEASE),)
```

```
obj-m := modify_syscall.o
```

```
else
```

```
KERNELDIR := /lib/modules/$(shell uname -r)/build
```

```
PWD := $(shell pwd)
```

```
modules:
```

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

```
endif
```

```
clean:
```

```
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

步骤三：编译模块。

```
#make
```

步骤四：加载模块前测试 78 号系统调用。

编写测试程序 before_syscall_test.c

```
#include<stdio.h>
```

```
#include<sys/time.h>
```

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    struct timeval    tv;
```

```
    time_t sec_t = 0;
```

```
    struct tm *tblock;
```

```
    tv.tv_sec=0;
```

```
    tv.tv_usec=0;
```

```
    tblock = localtime(&sec_t) ;
```

```
    printf("old syscall\n");
```

```
    printf("before call, the time is:    %s\n",  asctime(tblock));
```

```
    //before modify syscall 78 :gettimeofday
```

```
int ret1=syscall(78,&tv,NULL);

sec_t = tv.tv_sec;

tblock = localtime(&sec_t) ;

printf("after call, the time is: %s\n",  asctime(tblock));

printf("\nnew syscall\n");

//after modify syscall 78

int ret2=syscall(78,10,20,30);

printf("%d\n",ret2);

return 0;

}
```

编译源程序

```
# gcc -o test before_syscall_test.c
```

```
#运行可执行程序
```

```
# ./test
```

结果:

old syscall

before call, the time is: Thu Jan 1 08:00:00 1970

after call, the time is: Mon Feb 9 16:31:55 2015

after modify syscall

-1

由此处结果可以看出，在加载模块之前，78 号系统调用的功能是获取当前时间。

步骤五：先按下 Ctrl+Alt+F1 组合键进入纯命令行环境，然后加载模块。

```
#insmod modify_syscall.ko
```

步骤六：加载模块后测试 78 号系统调用。

编写测试程序 after_syscall_test.c

```
#include<stdio.h>
```

```
#include<sys/time.h>
```

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    struct timeval    tv;

    time_t sec_t = 0;

    struct tm *tblock;

    tv.tv_sec=0;

    tv.tv_usec=0;

    tblock = localtime(&sec_t) ;

    printf("old syscall\n");

    printf("before call, the time is;    %s\n",  asctime(tblock));

    //before modify syscall 78 :gettimeofday

    int ret1=syscall(78,&tv,NULL);

    sec_t = tv.tv_sec;

    tblock = localtime(&sec_t) ;

    printf("after call, the time is; %s\n",  asctime(tblock));

    printf("\nnew syscall\n");

    //after modify syscall 78

    int ret2=syscall(78,10,20,30);

    printf("%d\n",ret2);

    return 0;

}
```

编译源程序

```
# gcc -o test after_syscall_test.c
```

#运行可执行程序

```
# ./test
```

结果:

old syscall

before call, the time is: Thu Jan 1 08:00:00 1970

after call, the time is: Mon Feb 9 16:32:34 2015

after modify syscall

60

由此处结果可以看出，加载模块之后，78 号系统调用的功能已经被篡改为新的功能：

返回三个整数的和。

步骤七：卸载模块。

```
#rmmod modify_syscall
```

步骤八：模块卸载后测试 78 号系统调用。

编写测试程序 `restore_syscall_test.c`

```
#include<stdio.h>
```

```
#include<sys/time.h>
```

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    struct  timeval      tv;
```

```
    time_t sec_t = 0;
```

```
    struct tm *tblock;
```

```
    tv.tv_sec=0;
```

```
    tv.tv_usec=0;
```

```
    tblock = localtime(&sec_t) ;
```

```
    printf("old syscall\n");
```

```
    printf("before call, the time is;    %s\n",  asctime(tblock));
```

```
    //before modify syscall 78 :gettimeofday
```

```
    int ret1=syscall(78,&tv,NULL);
```

```
    sec_t = tv.tv_sec;
```

```
    tblock = localtime(&sec_t) ;
```

```
    printf("after call, the time is; %s\n",  asctime(tblock));
```

```
    printf("\nnew syscall\n");
```

```
    //after modify syscall 78
```

```
    int ret2=syscall(78,10,20,30);
```

```
    printf("%d\n",ret2);
```

```
    return 0;
```

```
}
```

编译源程序

```
# gcc -o test restore_syscall_test.c
```

```
#运行可执行程序
```

```
# ./test
```

结果：

```
old syscall
```

```
before call, the time is: Thu Jan 1 08:00:00 1970
```

```
after call, the time is: Mon Feb 9 16:33:45 2015
```

```
after modify syscall
```

```
-1
```

由此处结果可以看出，卸载模块之后，78 号系统调用的功能恢复正常。

实验五 文件系统

实验内容:

- 模拟 Ext2 文件系统原理设计实现一个类 Ext2 文件系统。
- 为 Linux 添加一个新的文件系统。

5.1 类 ext2 文件系统

5.1.1 实验目的

通过一个简单文件系统的设计,加深理解文件系统的内部功能及内部实现,为提出更好的文件系统做好准备。

5.1.2 实验内容

在分析 Linux 的文件系统的基础上,设计实现一个完全类似于 Ext 文件系统的虚拟多级文件系统。

1) 可以实现下列几条命令:

login

ls

create

delete

cd

close

read

write

format

password

2) 列目录时要列出文件类型、文件名、创建时间、最后访问时间和修改时间。

3) 源文件可以进行读写保护。

5.1.3 Ext2 文件系统分析

Ext 文件系统是对 minix 文件系统的扩展。minix 文件系统的软盘文件系统的结构如图 5-1 所示。

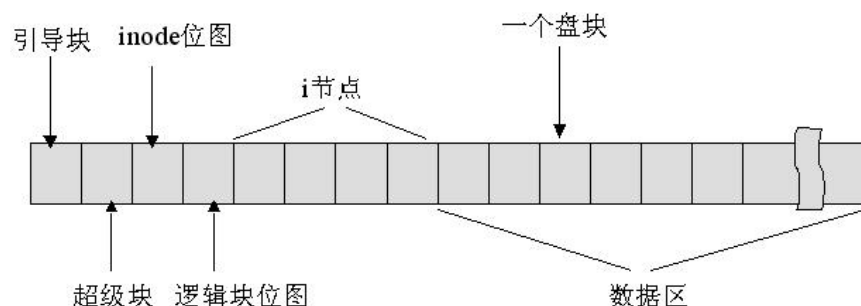


图 5-1 minix 文件系统 360KB 软盘文件系统示意图

其中文件名长 14 个字符，块设备最大容量 $64\text{M}(=8*(8*1024)*1024/(1024*1024))$ 。

Ext 文件系统：minix 的第一次扩展，文件名最大长度 255 个字符；磁盘分区大小可达 2GB。

Ext2 文件系统——第二代扩展文件系统：磁盘分区大小可达 4TB；磁盘布局采用了组块。其结构如图 5-2 所示。

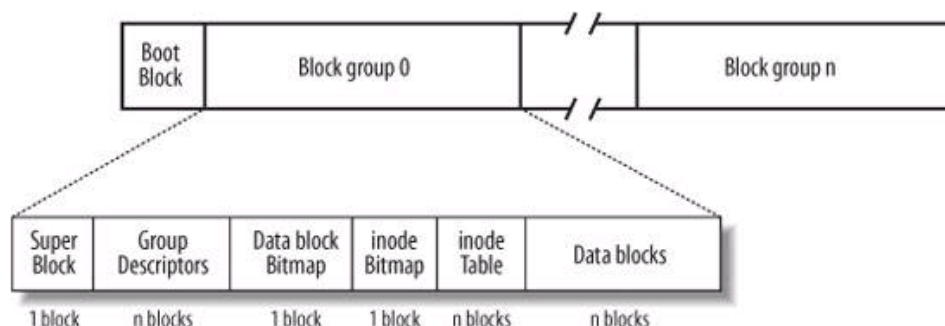


图 5-2 Ext2 文件系统结构示意图

Ext2 文件系统的引导块 BootBlock：每个硬盘分区的开头 1024 字节，即 0 byte 至 1023 byte 是分区的启动扇区。存放由 ROM BIOS 自动读入的引导程序和数据，但这只对引导设备有效，而对于非引导设备，该引导块不含代码。这个块与 Ext2 没有任何关系。

超级块 Super Block: 分区剩余的部分被分为若干个组，每个组里均由一个 Super Block 块一个 Group Descriptors（组描述符）块、Data block Bitmap、inode Bitmap、inode Table 和 Data blocks 组成。Group 0 中的 Super Block 被内核所用，定义了诸如文件系统的静态结构，包括块的大小、总块数、每组内 inode 数、空闲块、索引结点数等全局信息。其他 Group 中的 Super Block 则仅是 Group 0 中的 Super Block 的一个拷贝。超级块的数据结构在 ext2_fs.h 定义为 struct ext2_super_block，Linux 启动时 Group 0 中的 Super Block 的内容读入内存，某个组损坏可用来恢复。

```
struct ext2_super_block{
```

Type	Field	释意
__u32	s_inodes_count;	索引结点的总数
__u32	s_blocks_count;	文件系统块的总数
__u32	s_r_blocks_count;	为超级用户保留的块数
__u32	s_free_blocks_count;	空闲块总数
__u32	s_free_inodes_count;	空闲索引节点总数
__u32	s_first_data_block;	文件系统中第一个数据块
__u32	s_log_block_size;	用于计算逻辑块的大小
__s32	s_log_frag_size;	用于计算片的大小
__u32	s_blocks_per_group;	每个组的块个数
__u32	s_frags_per_group;	每个组的片个数
__u32	s_inodes_per_group;	每个组的索引节点数
__u32	s_mtime;	文件系统的安装时间
__u32	s_wtime;	最后一次对超级块进行写的时间
__u16	s_mnt_count;	安装计数
__s16	s_max_mnt_count;	最大可安装计数

```
};
```

组描述符：GroupDescriptors 定义了块位图的块号，索引结点位图的块号、索引结点表的起始块号，本组空闲块的个数等组内信息。文件系统根据这些信息来查找数据块位图，索引结点位图，索引结点表的位置。数据结构在 ext2_fs.h 中定义：

```
struct ext2_group_desc{
```

Type	Field	释意
__u32	bg_block_bitmap;	指向该组中块位图所在块的指针
__u32	bg_inode_bitmap;	指向该组中块节点位图所在块的指针
__u32	bg_inode_table;	指向该组中节点的首块的指针
__u16	bg_free_blocks_count;	本组空闲块的个数
__u16	bg_free_inodes_count;	本组空闲索引节点的个数
__u16	bg_used_dirs_count;	本组分配给目录的节点数
__u16	bg_pad;	填充
__u32	bg_reserved;	保留

```
};
```

数据块位图：Ext2 文件系统的数据块位图 DataBlockBitmap 这是 Ext2 管理存储空间的方法。即位图法。每个位对应一个数据块，位值为 0 表示空闲，1 表示已经分配。数据块位

图定义为一个块大小。于是，一个组中的数据块个数就决定了。假设块大小为 b 字节。可以区别的块数为 $b*8$ 个 数据块 DataBlocks，每个组的数据最大个数是在块大小定义后就确定了。所以组容量也就确定了。假设块大小为 b 字节。那么组容量就确定为 $(b*8)*b$ 字节，若 1 块=4K，则组块大小=4K*8*4K=128M。

索引结点位图：Ext2 文件系统的索引结点位图 `inodeBitmap` 与数据块位图相似，用来表示索引结点是否已经被使用。假设块大小为 b 字节，每个索引结点数据结构大小为 128 字节。最多可以有 $b*8$ 个索引结点，索引结点表需要占用的存储空间大小为 $(b*8)*128$ 字节。即 $(b*8)*128/b=8*128$ 个块

索引结点：`inode` 索引结点表由若干个索引结点数据结构组成，需要占用若干个块。Ext2 中的每个索引结点数据结构大小为 128 字节。每个索引结点即对应一个文件或是目录。是对其除文件名（目录名）以外的所有属性的描述。例如：文件类型，文件创建时间，访问时间，修改时间，文件所占数据块的个数，指向数据块的指针。其中，数据块指针是由 15 个元组的数据组成，如图 5-3 所示。

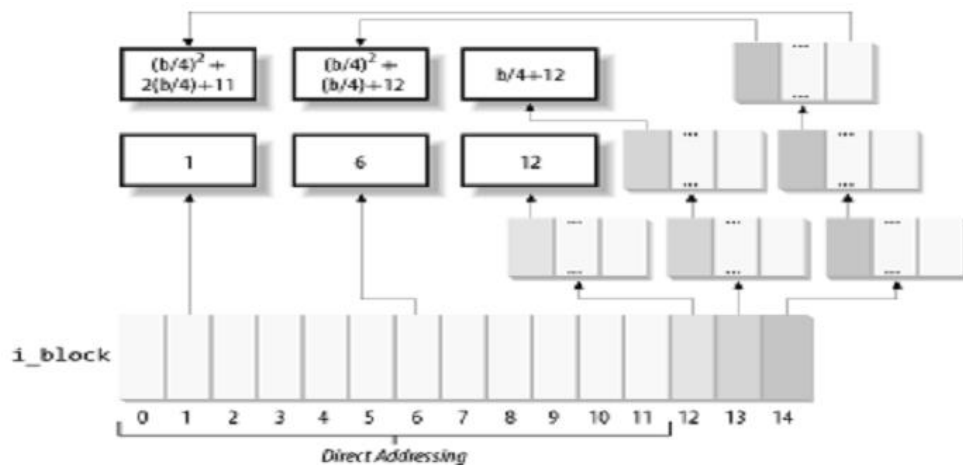


图 5-3 索引节点位图及其索引节点

前 12 个元组（0 至 11）直接指向数据块。第 12 号元组（12）是一个 1 级子索引。指向的不是数据块，而是存放数据块指针的块。类似的，第 13 号元组（13）是一个 2 级子索引。第 14 号元组（14）是一个 3 级子索引。这种结构可以适应大文件的存储。假设文件块大小为 b 。则当文件长度小于 $b*12$ 时，只要用前 12 个元组来指向其数据块。当文件长度大于 $b*12$ 时，则用多级索引机制来指向其数据块。这体现了 Ext2 的文件的物理结构，即索引文件。使用索引结点来指向文件的数据块，并且索引结点本身是聚集在一起的。

Ext2 文件系统_索引节点定义：

```
struct ext2_inode{
```

Type	Field	释意
__u16	i_mode;	文件类型及访问权限
__u16	i_uid;	文件拥有者的标识号 UID
__u32	i_size;	文件大小大小(字节)
__u32	i_atime;	最后一次访问时间
__u32	i_ctime;	创建时间
__u32	i_mtime;	该文件内容最后修改时间
__u32	i_dtime;	文件删除时间
__u16	i_gid;	文件的用户组的组号
__u16	i_links_count;	文件的链接计数
__u32	i_blocks;	文件的数据块个数（以 512 字节计）
__u32	i_flags;	打开文件的方式
__u32[]	i_block[EXT2_N_BLOCKS];	指向数据块的指针数组
__u32	i_generation;	文件的版本号 (用于 NFS)
__u32	i_file_acl;	文件访问控制表(ACL 已不再使用)
__u32	i_dir_acl;	目录访问控制表(ACL 已不再使用)
__u8	l_i_frag;	每块中的片数
__u8	l_i_fsize;	片的大小
__u32	l_i_reserved;	保留

```
};
```

Ext2 文件系统中并不是所有的文件都需要数据块，有些文件只需要索引结点即可。Ext2 中，目录是一种特殊的文件。其数据块中的内容即目录表：所包含的文件（当然包括子目录）的文件名，指向文件的索引节点号，文件类型。

ext2_fs.h 中目录体的数据结构定义：

```
struct ext2_dir_entry{
```

Type	Field	释意
__u32	inode;	索引节点号
__u16	rec_len;	目录项长度
__u8	name_len;	文件名长度
__u8	file_type;	文件类型(1:普通文件, 2:目录.....)
char[]	name[EXT2_NAME_LEN];	文件名

```
};
```

其中 MAX_NAME_LEN 定义为 255。即文件名最大长度为 255 个字符（字节同时，目

录项取为 4 的整数。故，目录项的长度范围是 12 至 264 字节。目录结构示意图如图 5-4 所示。

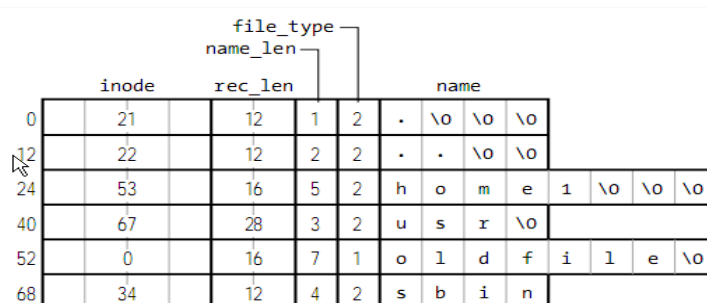


图 5-4 Ext2 的目录结构

其中，inode 指向 0 的那一行，表示此文件已经被删除。同时，上一行的目录项长度扩展为两者之和(12+16)。Ext2 文件系统的设计目标在于：一方面，从用户的角度看，实现“按名取存”，文件系统的用户只要知道所需文件的文件名，就可存取文件中的信息，而无需知道这些文件究竟存放在什么地方。另一方面，从系统角度看，文件系统对文件存储器的存储空间进行组织、分配和回收，负责文件的存储、检索、共享和保护。用户角度的文件系统被称为 VFS，虚拟文件系统。定义在这个层次上的操作，对于用户来说是文件系统所呈现所有属性。比如，列目录，建立目录，建立文件，打开文件，读写文件，重命名文件等。与每个操作相对应的是实际文件系统中的一系列底层操作。Ext2 文件系统的硬盘空间管理：存储空间的分配回收需要考虑两个因素：避免文件碎片和时效。Ext2 中新建立的文件并不是马上就分配数据块，而是分配索引结点。当有数据存入时，再分配数据块。故存储空间的分配与回收也分成两个部分索引结点的创建与删除数据块的分配与回收。

创建索引结点时，Ext2 使用 `ext2_new_inode()` 来创建一个新的索引结点。创建成功则返回索引结点号，否则返回 `NULL`。这个函数包括了以下几个步骤：

- (1) `get_empty_inode()` 返回一个空结点。
- (2) `lock_super()` 互斥操作。即对 `superblock` 的 `P()` 操作。
- (3) 如果索引结点是目录，则将其分配到空闲块最多的组中。
- (4) 如果索引结点不是目录，则从头开始“顺序查找”一个空结点分配给该文件。
- (5) `load_inode_bitmap()` 载入选中的组的索引结点位图块。找到选择首个空闲索引结点，分配之，并将对应位图位填成 1。
- (6) 将组描述符中的空闲索引结点个数减 1。
- (7) 将 `SuperBlock` 中的空闲索引结点个数减 1。
- (8) 填充刚刚分配的索引结点的数据结构。比如，文件属性，创建时间，访问时间等等。

- (9) 将新的索引结点插入索引结点 hash 表，加速查询。
- (10) mark_indoe_dirty() 将索引结点号填入 superblock 的 dirty inode list 中。
- (11) unlock_super() 互斥操作完毕，解锁。即，对 superblock 的 V()操作。
- (12) 返回新分配的索引结点号。

索引结点的删除时，Ext2 使用 ext2_free_inode()来删除一个索引结点。在此之前，内核必须已经事先完成了一系列的操作：删除文件占用的数据块，索引结点已经从 hash 表中删除，包含该索引结点的目录中索引结点号置 0、前一个目录项长度重新计算等。此函数包含以下步骤：

- (1) lock_super() 互斥操作，对 superblock 加锁。
- (2) 根据索引结点号以及索引结点总数来计算该索引结点所在的组号。
- (3) load_inode_bitmap() 从所在的组中调入索引结点位图。将对应位置 0。
- (4) clear_inode() 释放该索引结点所占用的 cache 空间。
- (5) 将组描述符中的空闲索引结点个数增 1。
- (6) 将 SuperBlock 中的空闲索引结点个数增 1。
- (7) 从 SuperBlock 的 dirty inode list 中清除该索引结点。
- (8) uplock_super() 互斥操作完毕。对 superblock 解锁。

对数据块进行寻址时，ext2 的单个文件所占有的数据块组织方式，在“硬盘数据结构”部分中已经分析过了。即 15 个元组长度的数据。前 12 个直接指向数据块，后 3 个分别采用 1 级，2 级，3 级索引方法。最大可支持的文件大小为 2TB。块大小，文件大小上限与索引级数的关系如表 5-1。

表 5-1 块大小，文件大小上限与索引级数的关系

块的大小	直接块	一级间接块	二级间接块	三级间接块
1024 字节	12KB	268KB	64.26MB	16.06GB
2048 字节	24KB	1.02MB	513.02MB	256.6GB
4096 字节	48KB	4.04MB	4GB	2TB

分配数据块进行分配时，内核使用 ext2_getblk()函数给需要数据空间的文件分配数据块。多个块与文件间的组织形式即前面所述的数据块寻址方式（多级索引）。为了在 Ext2 分区中找到空闲的数据块，Ext2 使用 ext2_alloc_block()函数。为了减少文件的碎片，Ext2 文件系统尽量从已经分配给该文件的最后一个数据块开始寻找空闲块。

Ext2 采用了预先分配数据块的方法。文件不仅仅得到其所申请的数据块，还得被预先（额外）分配了最多达 8 个的连续数据块。ext2_inode_info 数据结构中，i_prealloc_count 域保存了预先分配给文件而没有被使用的数据块。而 i_prealloc_block 域则保存着下一个将被预分配的块号。

ext2_alloc_block()函数得到一个指向索引结点和 goal 的指针。goal 是一个代表下一个将被分配的最优新块的逻辑块号。ext2_getblk()根据以下规则来设置 goal 参数（即判断最优的原则）：

(1) 如果已经被分配以及预先被分配给文件的块，是连续的块的话，goal 即预先分配的数据块的逻辑块号加 1。

(2) 如果上述规则不满足，并且至少一个块已经被预先分配，则 goal 即这些预先分配的数据块中的某个块的逻辑块号。

(3) 如果上述规则不满足，则 goal 即为文件的索引结点所在的组的第一个数据块的逻辑块号。

ext2_alloc_block()函数检查 goal 是否指向一个预先分配给该文件的一个块。是，则返回该块的逻辑块号。否则，忽略所有已经预分配的块，调用 ext2_new_block()函数，查找新的空闲数据块。

ext2_new_block()函数查找空闲块的策略。如果传递给 ext2_alloc_block()函数的预分配块（goal）空闲，则分配之 否则，检查紧跟预分配块之后的 64 个块是否有空闲，如果这 64 个块中没有空闲的块，则考虑所有的组。从包含 goal 的组开始过搜索。

对于每一个组：查找连续的 8 个空闲块。如果没有，则查找单个空闲块。一旦空闲块找到，搜索即停止。（first match 算法）结束分配之前，ext2_new_block()仍然试图预分配 8 个连接的空闲块。

释放数据块时，当一个进程删除文件或是将文件长度置 0 的时候，所有分配给该文件的数据块将被回收。此时调用的函数是 ext2_truncate()。该函数调用的参数是文件的索引结点地址。此函数将扫描索引结点中 i_block 数组，得到分配给该文件的数据块的块号信息。所有这些数据块，将由 ext2_free_blocks()函数负责回收。

ext2_free_blocks()函数调用的参数有：文件的索引结点，要被释放的数据块的首地址（块号），要被连续释放的块数。此函数首先调用 lock_super()给 superblock 上锁，然后进行如下操作：

(1) 得到数据块位图。

- (2) 将要被释放的数据块所对应的位图中的位置 0。
- (3) 将组描述符中的“空闲块数”域 (bg_free_blocks_count) 增 1。
- (4) 将 superblock 中的“空闲块数”域 (s_free_blocks_count) 增 1。

最后, 再调用 uplock_super() 给 superblock 解锁。

Ext2 的普通文件读写实现虚拟文件系统 (VFS) 上对文件的读写是通过系统调用 read() 或 write() 来实现的。从文件系统的层次看, 这些系统调用对应着一系列的“底层”操作。下面分析 Ext2 文件系统的写操作过程。

Ext2 的写操作函数是 ext2_file_write()。它有 4 个参数:

- (1) fd 文件描述符;
- (2) buf 写入数据的缓冲区地址;
- (3) count 写入的字节大小;
- (4) ppos 指向保存文件地址指针。

写函数的工作过程如下步骤:

- (1) 剔除对该文件拥有特权的特权用户。
- (2) 将写入地址移动到文件需要写入的地方。
- (3) 计算文件内的地址与文件系统中块的地址的对应关系。
- (4) 写操作前准备:
 - a. 调用 ext2_getblk() 得到空闲数据块;
 - b. 等待可能存在的读操作完成;
 - c. 将写缓冲区的数据复制到数据块;
 - d. 调用 update_vm_cache() 将 cache 中的数据同步。
- (5) 更新文件索引结点中的 i_size 域。
- (6) 设置索引结点中与时间相关域。比如, 文件访问时间, 修改时间。
- (7) 更新 ppos 指针。
- (8) 返回已经写入文件的字节数。

5.1.4 设计原理

Ext2 文件系统是一个实际可用的文件系统, 是庞大而复杂的。基于 Ext2 的思想和算法, 设计一个类 Ext2 的文件系统, 实现 Ext2 文件系统的一个功能子集, 以真正了解文件系统的运作机制。在设计实现中, 为了简化操作, 可用现有操作系统的文件来代替硬盘进行硬件模

拟。

1) 设计文件系统应该考虑的几个层次

- 介质的物理结构
- 物理操作——设备驱动程序完成
- 文件系统的组织结构（逻辑组织结构）
- 对组织结构其上的操作
- 为用户使用文件系统提供的接口

对于类Ext2 文件系统而言，由于采用硬盘文件代替模拟物理硬盘，不涉及具体的物理设备（硬盘）的操作，也就是说不涉及介质的物理结构和设备的驱动，仅考虑三个层次，即文件系统低层的组织结构、其上的具体操作以及为用户提供的接口命令。而类Ext2 文件系统的低层组织结构，实际上就是硬盘（用文件代替）的空间划分、存放信息的布局、采用的数据结构；其上的操作就是如何管理这些空间、如何读写维护其上的信息，为了实现这些功能，需要定义具体的操作（函数）；为了用户对文件系统使用的方便，当然要求该文件系统应为用户提供一些接口，这些接口可以是编程接口也可以是命令接口。类Ext2 文件系统的层次结构如图 5-5 所示。

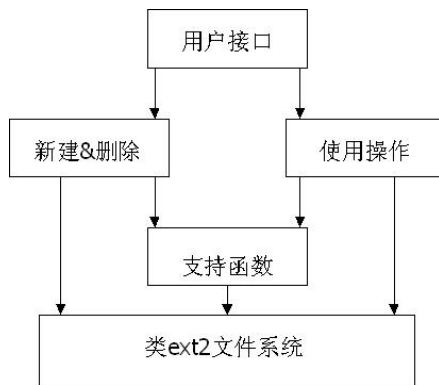


图 5-5 类Ext2 文件系统层次结构图

从另外一个角度可以认为，对于类Ext2 文件系统可以划分为三个部分：接口和用户界面设计、节点创建删除设计、节点操作设计。

2) 类ext2 文件系统的数据结构

(1) 块的定义

为简单起见，逻辑块大小与物理块大小作为参考均可定义为 512 字节。由于位图只占用一个块，因此，每个组的数据块个数以及索引结点的个数均确定为 $512 \times 8 = 4096$ 。进一步，每组的数据容量确定为 $4096 \times 512\text{B} = 2\text{MB}$ 。另外，类Ext2 中，假设只有一个用户，故可以省略

去文件的所有者ID的域。

(2) 组描述符

为简单起见，可以只定义一个组。因此，组描述符只占用一个块。同时，superblock 块可以省略，其功能由组描述符块代替，即组描述符块中需要增加文件系统大小，索引结点的大小，卷名等原属于superblock的域。由此可得组描述符的数据结构如下：

```
struct ext2_group_desc{
```

类型	bytes	域	释意
char[]	16	bg_volume_name[16];	卷名
int	2	bg_block_bitmap;	保存块位图的块号
int	2	bg_inode_bitmap;	保存索引结点位图的块号
int	2	bg_inode_table;	索引结点表的起始块号
int	2	bg_free_blocks_count;	本组空闲块的个数
int	2	bg_free_inodes_count;	本组空闲索引结点的个数
int	2	bg_used_dirs_count;	本组目录的个数
char[]	16	psw[16];	密码
char[]	24	bg_pad[24];	填充(0xff)
};			

合计 80 字节，只占用一个块。

(3) 索引结点数据结构

由于容量已经确定，文件（即硬盘）最大即为 2MB。需要 4096 个数据块。索引结点的数据结构中，仍然采用多级索引机制。由于文件系统总块数必然小于 4096×2 ，所以只需要 13 个二进制位即可对块进行全局计数，实际实现用 unsigned int 16 位变量，即 2 字节表示 1 个块号。

在一级子索引中，如果一个数据块都用来存放块号，则可以存放 $512/2=256$ 个。因此，只使用一级子索引可以容纳最大的文件为 $256 \times 512=128KB$ 。需要使用二级子索引。只使用二级子索引时，索引结点中的一个指针可以指向 256×256 个块，即 $256 \times 256 \times 512=8MB$ ，已经可以满足要求了。为了尽量“像”ext2，也为了简单起见，索引结点的直接索引定义 6 个，一级子索引定义 1 个，二级子索引定义 1 个。总计 8 个指针。

索引结点的数据结构定义：

```
struct ext2_inode {
```

类型	字节长度	域	释意
int	2	i_mode;	文件类型及访问权限
int	2	i_blocks;	文件的数据块个数
int	2	i_size;	大小(字节)

```

time_t    4      i_atime;    访问时间
time_t    4      i_ctime;    创建时间
time_t    4      i_mtime;    修改时间
time_t    4      i_dtime;    删除时间
int        16     i_block[8]; 指向数据块的指针
char[]     24     i_pad[24];  填充 1(0xff)
}

```

即，每个索引结点的长度为 84 字节。

其中，i_mode域构成一个文件访问类型及访问权限描述。即linux中的drwxrwxrwx描述。d为目录，r为读控制，w为写控制，x为可执行标志。并且，3个rwx 分别是所有者(owner)，组(group)，全局(universe)这三个对象的权限。为了简单的起见，模拟系统中，i_mode 的 16 位如下分配。高 8 位(high_i_mode)，是目录项中文件类型码的一个拷贝。低 8 位(low_i_mode)中的最低 3 位分别用来标识rwx3 个属性。高 5 位不用，用 0 填充。在显示文件访问权限时，3 个对象均使用低 3 位的标识。这样，由这 16 位，即可生成一个文件的完整的drwxrwxrwx描述。

特别的，注意在Unix 中，不带扩展名的文件定性为可执行文件。在类Ext2 文件系统中，凡是扩展名为.exe,.bin,.com及不带扩展名的，都被加上x标识。

i_block 域与文件大小以及数据块的关系如图 5-6 所示。

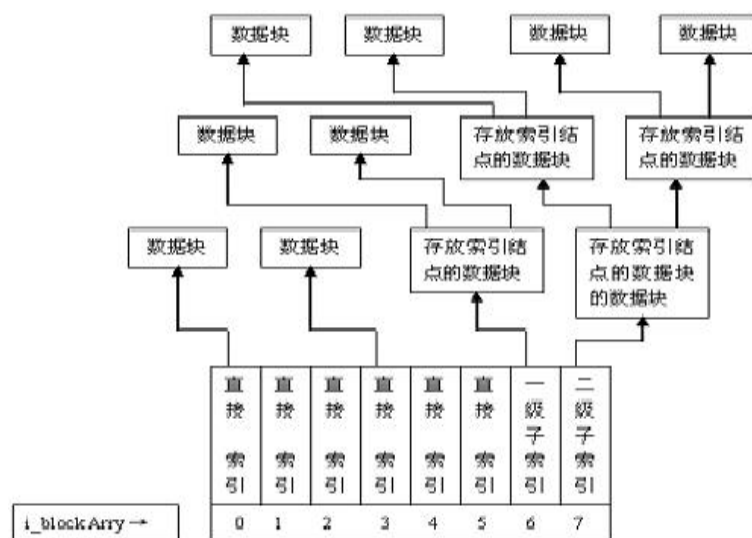


图 5-6 类Ext2 文件系统的二级文件索引

当文件长度小于等于 $512 \times 6 = 3072$ 字节（3KB）时，只用到直接索引。

当文件长度大于 3072 字节，并且小于等于 $3KB + 128KB$ 即 131KB 时，除使用直接索引处，

还将使用一级子索引。

当文件长度大于 131KB，并且小于 2MB（文件系统最大值）时，将开始使用二级子索引。

（4）索引结点表

由于每个索引结点大小为 64 个字节，最多有 $512 \times 8 = 4096$ 个索引结点。故，索引结点表的大小为 $64 \times 4096 = 256\text{KB}$ ，512 个块。为了和 Ext2 保持一致，索引结点从 1 开始计数，0 表示 NULL。数据块则从 0 开始计数。

（5）类 Ext2 文件系统的“硬盘”数据结构

基于以上若干定义，得到类 Ext2 文件系统的“硬盘”数据结构如表 5-2 所示。

表 5-2 类 Ext2 文件系统“硬盘” 数据结构

组描述符	数据块位图	索引结点位图	索引结点表	数据块
1 block	1 block	1 block	512 block	4096 block
512 Bytes	512 Bytes	512 Bytes	256KB	2MB

整个类 Ext2 文件系统所需要的“硬盘”空间为 $1+1+1+512+4096=4611$ 个块。共计 $4611 \times 512\text{bytes} = 2,360,832$ 字节 $= 2305.5\text{KB} = 2.25\text{MB}$ 。

3) 目录和文件

与 Ext2 相同，目录作为特殊的文件来处理。将第 1 个索引结点指向根目录。根目录的索引结点中直接索引域指向数据块 0。

目录体的数据结构与 Ext2 基本相同，唯一的区别在于索引节点号用 16 位来表示：

```
struct ext2_dir_entry {
```

Type	Bytes	Field	释意
int	2	inode;	索引节点号
int	2	rec_len;	目录项长度
int	2	name_len;	文件名长度
int	2	file_type;	文件类型(1:普通文件, 2:目录...)
char[]	15	name[EXT2_NAME_LEN];	文件名
char	1	dir_pad;	填充
};			

当文件系统在初始化时，根目录的数据块（即数据块 1）将被初始化。其所包含的所有索引节点号以及目录项长度域将被置 0。当文件被删除时，其所在目录项长度不变，索引节

点号将被置 0。

当新建一个文件时，程序将从目录的数据块查找索引节点号为 0 的目录项，并检查其长度是否够用。是，则分配给该文件，否则继续查找，直到找到长度够用，或者是长度为 0（即未被使用过）的地址，为文件建立目录项。

当建立的是一个目录时，将其所分配到的索引结点所指向的数据块清空。并且自动写入两个特殊的目录项。一个是当前目录“.”，其索引结点即指向本身的数据块。另一个是上一级目录“..”，其索引结点指向上一级目录的数据块。例如，/root 目录。其索引结点号为 1。并且，第 1 个数据块存放着该目录的目录项。/root 目录在文件系统初始化时自动生成。同时，在目录项中自动生成以下两项：

inode	rec_len	name_len	file_type	name		
1	8	1	2	.	\0	
1	9	2	2	.	.	\0

作为简化，文件类型使用 1 表示普通文件，2 表示目录。

4) “存储空间”管理

这里涉及到类 Ext2 文件系统的 5 个底层操作：索引结点的分配与释放、数据块的分配与释放以及数据块的寻址。

这些操作将采用 Ext2 基本相同的方法实现。区别在于：Ext2 中对 SuperBlock 的操作将变成对组描述符的操作。此外，数据块在分配时不采取预先分配策略。查找空闲块的方法可采用从某个起始点开始线性查找。

5) 类 Ext2 文件系统的操作

作为一个类似于 Ext2 的文件系统，应该提供诸如下相关操作。

操作命令	参数	功能
format	无	格式化文件系统
password	无	修改文件系统的密码
ls	无	列举当前目录的文件（目录）
create	文件类型，文件名	在当前目录创建文件（目录）
delete	文件类型，文件名	在当前目录删除文件（目录）
cd	目录名	切换目录
close	目录深度	按照目录深度关闭目录
read	文件名	从文件中读取数据
write	文件名	向文件中写入数据

6) 内存中数据结构

为了实现 5) 中定义的类型Ext2 文件系统的这些操作，内存中也必须有相应的数据结构。

```
unsigned int fopen_table[16]; //打开文件表，暂无实现  
ext2_group_desc group_desc; //组描述符  
ext2_inode inode; //inode节点  
ext2_dir_entry dir; //目录体  
unsigned int last_allco_inode=0; //上次分配的索引节点号  
unsigned int last_allco_block=0; //上次分配的数据块号
```

7) 程序结构

(1) 初始化类Ext2 文件系统

在已有的文件系统的基础上建立一个大小为FS_SPACE=2,360,832 字节(即 2.25MB)的文件vdisk，这个文件即用来模拟(代替)硬盘。以后，文件系统的所有操作，均通过读写这个文件实现。并且，完全模拟硬盘读写方式，一次读取 1 个块，即 512 字节。即使只有 1 个字节的修改，也通过读写一个数据块来实现。另外，常驻内存的数据结构也被初始化。

(2) 文件系统级(底层)函数及其子函数

这些函数完成所有文件系统底层的操作封装，并为上层即命令层提供服务。该层实现了所有对文件系统“硬盘”的块操作功能。例如：分配和回收索引结点与数据块、索引结点的读取与写入、数据块的读取与写入、索引结点及数据块位图的设置、组描述符的修改以及多级索引的实现等。

(3) 命令层函数

文件系统所支持的命令及其功能在这一层实现。一共实现 11 个命令：ls, create, delete, cd, close, read, write, format, password, login, logout。为了实现这些命令，本层使用底层所提供的服务。

(4) 用户接口层

主要功能是接收及识别用户命令，进行词法分析，提取命令及参数。组织调用命令层对应的命令实现相应功能。本层实际上是一个基于命令层基础上的shell。为了完善接口的功能，shell 程序中增加退出程序的命令exit。

5.1.5 实验步骤

根据 5.1.3 节中的设计原理和相关的分析编写源程序，并在Linux下进行调试，验证达

到类似Ext2 文件系统的要求。

1) 运行结果

(1) 初始化，输入初始密码 123。

```
root@dell-virtual-machine:/home/dell/shiyan/shiyan5# ./myext2fs
Hello! Welcome to Ext2_like file system!
please input the password(init:123):123

.=># ls
```

图 5-7 类Ext2 文件系统初始化

(2) password修改文件系统的密码。

```
.=># password
Please input the old password
123
Please input the new password:123456
Modify the password?[Y/N]y

.=>#
```

图 5-8 类Ext2 文件系统修改密码

(3) format格式化文件系统。

```
.=># format
Do you want to format the filesystem?
It will be dangerous to your data.
[Y/N]y

.=># ls
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014
Directory ..           Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014
Directory Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014
Directory Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014      Wed Jan 29 02:37:41 2014

.=>#
```

图 5-9 类Ext2 文件系统格式化

(4) create创建文件。

```
.=># create d mydir
Congratulations! mydir is created

.=># ls
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
Directory ..           Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
Directory Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
Directory Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
Directory mydir       Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014

.=>#
```

图 5-10 类Ext2 文件系统创建文件示意图

(5) cd切换目录，ls列出文件和目录。

```
.=># cd mydir
mydir=># ls
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014
Directory ..           Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
mydir=>#
```

图 5-11 类Ext2 文件系统切换目录，查看目录示意图

(6) read和write读写文件。

```
mydir=># create f myfile
Congratulations! myfile is created
mydir=># write myfile
hello
mydir=># read myfile
hello
mydir=># ls
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014      Wed Jan 29 02:39:29 2014
Directory ..           Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014      Mon Jan 27 20:08:39 2014
File      myfile        Wed Jan 29 02:42:11 2014      Wed Jan 29 02:42:44 2014      Wed Jan 29 02:42:33 2014
mydir=>#
```

图 5-12 类Ext2 文件系统读写文件示意图

2) 附程序源代码

```
#include <stdio.h>
#include "string.h"
#include "stdlib.h"
#include "time.h"
#include <sys/ioctl.h>
#include <termios.h>
#include <unistd.h> /* for STDIN_FILENO */
#define blocks 4611 //1+1+1+512+4096, 总块数
#define blocksiz 512 //每块字节数
#define inodesiz 64 //索引长度
#define data_begin_block 515 //数据开始块
#define dirsiz 32 //目录体长度
#define EXT2_NAME_LEN 15 //文件名长度
#define PATH "vdisk" //文件系统

typedef struct ext2_group_desc //组描述符 68 字节
{
```

```

    char bg_volume_name[16]; //卷名
    int bg_block_bitmap; //保存块位图的块号
    int bg_inode_bitmap; //保存索引结点位图的块号
    int bg_inode_table; //索引结点表的起始块号
    int bg_free_blocks_count; //本组空闲块的个数
    int bg_free_inodes_count; //本组空闲索引结点的个数
    int bg_used_dirs_count; //本组目录的个数
    char psw[16];
    char bg_pad[24]; //填充(0xff)
}ext2_group_desc;

typedef struct ext2_inode //索引节点 64 字节
{
    int i_mode; //文件类型及访问权限
    int i_blocks; //文件的数据块个数
    int i_size; //大小(字节)
    time_t i_atime; //访问时间
    time_t i_ctime; //创建时间
    time_t i_mtime; //修改时间
    time_t i_dtime; //删除时间
    int i_block[8]; //指向数据块的指针
    char i_pad[24]; //填充 1(0xff)
}ext2_inode;

typedef struct ext2_dir_entry //目录体 32 字节
{
    int inode; //索引节点号
    int rec_len; //目录项长度
    int name_len; //文件名长度
    int file_type; //文件类型(1:普通文件, 2:目录...)
    char name[EXT2_NAME_LEN]; //文件名
    char dir_pad; //填充
}ext2_dir_entry;

/*定义全局变量*/
ext2_group_desc group_desc; //组描述符
ext2_inode inode;
ext2_dir_entry dir; //目录体
FILE *f; /*文件指针*/
unsigned int last_allco_inode=0; //上次分配的索引节点号
unsigned int last_allco_block=0; //上次分配的数据块号

/******/
int getch ()

```

```

{
    int ch;
    struct termios oldt, newt;
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ECHO|ICANON);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    ch = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return ch;
}

/*****格式化文件系统*****/
/*
* 初始化组描述符
* 初始化数据块位图
* 初始化索引节点位图
* 初始化索引节点表 -添加一个索引节点
* 第一个数据块中写入当前目录和上一目录
*/
int format()
{
    FILE* fp=NULL;
    int i;
    unsigned int zero[blocksiz/4]; //零数组，用来初始化块为0
    time_t now;
    time(&now);
    while(fp==NULL)
        fp=fopen(PATH, "w+");
    for(i=0; i<blocksiz/4; i++)
        zero[i]=0;
    for(i=0; i<blocks; i++) //初始化所有 4611 块为0
    {
        fseek(fp, i*blocksiz, SEEK_SET);
        fwrite(&zero, blocksiz, 1, fp);
    }

    //初始化组描述符
    strcpy(group_desc.bg_volume_name, "abcd"); //初始化卷名为abcd
    group_desc.bg_block_bitmap=1; //保存块位图的块号
    group_desc.bg_inode_bitmap=2; //保存索引节点位图的块号
    group_desc.bg_inode_table=3; //索引节点表的起始块号
    group_desc.bg_free_blocks_count=4095; //除去一个初始化目录
    group_desc.bg_free_inodes_count=4095;

```

```

group_desc.bg_used_dirs_count=1;
strcpy(group_desc.psw, "123");
fseek(fp, 0, SEEK_SET);
fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp); //第一块为组描述符

//初始化数据块位图和索引节点位图，第一位置为 1
zero[0]=0x80000000;
fseek(fp, 1*blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp); //第二块为块位图，块位图的第一位为 1
fseek(fp, 2*blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp); //第三块为索引位图，索引节点位图的第一位为 1

//初始化索引节点表，添加一个索引节点
inode.i_mode=2;
inode.i_blocks=1;
inode.i_size=64;
inode.i_ctime=now;
inode.i_atime=now;
inode.i_mtime=now;
inode.i_dtime=0;
fseek(fp, 3*blocksiz, SEEK_SET);
fwrite(&inode, sizeof(ext2_inode), 1, fp); //第四块开始为索引节点表

//向第一个数据块写 当前目录
dir.inode=0;
dir.rec_len=32;
dir.name_len=1;
dir.file_type=2;
strcpy(dir.name, "."); //当前目录
fseek(fp, data_begin_block*blocksiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp);

//当前目录之后写 上一目录
dir.inode=0;
dir.rec_len=32;
dir.name_len=2;
dir.file_type=2;
strcpy(dir.name, ".."); //上一目录
fseek(fp, data_begin_block*blocksiz+dirsiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp); //第data_begin_block+1 =516 块
开始为数据
fclose(fp);
return 0;
}

```

```

//返回目录的起始存储位置，每个目录 32 字节
int dir_entry_position(int dir_entry_begin, int i_block[8])// dir_entry_begin
目录体的相对开始字节
{
    int dir_blocks=dir_entry_begin/512; // 存储目录需要的块数
    int block_offset=dir_entry_begin%512; // 块内偏移字节数
    int a;
    FILE* fp=NULL;
    if(dir_blocks<=5) //前六个直接索引
        return
data_begin_block*blocksiz+i_block[dir_blocks]*blocksiz+block_offset;
    else //间接索引
    {
        while(fp==NULL)
            fp=fopen(PATH, "r+");
        dir_blocks=dir_blocks-6;
        if(dir_blocks<128) //一个块 512 字节，一个int为 4 个字节 一级索引有 512/4
=128 个
        {
            int a;

fseek(fp, data_begin_block*blocksiz+i_block[6]*blocksiz+dir_blocks*4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);
            return data_begin_block*blocksiz+a*blocksiz+block_offset;
        }
        else //二级索引
        {
            dir_blocks=dir_blocks-128;

fseek(fp, data_begin_block*blocksiz+i_block[7]*blocksiz+dir_blocks/128*4, SEEK_SE
T);

            fread(&a, sizeof(int), 1, fp);

fseek(fp, data_begin_block*blocksiz+a*blocksiz+dir_blocks%128*4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);
            return data_begin_block*blocksiz+a*blocksiz+block_offset;
        }
        fclose(fp);
    }
}

/*

```

```

在当前目录 打开一个目录
current指向新打开的当前目录 (ext2_inode)
*/
int Open(ext2_inode *current, char *name)
{
    FILE* fp=NULL;
    int i;
    while(fp==NULL)
        fp=fopen(PATH, "r+");
    for(i=0; i<(current->i_size/32); i++)
    {
        fseek(fp, dir_entry_position(i*32, current->i_block), SEEK_SET); //定位目
录的偏移位置
        fread(&dir, sizeof(ext2_dir_entry), 1, fp);
        if(!strcmp(dir.name, name))
        {
            if(dir.file_type==2) //目录
            {
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fread(current, sizeof(ext2_inode), 1, fp);
                fclose(fp);
                return 0;
            }
        }
    }
    fclose(fp);
    return 1;
}

/*****关闭当前目录*****/
/*
关闭时仅修改最后访问时间
返回时 打开上一目录 作为当前目录
*/
int Close(ext2_inode *current)
{
    time_t now;
    ext2_dir_entry bentry;
    FILE* fout;
    fout=fopen(PATH, "r+");
    time(&now);
    current->i_atime=now; //修改最后访问时间
    fseek(fout, (data_begin_block+current->i_block[0])*blocksiz, SEEK_SET);
    fread(&bentry, sizeof(ext2_dir_entry), 1, fout); //current's dir_entry

```

```

    fseek(fout, 3*blocksiz+(bentry.inode)*sizeof(ext2_inode), SEEK_SET);
    fwrite(current, sizeof(ext2_inode), 1, fout);
    fclose(fout);
    return Open(current, "..");
}

/*
read file content from directory 'current' in file 'name'
*/
int Read(ext2_inode *current, char *name)
{
    FILE* fp=NULL;
    int i;
    while(fp==NULL)
        fp=fopen(PATH, "r+");
    for(i=0; i<(current->i_size/32); i++)
    {
        fseek(fp, dir_entry_position(i*32, current->i_block), SEEK_SET);
        fread(&dir, sizeof(ext2_dir_entry), 1, fp);
        if(!strcmp(dir.name, name))
        {
            if(dir.file_type==1)
            {
                time_t now;
                ext2_inode node;
                char content_char;
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fread(&node, sizeof(ext2_inode), 1, fp); // original inode
                i=0;
                for(i=0; i<node.i_size; i++)
                {
                    fseek(fp, dir_entry_position(i, node.i_block), SEEK_SET);
                    fread(&content_char, sizeof(char), 1, fp);
                    if(content_char==0xD)
                        printf("\n");
                    else
                        printf("%c", content_char);
                }
                printf("\n");
                time(&now);
                node.i_atime=now;
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fwrite(&node, sizeof(ext2_inode), 1, fp); //update inode
                fclose(fp);
            }
        }
    }
}

```

```

        return 0;
    }
}
}
fclose(fp);
return 1;
}

//寻找空索引
int FindInode()
{
    FILE* fp=NULL;
    unsigned int zero[blocksiz/4];
    int i;
    while(fp==NULL)
        fp=fopen(PATH,"r+");
    fseek(fp,2*blocksiz,SEEK_SET); //inode 位图
    fread(zero,blocksiz,1,fp); //zero保存索引节点位图
    for(i=last_allco_inode;i<(last_allco_inode+blocksiz/4);i++)
    {
        if(zero[i%(blocksiz/4)]!=0xffffffff)
        {
            unsigned int j=0x80000000,k=zero[i%(blocksiz/4)],l=i;
            for(i=0;i<32;i++)
            {
                if(!(k&j))
                {
                    zero[l%(blocksiz/4)]=zero[l%(blocksiz/4)]|j;
                    group_desc.bg_free_inodes_count-=1; //索引节点数减 1
                    fseek(fp,0,0);
                    fwrite(&group_desc,sizeof(ext2_group_desc),1,fp); //更新组描
述符

                    fseek(fp,2*blocksiz,SEEK_SET);
                    fwrite(zero,blocksiz,1,fp); //更新inode位图
                    last_allco_inode=l%(blocksiz/4);
                    fclose(fp);
                    return l%(blocksiz/4)*32+i;
                }
            }
            else
                j=j/2;
        }
    }
}
}

```



```

    fclose(fp);
    return -1;
}

//寻找空block
int FindBlock()
{
    FILE* fp=NULL;
    unsigned int zero[blocksiz/4];
    int i;
    while(fp==NULL)
        fp=fopen(PATH, "r+");
    fseek(fp, 1*blocksiz, SEEK_SET);
    fread(zero, blocksiz, 1, fp); //zero保存块位图
    for(i=last_allco_block; i<(last_allco_block+blocksiz/4); i++)
    {
        if(zero[i%(blocksiz/4)]!=0xffffffff)
        {
            unsigned int j=0X80000000, k=zero[i%(blocksiz/4)], l=i;
            for(i=0; i<32; i++)
            {
                if(!(k&j))
                {
                    zero[l%(blocksiz/4)]=zero[l%(blocksiz/4)]|j;
                    group_desc.bg_free_blocks_count-=1; //块数减1
                    fseek(fp, 0, 0);
                    fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp);
                    fseek(fp, 1*blocksiz, SEEK_SET);
                    fwrite(zero, blocksiz, 1, fp);
                    last_allco_block=l%(blocksiz/4);
                    fclose(fp);
                    return l%(blocksiz/4)*32+i;
                }
            }
            else
                j=j/2;
        }
    }
    fclose(fp);
    return -1;
}

```

```
//删除inode, 更新inode节点位图
```

```
void DelInode(int len)
{
    unsigned int zero[blocksiz/4], i;
    int j;
    f=fopen(PATH, "r+");
    fseek(f, 2*blocksiz, SEEK_SET);
    fread(zero, blocksiz, 1, f);
    i=0x80000000;
    for(j=0; j<len%32; j++)
        i=i/2;
    zero[len/32]=zero[len/32]^i;
    fseek(f, 2*blocksiz, SEEK_SET);
    fwrite(zero, blocksiz, 1, f);
    fclose(f);
}
```

```
//删除block块, 更新块位图
```

```
void DelBlock(int len)
{
    unsigned int zero[blocksiz/4], i;
    int j;
    f=fopen(PATH, "r+");
    fseek(f, 1*blocksiz, SEEK_SET);
    fread(zero, blocksiz, 1, f);
    i=0x80000000;
    for(j=0; j<len%32; j++)
        i=i/2;
    zero[len/32]=zero[len/32]^i;
    fseek(f, 1*blocksiz, SEEK_SET);
    fwrite(zero, blocksiz, 1, f);
    fclose(f);
}
```

```
void add_block(ext2_inode *current, int i, int j)
```

```
{
    FILE *fp=NULL;
    while(fp==NULL)
        fp=fopen(PATH, "r+");
    if(i<6)//直接索引
    {
        current->i_block[i]=j;
    }
}
```

```

    }
    else
    {
        i=i-6;
        if(i==0)
        {
            current->i_block[6]=FindBlock();

fseek(fp, data_begin_block*blocksiz+current->i_block[6]*blocksiz, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp);
        }
        else if(i<128) //一级索引
        {

fseek(fp, data_begin_block*blocksiz+current->i_block[6]*blocksiz+i*4, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp);
        }
        else //二级索引
        {
            i=i-128;
            if(i==0)
            {
                current->i_block[7]=FindBlock();

fseek(fp, data_begin_block*blocksiz+current->i_block[7]*blocksiz, SEEK_SET);
                i=FindBlock();
                fwrite(&i, sizeof(int), 1, fp);
                fseek(fp, data_begin_block*blocksiz+i*blocksiz, SEEK_SET);
                fwrite(&j, sizeof(int), 1, fp);
            }
            if(i%128==0)
            {

fseek(fp, data_begin_block*blocksiz+current->i_block[7]*blocksiz+i/128*4, SEEK_SE
T);

                i=FindBlock();
                fwrite(&i, sizeof(int), 1, fp);
                fseek(fp, data_begin_block*blocksiz+i*blocksiz, SEEK_SET);
                fwrite(&j, sizeof(int), 1, fp);
            }
        }
        else
        {

fseek(fp, data_begin_block*blocksiz+current->i_block[7]*blocksiz+i/128*4, SEEK_SE

```

```

T);

        fread(&i, sizeof(int), 1, fp);

fseek(fp, data_begin_block*blocksiz+i*blocksiz+i%128*4, SEEK_SET);
        fwrite(&j, sizeof(int), 1, fp);
    }
}
}
}

// 为当前目录寻找一个空目录体
int FindEntry(ext2_inode *current)
{
    FILE* fout=NULL;
    int location; //条目的绝对地址
    int block_location; //块号
    int temp; //每个block 可以存放的INT 数量
    int remain_block; //剩余块数
    location=data_begin_block*blocksiz;
    temp=blocksiz/sizeof(int);
    fout=fopen(PATH, "r+");
    if (current->i_size % blocksiz ==0) //一个BLOCK 使用完后增加一个块
    {
        add_block(current, current->i_blocks, FindBlock());
        current->i_blocks++;
    }
    if (current->i_blocks<6) //前 6 个块直接索引
    {
        location+=current->i_block[current->i_blocks-1]*blocksiz;
        location+=current->i_size % blocksiz;
    }
    else if (current->i_blocks<temp+5) //一级索引
    {
        block_location=current->i_block[6];

fseek(fout, (data_begin_block+block_location)*blocksiz+(current->i_blocks-6)*siz
eof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        location+=block_location*blocksiz;
        location+=current->i_size % blocksiz;
    }
    else //二级索引

```

```

    {
        block_location=current->i_block[7];
        remain_block=current->i_blocks-6-temp;

fseek(fout, (data_begin_block+block_location)*blocksiz+(int)((remain_block-1)/temp+1)*sizeof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        remain_block=remain_block % temp;

fseek(fout, (data_begin_block+block_location)*blocksiz+remain_block*sizeof(int),
SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        location+=block_location*blocksiz;
        location+=current->i_size % blocksiz+dirsiz;
    }
    current->i_size+=dirsiz;
    fclose(fout);
    return location;
}

/*****创建文件或者目录*****/
/*
 * type=1 创建文件      type=2 创建目录
 * current 当前目录索引节点
 * name 文件名或目录名
 */
int Create(int type, ext2_inode *current, char *name)
{
    FILE* fout=NULL;
    int i;
    int block_location;//block location
    int node_location;//node location
    int dir_entry_location;//dir entry location

    time_t now;
    ext2_inode ainode;
    ext2_dir_entry aentry, bentry; //bentry保存当前系统的目录体信息
    time(&now);
    fout=fopen(PATH, "r+");
    node_location=FindInode();
    for(i=0; i<current->i_size/dirsiz; i++)
    {
        fseek(fout, dir_entry_position
(i*sizeof(ext2_dir_entry), current->i_block), SEEK_SET);

```

```

    fread(&aentry, sizeof(ext2_dir_entry), 1, fout);
    if(aentry.file_type==type&&!strcmp(aentry.name, name))
        return 1;
}
fseek(fout, (data_begin_block+current->i_block[0])*blocksiz, SEEK_SET);
fread(&bentry, sizeof(ext2_dir_entry), 1, fout); //current's dir_entry
if (type==1) //文件
{
    ainode.i_mode=1;
    ainode.i_blocks=0; //文件暂无内容
    ainode.i_size=0; //初始文件大小为0
    ainode.i_atime=now;
    ainode.i_ctime=now;
    ainode.i_mtime=now;
    ainode.i_dtime=0;
    for(i=0;i<8;i++)
    { ainode.i_block[i]=0;}
    for(i=0;i<24;i++)
    {
        ainode.i_pad[i]=(char) (0xff);
    }
}
else //目录
{
    ainode.i_mode=2;//目录
    ainode.i_blocks=1;//目录 当前和上一目录
    ainode.i_size=64;//初始大小 32*2=64
    ainode.i_atime=now;
    ainode.i_ctime=now;
    ainode.i_mtime=now;
    ainode.i_dtime=0;
    block_location=FindBlock();
    ainode.i_block[0]=block_location;
    for(i=1;i<8;i++)
    { ainode.i_block[i]=0;}
    for(i=0;i<24;i++)
    { ainode.i_pad[i]=(char) (0xff);}
    //当前目录
    aentry.inode=node_location;
    aentry.rec_len=sizeof(ext2_dir_entry);
    aentry.name_len=1;
    aentry.file_type=2;
    strcpy(aentry.name, ".");
    aentry.dir_pad=0;
}

```

```

        fseek(fout, (data_begin_block+block_location)*blocksiz, SEEK_SET);
        fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);
        //上一级目录
        aentry.inode=bentry.inode;
        aentry.rec_len=sizeof(ext2_dir_entry);
        aentry.name_len=2;
        aentry.file_type=2;
        strcpy(aentry.name, "..");
        aentry.dir_pad=0;
        fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);
        //一个空条目
        aentry.inode=0;
        aentry.rec_len=sizeof(ext2_dir_entry);
        aentry.name_len=0;
        aentry.file_type=0;
        aentry.name[EXT2_NAME_LEN]=0;
        aentry.dir_pad=0;
        fwrite(&aentry, sizeof(ext2_dir_entry), 14, fout); //清空数据块
    } //end else
    //保存新建inode
    fseek(fout, 3*blocksiz+(node_location)*sizeof(ext2_inode), SEEK_SET);
    fwrite(&ainode, sizeof(ext2_inode), 1, fout);
    // 将新建inode 的信息写入current 指向的数据块
    aentry.inode=node_location;
    aentry.rec_len=dirsiz;
    aentry.name_len=strlen(name);
    if (type==1) {aentry.file_type=1;} //文件
    else {aentry.file_type=2;} //目录
    strcpy(aentry.name, name);
    aentry.dir_pad=0;
    dir_entry_location=FindEntry(current);
    fseek(fout, dir_entry_location, SEEK_SET); //定位条目位置
    fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);
    //保存current 的信息, bentry 是current 指向的block 中的第一条
    fseek(fout, 3*blocksiz+(bentry.inode)*sizeof(ext2_inode), SEEK_SET);
    fwrite(current, sizeof(ext2_inode), 1, fout);
    fclose(fout);
    return 0;
}

/*****
/*
* write data to file 'name' in directory 'current'

```

```

* if there isn't this file in this directory ,remaind create a new one
*/
int Write(ext2_inode *current, char *name)
{
    FILE* fp=NULL;
    ext2_dir_entry dir;
    ext2_inode node;
    time_t now;
    char str;
    int i;
    while(fp==NULL)
        fp=fopen(PATH, "r+");
    while(1)
    {
        for(i=0; i<(current->i_size/32); i++)
        {
            fseek(fp, dir_entry_position(i*32, current->i_block), SEEK_SET);
            fread(&dir, sizeof(ext2_dir_entry), 1, fp);
            if(!strcmp(dir.name, name))
            {
                if(dir.file_type==1)
                {
                    fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                    fread(&node, sizeof(ext2_inode), 1, fp);
                    break;
                }
            }
        }
        if(i<current->i_size/32) //have file
            break;
        //Create(1, current, name); //have not file ,create a new file
        printf("There isn't this file, please create it first\n");
        return 0;
    }
    str=getch();
    while(str!=27)
    {
        printf("%c", str);
        if(!(node.i_size%512))
        {
            add_block(&node, node.i_size/512, FindBlock());
            node.i_blocks+=1;
        }
        fseek(fp, dir_entry_position(node.i_size, node.i_block), SEEK_SET);
    }
}

```



```
        else
            printf("Directory\t%s\t\t%s", dir.name, timestr);
    }
    fclose(f);
}

int initialize(ext2_inode *cu)
{
    f=fopen(PATH, "r+");
    fseek(f, 3*blocksiz, 0);
    fread(cu, sizeof(ext2_inode), 1, f);
    fclose(f);
    return 0;
}

/*****修改文件系统密码*****/
/*
* 修改成功返回 0
* 修改不成功返回 1
*/
int Password()
{
    char psw[16], ch[10];
    printf("Please input the old password\n");
    scanf("%s", psw);
    if(strcmp(psw, group_desc.psw) != 0)
    {
        printf("Password error!\n");
        return 1;
    }
    while(1)
    {
        printf("Please input the new password:");
        scanf("%s", psw);
        while(1)
        {
            printf("Modify the password?[Y/N]");
            scanf("%s", ch);
            if(ch[0] == 'N' || ch[0] == 'n')
            {
                printf("You canceled the modify of your password\n");
                return 1;
            }
            else if(ch[0] == 'Y' || ch[0] == 'y')
```

```

        {
            strcpy(group_desc.psw, psw);
            f=fopen(PATH, "r+");
            fseek(f, 0, 0);
            fwrite(&group_desc, sizeof(ext2_group_desc), 1, f);
            fclose(f);
            return 0;
        }
    else
        printf("Meaningless command\n");
    }
}

/*****/
int login()
{
    char psw[16];
    printf("please input the password(init:123):");
    scanf("%s", psw);
    return strcmp(group_desc.psw, psw);
}

/*****/
void exitdisplay()
{
    printf("Thank you for using~ Byebye!\n");
    return;
}

/*****/初始化文件系统*****/
/*返回 1 初始化失败, 返回 0 初始化成功*/
int initfs(ext2_inode *cu)
{
    f=fopen(PATH, "r+");
    if(f==NULL)
    {
        //char ch[20];/*****/
        char ch;
        int i;
        printf("File system couldn't be found. Do you want to create
one?\n[Y/N]");
        i=1;
        while(i)

```

```

    {
        scanf("%c",&ch);/*****/
        switch(ch)
        {
            case 'Y':
            case 'y':/*****/
                if(format()!=0) return 1;
                f=fopen(PATH,"r");
                i=0;
                break;
            case 'N':
            case 'n':/*****/
                exitdisplay();
                return 1;
            default:
                printf("Sorry, meaningless command\n");
                break;
        }
    }
}

fseek(f, 0, SEEK_SET);
fread(&group_desc, sizeof(ext2_group_desc), 1, f);
fseek(f, 3*blocksiz, SEEK_SET);
fread(&inode, sizeof(ext2_inode), 1, f);
fclose(f);
initialize(cu);
return 0;
}

/****获取当前目录的目录名*****/
void getstring(char *cs, ext2_inode node)
{
    ext2_inode current=node;
    int i, j;
    ext2_dir_entry dir;
    f=fopen(PATH, "r+");
    Open(&current, ".."); //current指向上一目录
    for(i=0; i<node.i_size/32; i++)
    {
        fseek(f, dir_entry_position(i*32, node.i_block), SEEK_SET);
        fread(&dir, sizeof(ext2_dir_entry), 1, f);
        if(!strcmp(dir.name, "."))
        {
            j=dir.inode;

```

```

        break;
    }
}
for(i=0;i<current.i_size/32;i++)
{
    fseek(f, dir_entry_position(i*32, current.i_block), SEEK_SET);
    fread(&dir, sizeof(ext2_dir_entry), 1, f);
    if(dir.inode==j)
    {
        strcpy(cs, dir.name);
        return;
    }
}
}

/*****在当前目录删除目录或者文件*****/
int Delet(int type, ext2_inode *current, char *name)
{
    FILE* fout=NULL;
    int i, j, t, k, flag;
    //int Nlocation, Elocation, Blocation,
    int Blocation2, Blocation3;
    int node_location, dir_entry_location, block_location;
    int block_location2, block_location3;
    ext2_inode cinode;
    ext2_dir_entry bentry, centry, dentry;
    //一个空条目
    dentry.inode=0;
    dentry.rec_len=sizeof(ext2_dir_entry);
    dentry.name_len=0;
    dentry.file_type=0;
    strcpy(dentry.name, "");
    dentry.dir_pad=0;
    fout=fopen(PATH, "r+");
    t=(int)(current->i_size/dirsiz); //总条目数
    flag=0; //是否找到文件或目录
    for(i=0;i<t;i++)
    {
        dir_entry_location=dir_entry_position(i*dirsiz, current->i_block);
        fseek(fout, dir_entry_location, SEEK_SET);
        fread(&centry, sizeof(ext2_dir_entry), 1, fout);
        if ((strcmp(centry.name, name)==0)&&(centry.file_type==type))
        { flag=1; j=i; break; }
    }
}

```

```

    if (flag)
    {
        node_location=centry.inode;
        fseek(fout, 3*blocksiz+node_location*sizeof(ext2_inode), SEEK_SET); //定位INODE 位置
        fread(&cinode, sizeof(ext2_inode), 1, fout);
        block_location=cinode.i_block[0];
        //删文件夹
        if (type==2)
        {
            if (cinode.i_size>2*dirsiz) {printf("The folder is not empty!\n");}
            else
            {
                DelBlock(block_location);
                DelInode(node_location);
            }
        }

        dir_entry_location=dir_entry_position(current->i_size, current->i_block); //找到current 指向条目的最后一条
        fseek(fout, dir_entry_location, SEEK_SET);
        fread(&centry, dirsiz, 1, fout); //将最后一条条目存入centry
        fseek(fout, dir_entry_location, SEEK_SET);
        fwrite(&dentry, dirsiz, 1, fout); //清空该位置
        dir_entry_location-=data_begin_block*blocksiz; //在数据中的位置
        //如果这个位置刚好是一个块的起始位置，则删掉这个块
        if (dir_entry_location % blocksiz==0)
        {
            DelBlock((int) (dir_entry_location/blocksiz));
            current->i_blocks--;
            if(current->i_blocks==6)
                DelBlock(current->i_block[6]);
            else if(current->i_blocks==(blocksiz/sizeof(int)+6))
            {
                int a;

                fseek(fout, data_begin_block*blocksiz+current->i_block[7]*blocksiz, SEEK_SET);
                fread(&a, sizeof(int), 1, fout);
                DelBlock(a);
                DelBlock(current->i_block[7]);
            }
            else
                if(!((current->i_blocks-6-blocksiz/sizeof(int))%(blocksiz/sizeof(int))))
            {
                int a;

```

```

fseek(fout, data_begin_block*blocksiz+current->i_block[7]*blocksiz+((current->i_
blocks-6-blocksiz/sizeof(int))/(blocksiz/sizeof(int))), SEEK_SET);
        fread(&a, sizeof(int), 1, fout);
        DelBlock(a);
    }
}
current->i_size-=dirlsiz;
if (j*dirlsiz<current->i_size) //删除的条目如果不是最后一条, 用
centry覆盖
    {

dir_entry_location=dir_entry_position(j*dirlsiz, current->i_block);
        fseek(fout, dir_entry_location, SEEK_SET);
        fwrite(&centry, dirlsiz, 1, fout);
    }
    printf("The %s is deleted!", name);
}
}
//删文件
else
{
    //删直接指向的块
    for(i=0;i<6;i++)
    {
        if (cinode.i_blocks==0) {break;}
        block_location=cinode.i_block[i];
        DelBlock(block_location);
        cinode.i_blocks--;
    }
    //删一级索引中的块
    if (cinode.i_blocks>0)
    {
        block_location=cinode.i_block[6];

fseek(fout, (data_begin_block+block_location)*blocksiz, SEEK_SET);
        for(i=0;i<blocksiz/sizeof(int);i++)
        {
            if (cinode.i_blocks==0) {break;}
            fread(&Blocation2, sizeof(int), 1, fout);
            DelBlock(Blocation2);
            cinode.i_blocks--;
        }
        DelBlock(block_location); // 删除一级索引
    }
}

```

```

        if (cinode.i_blocks>0) //有二级索引存在
        {
            block_location=cinode.i_block[7];
            for(i=0;i<blocksiz/sizeof(int);i++)
            {

fseek(fout, (data_begin_block+block_location)*blocksiz+i*sizeof(int), SEEK_SET);
                fread(&Blocation2, sizeof(int), 1, fout);
                fseek(fout, (data_begin_block+Blocation2)*blocksiz, SEEK_SET);
                for(k=0;k<blocksiz/sizeof(int);k++)
                {
                    if (cinode.i_blocks==0) {break;}
                    fread(&Blocation3, sizeof(int), 1, fout);
                    DelBlock(Blocation3);
                    cinode.i_blocks--;
                }
                DelBlock(Blocation2); //删除二级索引
            }
            DelBlock(block_location); // 删除一级索引
        }
        DelInode(node_location); //删除文件的inode

dir_entry_location=dir_entry_position(current->i_size, current->i_block); //找到
current 指向条目的最后一条
        fseek(fout, dir_entry_location, SEEK_SET);
        fread(&centry, dirsiz, 1, fout); //将最后一条条目存入centry
        fseek(fout, dir_entry_location, SEEK_SET);
        fwrite(&dentry, dirsiz, 1, fout); //清空该位置
        dir_entry_location-=data_begin_block*blocksiz; //在数据中的位置
        //如果这个位置刚好是一个块的起始位置，则删掉这个块
        if (dir_entry_location % blocksiz==0)
        {

DelBlock((int) (dir_entry_location/blocksiz)); current->i_blocks--;
            if(current->i_blocks==6)
                DelBlock(current->i_block[6]);
            else if(current->i_blocks==(blocksiz/sizeof(int)+6))
            {
                int a;

fseek(fout, data_begin_block*blocksiz+current->i_block[7]*blocksiz, SEEK_SET);
                fread(&a, sizeof(int), 1, fout);
                DelBlock(a);
                DelBlock(current->i_block[7]);
            }
        }
    }
}

```



```

        }
        else
if(!((current->i_blocks-6-blocksiz/sizeof(int))%(blocksiz/sizeof(int))))
        {
            int a;

fseek(fout, data_begin_block*blocksiz+current->i_block[7]*blocksiz+((current->i_
blocks-6-blocksiz/sizeof(int))/(blocksiz/sizeof(int))), SEEK_SET);
            fread(&a, sizeof(int), 1, fout);
            DelBlock(a);
        }
    }
    current->i_size-=dirsiz;
    if (j*dirsiz<current->i_size)// 删除的条目如果不是最后一条, 用
centry 覆盖
    {

dir_entry_location=dir_entry_position(j*dirsiz, current->i_block);
        fseek(fout, dir_entry_location, SEEK_SET);
        fwrite(&centry, dirsiz, 1, fout);
    }
}
fseek(fout, (data_begin_block+current->i_block[0])*blocksiz, SEEK_SET);
fread(&bentry, sizeof(ext2_dir_entry), 1, fout); //current's dir_entry
fseek(fout, 3*blocksiz+(bentry.inode+1)*sizeof(ext2_inode), SEEK_SET);
fwrite(&current, sizeof(ext2_inode), 1, fout); //将current 修改的数据写回
文件
    }
    else
    {
        fclose(fout);
        return 1;
    }
    fclose(fout);
    return 0;
}

/*main shell*/
void shellloop(ext2_inode currentdir)
{
    char command[10], var1[10], var2[128], path[10];
    ext2_inode temp;
    int i, j;

```

```
char currentstring[20];
char
ctable[12][10]={"create","delete","cd","close","read","write","password","forma
t","exit","login","logout","ls"};
while(1)
{
    getstring(currentstring,currentdir);//获取当前目录的目录名
    printf("\n%s=># ",currentstring);
    scanf("%s",command);
    for(i=0;i<12;i++)
        if(!strcmp(command,ctable[i]))
            break;
    if(i==0||i==1) //创建, 删除 文件/目录
    {
        scanf("%s",var1);/*****/
        scanf("%s",var2);
        if(var1[0]=='f') j=1; //创建文件
        else if(var1[0]=='d') j=2; //创建目录
        else {printf("the first variant must be [f/d]");continue;}
        if(i==0)
        {
            if(Create(j,&currentdir,var2)==1)
                printf("Failed! %s can't be created\n",var2);
            else
                printf("Congratulations! %s is created\n",var2);
        }
        else
        {
            if(Delet(j,&currentdir,var2)==1)
                printf("Failed! %s can't be deleted!\n",var2);
            else
                printf("Congratulations! %s is deleted!\n",var2);
        }
    }
    else if(i==2) //open == cd change dir
    {
        scanf("%s",var2);
        i=0;j=0;
        temp=currentdir;
        while(1)
        {
            path[i]=var2[j];
            if(path[i]=='/')
            {
```

```
        if(j==0)
            initialize(&currentdir);
        else if(i==0)
        {
            printf("path input error!\n");
            break;
        }
        else
        {
            path[i]='\0';
            if(Open(&currentdir, path)==1)
            {
                printf("path input error!\n");
                currentdir=temp;
            }
        }
        i=0;
    }
    else if(path[i]!='\0')
    {
        if(i==0) break;
        if(Open(&currentdir, path)==1)
        {
            printf("path input error!\n");
            currentdir=temp;
        }
        break;
    }
    else i++;
    j++;
}
}
else if(i==3)//close
{
    /*imagine the second variable suply number of layers to get out of*/
    scanf("%d",&i);
    for(j=0;j<i;j++)
        if(Close(&currentdir)==1)
        {
            printf("Warning! the number %d is too large\n",i);
            break;
        }
}
else if(i==4)//read
```

```
{
    scanf("%s", var2);
    if(Read(&currentdir, var2)==1)
        printf("Failed! The file can't be read\n");
}
else if(i==5)//write
{
    scanf("%s", var2);
    if(Write(&currentdir, var2)==1)
        printf("Failed! The file can't be written\n");
}
else if(i==6)//password
    Password();
else if(i==7) //format
{
    while(1)
    {
        printf("Do you want to format the filesystem?\n It will be
dangerous to your data. \n");
        printf("[Y/N]");
        scanf("%s", var1);
        if(var1[0]=='N' || var1[0]=='n')
            break;
        else if(var1[0]=='Y' || var1[0]=='y')
        {
            format();
            break;
        }
        else
            printf("please input [Y/N]");
    }
}
else if(i==8) //exit
{
    while(1)
    {
        printf("Do you want to exit from filesystem?[Y/N]");
        scanf("%s", &var2);
        if(var2[0]=='N' || var2[0]=='n')
            break;
        else if(var2[0]=='Y' || var2[0]=='y')
            return;
        else
            printf("please input [Y/N]");
    }
}
```

```
    }
}
else if(i==9) //login
    printf("Failed! You havn't logged out yet\n");
else if(i==10)//logout
{
    while(i)
    {
        printf("Do you want to logout from filesystem?[Y/N]");
        scanf("%s", var1);
        if(var1[0]=='N' || var1[0]=='n')
            break;
        else if(var1[0]=='Y' || var1[0]=='y')
        {
            initialize(&currentdir);
            while(1)
            {
                printf("$$$$=># ");
                scanf("%s", var2);
                if(strcmp(var2, "login")==0)
                {
                    if(login()==0)
                    {
                        i=0;
                        break;
                    }
                }
                else if(strcmp(var2, "exit")==0)
                    return;
            }
        }
    }
    else
        printf("please input [Y/N]");
}
}

else if(i==11) //ls
    Ls(&currentdir);
else
    printf("Failed! Command not available\n");
}
}

int main()
{
```

```
ext2_inode cu;/*current user*/
printf("Hello! Welcome to Ext2_like file system!\n");
if(initfs(&cu)==1) return 0;
if(login()!=0)/*****/
{
    printf("Wrong password!It will terminate right away.\n");
    exitdisplay();
    return 0;
}
shellloop(cu);
exitdisplay();
return 0;
}
```

5.2 添加一个文件系统

5.2.1 实验目的

文件系统是操作系统中最直观的部分，因为用户可以通过文件直接地和操作系统交互，操作系统也必须为计算机提供数据计算、数据存储的功能。本实验通过克隆添加一个文件系统，进一步理解 Linux 环境下文件系统添加过程。

5.2.2 实验内容

在 3.8.x 内核下添加一个与 Ext2 文件系统功能相同的自定义文件系统 myext2。

5.2.3 实验原理

要克隆添加一个与 Ext2 完全相同的文件系统 myext2，首先是确定实现 Ext2 文件系统的内核源码是由哪些文件组成。通过对源文件中代码名字的修改，再修改 Linux 对 myext2 文件系统的一些操作，选中新添加的选项，重新编译内核，就可以使用了。

5.2.4 实验步骤

添加一个克隆 Ext2 文件系统的自定义文件系统 myext2，本实验基于 3.8.13 版本内核。

步骤一：代码拷贝。

首先拷贝 Ext2 文件系统所有的源代码。与 Ext2 密切相关的代码分为两部分，一部分位于 linux/fs/ext2/目录下，另一部分位于 linux/include/linux/目录下的 ext2_fs.h 文件和 ext2_fs_sb.h（在 3.8.13 内核下拷贝或网上下载）文件。

```
#cd /usr/src/linux-3.8.13

#cd fs

cp -r ext2 myext2

#cd ../include/linux

#cp ext2_fs.h myext2_fs.h

进入 ext2_fs_sb.h 文件目录

#cp ext2_fs_sb.h ../include/linux/myext2_fs_sb.h
```

这样就完成所有源代码的拷贝操作。

步骤二：修改拷贝的源代码。

对于克隆文件系统来说，这样当然还远远不够，因为文件里面的数据结构名、函数名、以及相关的一些宏等内容还没有根据 myext2 改掉，编译根本通不过。所以应该对 myext2 目

录下的所有文件进行更改，也要对以上的二个头文件也进行修改，具体修改方法如下。

1) 对新复制的两个头文件 `myext2_fs.h` 和 `myext2_fs_sb.h` 中的 `ext2` 为 `myext2`, `EXT2` 替换为 `MYEXT2`。这一操作可以使用 Ubuntu 中自带的 `gedit` 的查找和替换功能完成，或者使用 `vim` 替换功能。

```
#cd include/linux

#vim satext2_fs.h

/* 进入到 vim 里面之后输入如下内容*/

:%s/ext2/myext2/g

:%s/EXT2/MYEXT2/g
```

2) 对 `fs/myext2` 目录下的所有源文件(`.c` 和 `.h`) 执行上述替换，即替换所有的 `ext2` 为 `myext2`, `EXT2` 为 `MYEXT2`。这一操作可以使用 `gedit` 的查找或替换功能完成或者使用如下脚本。

```
#!/bin/sh
SCRIPT=substitute.sh
for f in *;
do
if [ $f = $SCRIPT ]; then
echo "skip $f"
continue
fi
echo -n "substitute ext2 to myext2 in $f..."
cat $f | sed 's/ext2/myext2/g' > ${f}_tmp
mv ${f}_tmp $f
echo "done"
echo -n "substitute EXT2 to MYEXT2 in $f..."
cat $f | sed 's/EXT2/MYEXT2/g' > ${f}_tmp
mv ${f}_tmp $f
echo "done"
done
```

把这个脚本命名为 `substitute.sh`，放在 `fs/myext2` 下面，加上可执行权限 `chmod +x`，运行之后就可以把当前目录里所有文件里面的“`ext2`”和“`EXT2`”都替换成对应的“`myext2`”和“`MYEXT2`”。同时对 `ext2.h` 文件重命名为 `myext2.h`。

步骤三：修改编译选项。

因为我们新增加了一个文件系统，而这个文件系统之前在内核的编译选项中是不存在的，所以我们要修改相关的配置，来使得 `make menuconfig` 中出现 `myext2` 选项。总体说来，

有这么三个地方需要修改。这三个地方的修改方式都是一样的，把有关 ext2 的选项的内容复制一份，然后把 ext2 和 EXT2 替换为 myext2 和 MYEXT2。

1) fs/Kconfig。

在 Kconfig 中添加如下有关内容.....

```
config MYEXT2_FS
    tristate "myext2 fs support"
    help
        Ext2 is a standard Linux file system for hard disks.
        To compile this file system support as a module, choose M here: the
        module will be called myext2.
        If unsure, say Y.
config MYEXT2_FS_XATTR
    bool "MyExt2 extended attributes"
    depends on MYEXT2_FS
    help
        Extended attributes are name:value pairs associated with inodes by
        the kernel or by users (see the attr(5) manual page, or visit
        <http://acl.bestbits.at/> for details).
        If unsure, say N.
config MYEXT2_FS_POSIX_ACL
    bool "MyExt2 POSIX Access Control Lists"
    depends on MYEXT2_FS_XATTR
    select FS_POSIX_ACL
    help
        Posix Access Control Lists (ACLs) support permissions for users and
        groups beyond the owner/group/world scheme.
        To learn more about Access Control Lists, visit the Posix ACLs for
        Linux website <http://acl.bestbits.at/>.
        If you don't know what Access Control Lists are, say N
config MYEXT2_FS_SECURITY
    bool "MyExt2 Security Labels"
    depends on MYEXT2_FS_XATTR
    help
        Security labels support alternative access control models
        implemented by security modules like SELinux. This option
        enables an extended attribute handler for file security
        labels in the myext2 filesystem.
        If you are not using a security module that requires using
        extended attributes for file security labels, say N.
config MYEXT2_FS_XIP
    bool "Ext2 execute in place support"
    depends on MYEXT2_FS && MMU
```

help

Execute in place can be used on memory-backed block devices. If you enable this option, you can select to mount block devices which are capable of this feature without using the page cache.

If you do not use a block device that is capable of using this, or if unsure, say N.

2) fs/Makefile 。

Makefile 的修改是告诉内核编译系统, 当myext2 对应的宏被选上的时候, 到fs/myext2 目录下去编译myext2 文件系统。

Makefile中添加一行代码:

```
obj-$(CONFIG_JBD2)           += jbd2/
obj-$(CONFIG_EXT2_FS)        += ext2/
obj-$(CONFIG_MYEXT2_FS)      += myext2/
obj-$(CONFIG_CRAMFS)         += cramfs/
obj-y                        += ramfs/
```

3) arch/x86/configs/i386_defconfig 。

这个文件的改动是修改默认的编译选项。仿照EXT2 的内容添加MYEXT2 内容。

```
CONFIG_EXT2_FS=y
CONFIG_EXT2_FS_XATTR=y
CONFIG_EXT2_FS_POSIX_ACL=y
# CONFIG_EXT2_FS_SECURITY is not set
# CONFIG_EXT2_FS_XIP is not set
CONFIG_MYEXT2_FS=y
CONFIG_MYEXT2_FS_XATTR=y
CONFIG_MYEXT2_FS_POSIX_ACL=y
# CONFIG_MYEXT2_FS_SECURITY is not set
# CONFIG_MYEXT2_FS_XIP is not set
```

步骤四：修改相关头文件。

如果现在开始 make, 是不能够通过编译的。因为 fs/satext2 中的一些函数还是需要引用一些头文件, 而这些头文件中只有 ext2 相关的函数的定义, 而我们需要的是 satext2 相关的函数定义。根据经验来看, 有这么几个文件需要修改。

1) include/asm-generic/bitops/ext2-atomic.h

比照 ext2 内容, 在其下面复制添加一份 myext2 内容。

```

#define myext2_set_bit_atomic(lock, nr, addr)      \
({                                                 \
    int ret;                                       \
    spin_lock(lock);                             \
    ret = ext2_set_bit((nr), (unsigned long *) (addr)); \
    spin_unlock(lock);                           \
    ret;                                           \
})

#define myext2_clear_bit_atomic(lock, nr, addr)    \
({                                                 \
    int ret;                                       \
    spin_lock(lock);                             \
    ret = ext2_clear_bit((nr), (unsigned long *) (addr)); \
    spin_unlock(lock);                           \
    ret;                                           \
})

```

2) include/asm-generic/bitops/ext2-atomic-setbit.h

依旧仿照 ext2 内容，在其下面复制添加 myext2 内容。

```

#define ext2_set_bit_atomic(lock, nr, addr)      \
    test_and_set_bit((nr), (unsigned long *) (addr)) \
#define ext2_clear_bit_atomic(lock, nr, addr)    \
    test_and_clear_bit((nr), (unsigned long *) (addr)) \
+
#define myext2_set_bit_atomic(lock, nr, addr)    \
    test_and_set_bit((nr), (unsigned long *) (addr)) \
#define myext2_clear_bit_atomic(lock, nr, addr)  \
    test_and_clear_bit((nr), (unsigned long *) (addr)) \

```

3) include/linux/magic.h

```

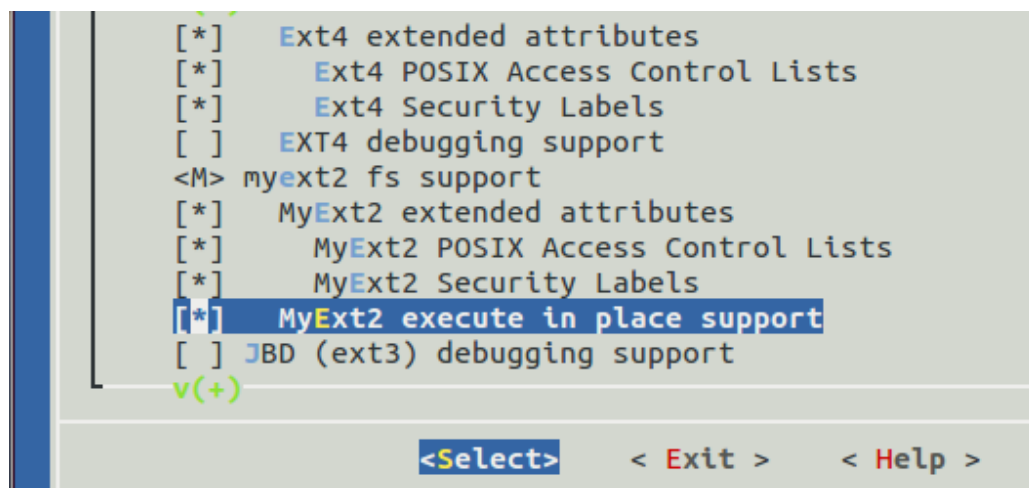
#define CODA_SUPER_MAGIC      0x73757245
#define EFS_SUPER_MAGIC      0x414A53
#define EXT2_SUPER_MAGIC      0xEF53
#define MYEXT2_SUPER_MAGIC    0xEF53
#define EXT3_SUPER_MAGIC      0xEF53

```

步骤五：重新编译内核。

编译内核的详细步骤参考实验一。

File system 下选中 myext2 fs support 各选项。



步骤六：验证新添加的文件系统。

重启系统，选中新编译的内核进入系统。

首先创建一个大小为 1M，名字为 myfs，内容全为 0 的文件。

```
#dd if=/dev/zero of=myfs bs=1M count=1
```

然后将 myfs 格式化为 ext2 文件系统。

```
#mkfs.ext2 myfs
```

```
root@ubuntu:/# pwd
/
root@ubuntu:/# dd if=/dev/zero of=myfs bs=1M count=1
记录了1+0 的读入
记录了1+0 的写出
1048576字节(1.0 MB)已复制, 0.014773 秒, 71.0 MB/秒
root@ubuntu:/# /sbin/mkfs.ext2 myfs
mke2fs 1.42.5 (29-Jul-2012)
myfs is not a block special device.
无论如何也要继续? (y,n) y
Discarding device blocks: 完成
文件系统标签=
OS type: Linux
块大小=1024 (log=0)
分块大小=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
128 inodes, 1024 blocks
51 blocks (4.98%) reserved for the super user
第一个数据块=1
Maximum filesystem blocks=1048576
1 block group
8192 blocks per group, 8192 fragments per group
128 inodes per group

Allocating group tables: 完成
正在写入inode表: 完成
Writing superblocks and filesystem accounting information: 完成
```

从理论上来看，除了名字外 myext2 和 ext2 是完全一致的。所以，下面我们可以试着用 myext2 文件系统格式去 mount 刚刚做出来的 ext2 文件系统。接着开始挂载，输入：

```
#mount -t myext2 -o loop ./myfs /mnt/myext2fs
```

(先在/mnt 目录建立 myext2fs 文件夹，使用 mkdir 命令)

将 myfs 通过 loop 设备 mount 到/mnt/myext2fs 目录下。请注意，用的参数是-t myext2，也就是用 myext2 文件系统格式去 mount 的，发现这样 mount 是可以的，也就证明了新内核已经支持新文件系统 myext2。

```
#mount
```

```

root@ubuntu:/# mount -t myext2 -o loop ./myfs /mnt/myext2fs
root@ubuntu:/# mount
/dev/loop0 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/cgroup type tmpfs (rw)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
none on /run/user type tmpfs (rw,noexec,nosuid,nodev,size=104857600,mode=0755)
/dev/sda7 on /host type fuseblk (rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other,blksize=4096)
gvfsd-fuse on /run/user/zhuxuan/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,user=zhuxuan)
/myfs on /mnt/myext2fs type myext2 (rw)

```

```

root@ubuntu:/mnt/myext2fs# df -HT
文件系统      类型      容量  已用  可用  已用%  挂载点
/dev/loop0    ext4       21G   16G   3.8G   81%   /
none          tmpfs      4.1k    0    4.1k    0%   /sys/fs/cgroup
udev          devtmpfs   988M   13k   988M    1%   /dev
tmpfs         tmpfs      211M  832k   211M    1%   /run
none          tmpfs      5.3M    0    5.3M    0%   /run/lock
none          tmpfs      1.1G  160k   1.1G    1%   /run/shm
none          tmpfs      105M   46k   105M    1%   /run/user
/dev/sda7     fuseblk    93G   22G   71G   24%   /host
/dev/loop1    myext2     1.1M   19k   957k    2%   /mnt/myext2fs

```

可以看到 myext2 文件系统已经挂在/mnt 文件目录上，之后还可以卸载。输入：

```
#umount /mnt/myext2fs
```

再次查看：#mount

```

root@ubuntu:/# mount
/dev/loop0 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
none on /sys/fs/cgroup type tmpfs (rw)
none on /sys/fs/fuse/connections type fusectl (rw)
none on /sys/kernel/debug type debugfs (rw)
none on /sys/kernel/security type securityfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
none on /run/user type tmpfs (rw,noexec,nosuid,nodev,size=104857600,mode=0755)
/dev/sda7 on /host type fuseblk (rw,nosuid,nodev,relatime,user_id=0,group_id=0,allow_other,blksize=4096)
gvfsd-fuse on /run/user/zhuxuan/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,user=zhuxuan)
root@ubuntu:/# df -HT
文件系统      类型      容量  已用  可用  已用%  挂载点
/dev/loop0    ext4       21G   16G   3.8G   81%   /
none          tmpfs      4.1k    0    4.1k    0%   /sys/fs/cgroup
udev          devtmpfs   988M   13k   988M    1%   /dev
tmpfs         tmpfs      211M  832k   211M    1%   /run
none          tmpfs      5.3M    0    5.3M    0%   /run/lock
none          tmpfs      1.1G  160k   1.1G    1%   /run/shm
none          tmpfs      105M   41k   105M    1%   /run/user
/dev/sda7     fuseblk    93G   22G   71G   24%   /host
root@ubuntu:/#

```

可以看出,通过对原文件系统的代码的修改,我们最终实现了挂载上了自己的文件系统,在学习内核知识的过程中,这样的重利用算是一种很常见且有效的方法。

实验六 字符设备驱动程序

6.1 一个简单的字符驱动程序

6.1.1 实验目的

- 1) 理解 linux 字符设备驱动程序的基本原理；
- 2) 掌握字符设备的驱动运作机制；
- 3) 在了解驱动程序的编写原则和过程的基础上，学会编写字符设备驱动程序。

6.1.2 实验内容

编写一个简单的字符设备驱动程序，该字符设备并不驱动特定的硬件，而是用内核空间模拟字符设备，要求该字符设备包括以下几个基本操作，打开、读、写和释放，并编写测试程序用于测试所编写的字符设备驱动程序。在此基础上，编写程序实现对该字符设备的同步操作。

6.1.3 实验原理

1) Linux下设备驱动程序基础知识

Linux 函数（系统调用）是应用程序和操作系统内核之间的接口，而设备驱动程序是内核和硬件设备之间的接口，设备驱动程序屏蔽硬件细节，且设备被映射成特殊的文件进行处理。每个设备都对应一个文件名，在内核中也对应一个索引节点，应用程序可以通过设备的文件名来访问硬件设备。Linux 为文件和设备提供了一致性的接口，用户操作设备文件和操作普通文件类似。例如通过 `open()` 函数可打开设备文件，建立起应用程序与目标程序的连接；之后，可以通过`read()`、`write()`等常规文件函数对目标设备进行数据传输操作。

设备驱动程序封装了如何控制设备的细节，它们可以使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和I/O 控制等操作，而驱动程序的主要任务也就是要实现这些系统调用函数。设备驱动程序是一些函数和数据结构的集合，这些函数和数据结构是为实现设备管理的一个简单接口。操作系统内核使用这个接口来请求驱动程序对设备进行I/O操作。

2) 字符设备驱动重要的数据结构

字符设备是以字节为单位逐个进行I/O 操作的设备，不经过系统的 I/O 缓冲区，所以需要管理自己的缓冲区结构。

在字符设备驱动程序中，主要涉及 3 个重要的内核数据结构，分别是`file_operations`、

file和inode, 当用户访问设备文件时, 每个文件的 file 结构都有一个索引节点 inode 与之对应。在内核的inode 结构中, 有一个名为 i_fop 成员, 其类型为 file_operations。file_operations定义文件的各种操作, 用户对文件进行诸如open 、close、read 、write 等操作时,Linux 内核将通过file_operations结构访问驱动程序提供的函数。内核通过这 3 个数据结构的关联, 将用户对设备文件的操作转换为对驱动程序相关函数的调用。

include/linux/fs.h中的数据结构 file_operations如下:

```
struct file_operations {
...

    loff_t (*llseek) (struct file *, loff_t, int);

    ssize_t (*read) (struct file *, char *, size_t, loff_t *);

    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);

    int (*readdir) (struct file *, void *, filldir_t);

    unsigned int (*poll) (struct file *, struct poll_table_struct *);

    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

    int (*open) (struct inode *, struct file *);

    int (*flush) (struct file *);

    int (*release) (struct inode *, struct file *) ;

    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long,
loff_t *);

    ...

};
```

file 代表一个打开了的文件。它由内核在使用 open () 函数时建立, 并传递给该文件上进行操作的所有函数, 直到最后的close() 函数。当文件的所有操作结束后, 内核会释放该数据结构。在file 结构中, 一些比较重要的成员含义如下:

```
struct file{
...

    mode_t f_mode;      /*文件模式, 用于标记文件是否可读或可写*/

    loff_t f_pos;        /*当前的读写位置*/

    file_operations *f_op;/* 与文件相关操作。内核执行 open() 操作时对该指针赋值
*/
```



```
void *private_data; /* 驱动程序可将这个字段用于任何目的，但是要在 release()
方法中，释放该字段占用的主存*/
```

```
}
```

内核用inode 结构在内部标识文件,它和 file 结构不同,后者标识打开的文件描述符。对于单个文件,可能会有许多个表示打开的文件描述符的 file 结构,但它们都指向同一个 inode结构, inode 结构中有两个重要的字段:

```
struct cdev *i_cdev;
```

```
dev_t i_rdev;
```

类型dev_t描述设备号,cdev 结构表示字符设备的内核数据结构,内核不推荐开发者直接通过结构的i_rdev 结构获得主次设备号,而提供下面两个函数取得主、次设备号。

```
Unsigned int iminor(struct inode *inode);
```

```
Unsigned int imajor(struct inode *inode);
```

3) 设备文件的创建

实验中需要对一个设备文件进行操作,这里有两种方案,一是在终端中人工使用命令进行创建,二是直接在模块程序中予以创建,其中第二种方法的代码将在下面的程序中以粗体字呈现。

4) 并发控制

在驱动程序中,当多个线程同时访问相同的资源时(驱动程序中的全局变量是一种典型的共享资源),可能会引发“竞争”,因此必须对共享资源进行并发控制。在此驱动程序中,可用信号量机制来实现并发控制。

首先要包含相应的头文件<asm/semaphore.h>,添加信号量 struct semaphore sem ,接着就是初始化,使用void sema_init(struct semaphore *sem, int val),内核对信号量的P、V 操作通过void down_interruptible(struct semaphore *sem) 和void up(struct semaphore *sem) 完成。在Linux 驱动程序中,可以使用等待队列(wait queue)来实现阻塞操作。等待队列可以用来同步对系统资源的访问,这里所讲述Linux 信号量在内核中也是由等待队列来实现的。参考程序中定义设备的设备为“globalvar”,它可以被多个进程打开,但是每次只有当一个进程写入了一个数据之后本进程或其它进程才可以读取该数据,否则一直阻塞。

6.1.4 实验步骤

1) 相关数据结构与函数

(1) 设备驱动程序结构

字符设备的结构描述如下：

```
struct Scull_Dev{
    struct cdev devm; //字符设备
    struct semaphore sem; //信号量，实现读写时的 PV 操作
    wait_queue_head_t outq; //等待队列，实现阻塞操作
    int flag; //阻塞唤醒条件
    char buffer[MAXNUM+1]; //字符缓冲区
    char *rd,*wr,*end; //读，写，尾指针
};
```

(2) 字符设备的数据接口

字符设备的数据接口将文件的读、写、打开、释放等操作映射为相应的函数。

```
struct file_operations globalvar_fops =
{
    .read=globalvar_read,
    .write=globalvar_write,
    .open=globalvar_open,
    .release=globalvar_release,
};
```

(3) 字符设备的注册与注销

字符设备的注册采用静态申请和动态分配相结合的方式，使用register_chrdev_region函数和alloc_chrdev_region函数来完成。

字符设备的注销采用unregister_chrdev_region函数来完成。

具体实现见下文代码。

(4) 字符设备的打开与释放

打开设备是通过调用 file_operations 结构中的函数 open()来完成的。设备的打开提供给驱动程序初始化的能力，从而为以后的操作准备，此外还会递增设备的使用记数，防止在文件被关闭前被卸载出内核。

释放设备是通过调用 file_operations 结构中的函数 release()来完成的。设备的释放作用

刚好与打开相反，但基本的释放操作只包括设备的使用计数递减。

（5）字符设备的读写操作

字符设备的读写操作相对比较简单，直接使用函数 `read()` 和 `write()` 就可以了。文件读操作的原型如下：

```
Ssize_t device_read(struct file* filp, char __user* buff, size_t len, loff_t* offset);
```

其中，`filp` 是文件对象指针；`buff` 是用户态缓冲区，用来接受读到的数据；`len` 是希望读取的数据量；`offset` 是用户访问文件的当前偏移。

文件写操作的原型和读操作没有区别，只是操作方向改变而已。

由于内核空间与用户空间的内存不能直接互访，因此借助函数 `copy_to_user()` 完成用户空间到内核空间的复制，函数 `copy_from_user()` 完成内核空间到用户空间的复制。

2) 实验步骤

（1）编写模块程序

利用 **linux** 的动态模块，编写的设备程序 `globalvar.c` 其代码如下：

```
#include<linux/module.h>
#include<linux/init.h>
#include<linux/fs.h>
#include<asm/uaccess.h>
#include<linux/wait.h>
#include<linux/semaphore.h>
#include <linux/sched.h>
#include <linux/cdev.h>
#include <linux/types.h>
#include<linux/kdev_t.h>
#include <linux/device.h>

#define MAXNUM 100
#define MAJOR_NUM 456 //主设备号 ， 没有被使用

struct Scull_Dev{
    struct cdev devm; //字符设备
    struct semaphore sem; //信号量，实现读写时的 PV 操作
    wait_queue_head_t outq; //等待队列，实现阻塞操作
    int flag; //阻塞唤醒标志
```

```
char buffer[MAXNUM+1]; //字符缓冲区
char *rd,*wr,*end; //读，写，尾指针
};

struct Scull_Dev globalvar;
static struct class *my_class;
int major=MAJOR_NUM;

static ssize_t globalvar_read(struct file *,char *,size_t ,loff_t *);
static ssize_t globalvar_write(struct file *,const char *,size_t ,loff_t *);
static int globalvar_open(struct inode *inode,struct file *filp);
static int globalvar_release(struct inode *inode,struct file *filp);
struct file_operations globalvar_fops =
{
    .read=globalvar_read,
    .write=globalvar_write,
    .open=globalvar_open,
    .release=globalvar_release,
};

static int  globalvar_init(void)
{
    int result = 0;
    int err = 0;
    dev_t dev = MKDEV(major, 0);
    if(major)
    {
        //静态申请设备编号
        result = register_chrdev_region(dev, 1, "charmем");
    }
    else
    {
        //动态分配设备号
        result = alloc_chrdev_region(&dev, 0, 1, "charmем");
        major = MAJOR(dev);
    }
}
```

```

    }
    if(result < 0)
        return result;
    cdev_init(&globalvar.devm, &globalvar_fops);
    globalvar.devm.owner = THIS_MODULE;
    err = cdev_add(&globalvar.devm, dev, 1);
    if(err)
        printk(KERN_INFO "Error %d adding char_mem device", err);
    else{
        printk("globalvar register  success\n");
        sema_init(&globalvar.sem,1); //初始化信号量
        init_waitqueue_head(&globalvar.outq); //初始化等待队列
        globalvar.rd = globalvar.buffer; //读指针
        globalvar.wr = globalvar.buffer; //写指针
        globalvar.end = globalvar.buffer + MAXNUM; //缓冲区尾指针
        globalvar.flag = 0; // 阻塞唤醒标志置 0
    }
    my_class = class_create(THIS_MODULE, "chardev0");
    device_create(my_class, NULL, dev, NULL, "chardev0");
    return 0;
}

static int globalvar_open(struct inode *inode,struct file *filp)
{
    try_module_get(THIS_MODULE); //模块计数加一
    printk("This chrdev is in open\n");
    return(0);
}

static int globalvar_release(struct inode *inode,struct file *filp)
{
    module_put(THIS_MODULE); //模块计数减一
    printk("This chrdev is in release\n");
    return(0);
}

```

```
static void globalvar_exit(void)
{
    device_destroy(my_class, MKDEV(major, 0));
    class_destroy(my_class);
    cdev_del(&globalvar.devm);
    unregister_chrdev_region(MKDEV(major, 0), 1); //注销设备
}

static ssize_t globalvar_read(struct file *filp, char *buf, size_t len, loff_t *off)
{
    if(wait_event_interruptible(globalvar.outq, globalvar.flag!=0)) //不可读时 阻塞读进程
    {
        return -ERESTARTSYS;
    }
    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    globalvar.flag = 0;
    printk("into the read function\n");
    printk("the rd is %c\n", *globalvar.rd); //读指针
    if(globalvar.rd < globalvar.wr)
        len = min(len, (size_t)(globalvar.wr - globalvar.rd)); //更新读写长度
    else
        len = min(len, (size_t)(globalvar.end - globalvar.rd));
    printk("the len is %d\n", len);
    if(copy_to_user(buf, globalvar.rd, len))
    {
        printk(KERN_ALERT "copy failed\n");
        up(&globalvar.sem);
        return -EFAULT;
    }
    printk("the read buffer is %s\n", globalvar.buffer);
    globalvar.rd = globalvar.rd + len;
}
```

```
    if(globalvar.rd == globalvar.end)
        globalvar.rd = globalvar.buffer; //字符缓冲区循环
    up(&globalvar.sem); //V 操作
    return len;
}

static ssize_t globalvar_write(struct file *filp, const char *buf, size_t len, loff_t *off)
{
    if(down_interruptible(&globalvar.sem)) //P 操作
    {
        return -ERESTARTSYS;
    }
    if(globalvar.rd <= globalvar.wr)
        len = min(len, (size_t)(globalvar.end - globalvar.wr));
    else
        len = min(len, (size_t)(globalvar.rd - globalvar.wr - 1));
    printk("the write len is %d\n", len);
    if(copy_from_user(globalvar.wr, buf, len))
    {
        up(&globalvar.sem); //V 操作
        return -EFAULT;
    }
    printk("the write buffer is %s\n", globalvar.buffer);
    printk("the len of buffer is %d\n", strlen(globalvar.buffer));
    globalvar.wr = globalvar.wr + len;
    if(globalvar.wr == globalvar.end)
        globalvar.wr = globalvar.buffer; //循环
    up(&globalvar.sem); //V 操作
    globalvar.flag = 1; //条件成立，可以唤醒读进程
    wake_up_interruptible(&globalvar.outq); //唤醒读进程
    return len;
}

module_init(globalvar_init);
module_exit(globalvar_exit);
```

```
MODULE_LICENSE("GPL");
```

(2) 同目录下建立 Makefile 文件

为了便于对其上述设备驱动程序 `globalvar.c`（模块源程序）进行编译，建立 `Makefile` 文件，代码如下：

```
ifneq ($(KERNELRELEASE),)
obj-m := globalvar.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean
```

(3) 编译模块程序

利用 `Makefile` 对字符设备驱动程序 `globalvar.c` 进行编译，编译后在当前目录生成 `globalvar.ko`

```
#make
```

(4) 加载模块

```
#insmod globalvar.ko
```

(5) 创建设备节点

如果使用的是第一种方法（不带上述程序里的斜体字代码部分），则需要执行下面的语句创建此设备文件；如果使用的是第二种自动创建设备文件的方法，则在模块加载后无需这一步。

```
#mknod /dev/chardev0 c 456 0
```

(6) 编写读写测试程序 `read.c` 和 `write.c`

为了验证字符设备驱动程序 `globalvar.c` 的可用性，编写用户程序 `read.c` 和 `write.c`，以测试对字符设备的读与写。其读程序 `read.c` 和写程序 `write.c` 代码如下。

read.c:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
```



```
#include<fcntl.h>
#include<string.h>
main(void)
{
    int fd,i;
    char num[101];
    fd= open("/dev/charm0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            for(i=0;i<101;i++)
                num[i]='\0';
            read(fd,num,100);
            printf("%s\n",num);
            if(strcmp(num,"quit")==0)
            {
                close(fd);
                break;
            }
        }
    }
    else
    {
        printf("device open failure,%d\n",fd);
    }
}
```

write.c:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<string.h>
```

```
main()
{
    int fd;
    char num[100];

    fd= open("/dev/charm0",O_RDWR,S_IRUSR|S_IWUSR);
    if(fd!=-1)
    {
        while(1)
        {
            printf("Please input the global:\n");
            scanf("%s",num);
            write(fd,num,strlen(num));
            if(strcmp(num,"quit")==0)
            {
                close(fd);
                break;
            }
        }
    }
    else
    {
        printf("device open failure\n");
    }
}
```

(7) 编译测试程序

```
#gcc read.c -o read
```

```
#gcc write.c -o write
```

(8) 打开两个终端同时运行读写测试程序

```
#./read
```

```
#./write
```

运行效果如图 6.1 所示。

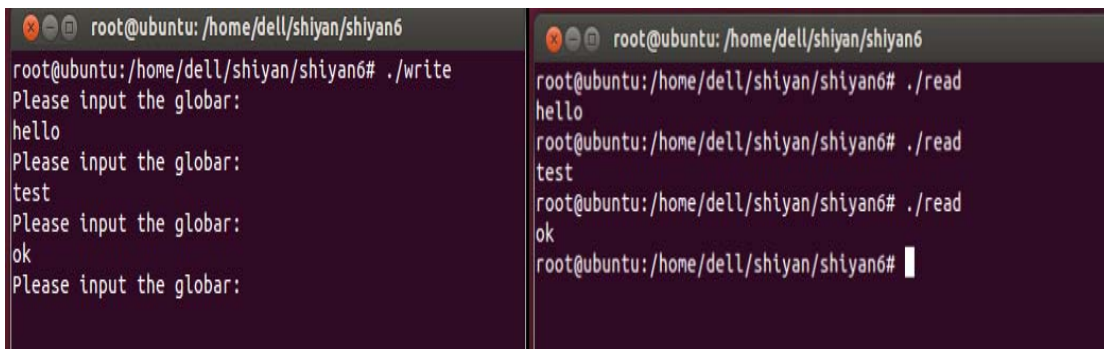


图 6.1 字符设备测试截图

(9) 删除设备节点

使用第二种方法创建设备文件时，也无需此步骤。

```
#rm /dev/chardev0
```

(10) 卸载模块

```
#rmmod globalvar
```

6.2 键盘指示灯驱动

6.2.1 实验目的

了解键盘指示灯的驱动原理，为以后实现更复杂的硬件驱动奠定基础。

6.2.2 实验内容

编写一个简单的键盘指示灯驱动程序。该驱动并不是真正的键盘指示灯驱动，而是用字符设备文件模拟键盘，用内存数据模拟键盘扫描码，在输入匹配正确的情况下，驱动键盘指示灯亮或灭。

6.2.3 实验原理

1) 键盘指示灯的驱动过程

键盘指示灯属于键盘的一部分，其驱动也在键盘中断服务例程中实现。例如，当在键盘上敲下 Caps Lock 键，中断控制器接收到键盘的中断请求，向 CPU 发出中断申请信号，并为 CPU 提供中断类型号，使得 CPU 能够计算出发出中断请求设备的中断服务程序入口地址指针，进而进入键盘中断服务例程。键盘中断服务例程根据扫描码进入相应的处理程序，通过键盘端口地址写操作实现对指示灯的控制。

2) 实现方式

(1) 字符设备的描述结构

```
struct Scull_dev{  
    struct cdev mydev;  
    char buffer[2];  
};
```

其中，mydev 模拟键盘设备文件，buffer 为数据缓冲区。

(2) 字符设备的数据接口

```
struct file_operations dev_op={
    .write=dev_write,
    .open=dev_open,
    .release=dev_release,
};
```

其中，file_operations 结构中的函数 open() 来完成设备的打开，设备的打开提供给驱动程序初始化的能力，从而为以后的操作准备，此外还会递增设备的使用记数，防止在文件被关闭前被卸载出内核。

file_operations 结构中的函数 release() 来完成设备的释放。设备的释放作用刚好与打开相反。但基本的释放操作只包括设备的使用计数递减。

write() 函数来完成对字符设备的写操作。文件读操作的原型如下：

```
static size_t dev_write(struct file *filp,char *buf,size_t len,loff_t *off)
```

其中，filp 是文件对象指针；buf 是用户态缓冲区，用来接受读到的数据；len 是希望读取的数据量；off 是用户访问文件的当前偏移。

(3) 内核空间 and 用户空间的数据拷贝

在本次实验中，字符设备文件的数据需要从用户输入获取，但内核空间与用户空间的内存不能直接互访，因此借助函数 copy_from_user() 完成用户空间到内核空间的复制。copy_from_user() 函数原型如下：

```
unsigned long copy_from_user (void __user *to, const void *from, unsigned long n);
```

其中，to 是内核空间的指针，from 是用户空间指针，n 表示从内核空间向用户空间拷贝数据的字节数。

(4) 端口写操作

本次实验使用 linux 内核提供的函数嵌入汇编写端口命令，完成对键盘端口的写操作，最终实现对键盘指示灯的驱动。分配给键盘控制器的 IO 端口范围是 0x60-0x6f,但实际上使用的只有 0x60 和 0x6f 两个端口地址。实验中会用到 0x60 端口，发送键盘命令。对指示灯控制，使用命令码 0xed 实现。该命令码需要使用参数控制三个指示灯，所以还需再对该端口地址发送参数。

键盘指示灯由一个字节表示，其中位 7-3 保留全为 0，位 2 表示 caps-lock 键，位 1 表示 num-lock 键，位 0 表示 scroll-lock 键。例如，如果驱动 caps-lock 灯亮，参数则为 0x04。

下面是键盘端口写操作的内嵌汇编指令。

```
asm volatile("outb %0,%1" :: "a" (v), "dN" (port));
```

其中，%0, %1 均代表寄存器。

"a", "dN" 为操作数限定字符，在 asm 语句中对硬件寄存器的引用必须用""作为寄存器名称的前缀。

v 是 8 位的命令码/参数, port 是 16 位的端口地址。

6.2.4 实验步骤

1) 编写键盘指示灯 “led_driver.c” 模块程序

“led_driver.c” 模块源程序如下:

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/cdev.h>
#include<linux/init.h>
#include<linux/kdev_t.h>
#include<linux/types.h>
#include<linux/sched.h>
#include<linux/fs.h>
#include <asm/uaccess.h>

#define MAJOR_NUM 444;
int major=MAJOR_NUM;

struct Scull_dev{
struct cdev mydev;
char buffer[2];
};

struct Scull_dev sdev;

static size_t dev_write(struct file *,char *,size_t,loff_t *);
static int dev_open(struct inode*,struct file *filp);
static int dev_release(struct inode*,struct file *filp);

struct file_operations dev_op={
.write=dev_write,
.open=dev_open,
.release=dev_release,
};
//向键盘控制器端口发送命令
static inline void my_outb(u8 v, u16 port)
{
asm volatile("outb %0,%1" : : "a" (v), "dN" (port));
}

static int __init dev_init(void)
{
int result=0;
```

```
int error=0;
//将主设备号和次设备号转换成 dev_t 类型
dev_t dev=MKDEV(major,0);
if(major){
//为字符驱动获取一个或多个设备编号
result=register_chrdev_region(dev,1,"mydev");
}
else{
result=alloc_chrdev_region(&dev,0,1,"mydev");
}
if(result<0) return result;
//设备初始化
cdev_init(&sdev.mydev,&dev_op);
sdev.mydev.owner=THIS_MODULE;
//向系统注册设备
error=cdev_add(&sdev.mydev,dev,1);
if(error)
    printk(KERN_INFO "Error %d adding my char device", error);
else printk("adding my char device succesful\n");
return 0;
}

static void __exit dev_exit(void)
{
//注销设备
cdev_del(&sdev.mydev);
//释放设备号
unregister_chrdev_region(MKDEV(major,0),1);
}

static int dev_open(struct inode *inode,struct file *filp)
{
//递增设备使用计数
try_module_get(THIS_MODULE);
printk("this device is in open!\n");
return 0;
}

static int dev_release(struct inode *inode,struct file *filp)
{
//递减设备使用计数
module_put(THIS_MODULE);
printk("this device is in release!\n\n");
return 0;
}
```

```

}

static size_t dev_write(struct file *filp, char *buf, size_t len, loff_t *off)
{
//实现用户空间到内核空间的数据拷贝
if(copy_from_user(sdev.buffer, buf, len))
    return -EFAULT;
if(sdev.buffer[0]=='l')
{
    my_outb(0xed, 0x0060);
    my_outb(0x07, 0x0060);
}
else if(sdev.buffer[0]=='f')
{
    my_outb(0xed, 0x0060);
    my_outb(0x00, 0x0060);
}
printf("you had written %s to the sdev.\n", sdev.buffer);
return len;
}

```

```

MODULE_LICENSE("GPL");
module_init(dev_init);
module_exit(dev_exit);

```

2) 编写键盘指示灯 “led_driver.c” 模块程序

为了便于对其上述设备驱动程序 led_driver.c（模块源程序）进行编译，建立 Makefile 文件，代码如下：

```

ifneq ($(KERNELRELEASE),)
obj-m := led_driver.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
$(MAKE) -C $(KERNELDIR) M=$(PWD) clean

```

3) 加载 led_driver 模块

```
insmod led_driver.ko
```

4) 创建设备节点

```
mknod /dev/mydev c 444 0
```

5) 编写测试 test.c 程序

test.c 源程序如下所示:

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>

int main()
{
    int fd;
    char buf[2];
    fd= open("/dev/mydev",O_RDWR);
    if(fd!=-1)
    {
        while(1){
            printf("turn on:l;turn off:f\n");
            printf("input:");
            scanf("%s",buf);
            write(fd,buf,1);
        }
    }
    else
    {
        printf("device open failure\n");
        return -1;
    }
    close(fd);
    return 0;
}
```

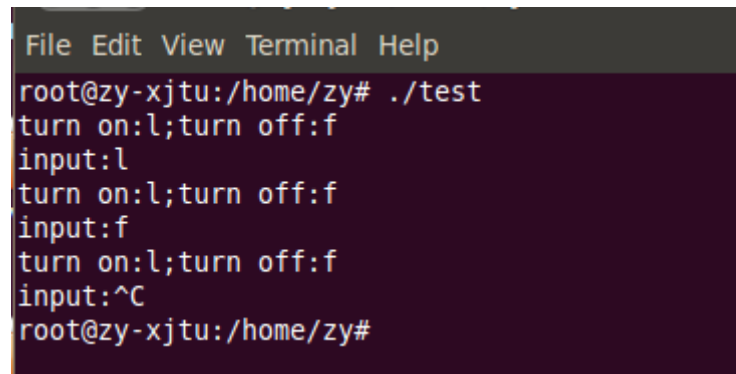
6) 编译测试程序

```
gcc -o test test.c
```

7) 运行 test 查看结果

```
./test
```

测试过程如图 6.2 所示:



```
File Edit View Terminal Help
root@zy-xjtu:/home/zy# ./test
turn on:l;turn off:f
input:l
turn on:l;turn off:f
input:f
turn on:l;turn off:f
input:^C
root@zy-xjtu:/home/zy#
```

图 6.2 test 运行界面

根据提示输入 l 时，键盘指示灯亮；

输入 f 时，键盘指示灯灭。

8) 删除设备节点

```
rm /dev/mydev
```

9) 卸载模块

```
rmmod led_driver
```

除了控制键盘指示灯，也可以利用设备驱动来模拟控制蜂鸣器，读者可查阅相关资料予以实现。

实验七 一个最小操作系统的实现

7.1 引导启动程序

7.1.1 实验目的

编程实现一段简单的引导启动程序，编译后能够从 U 盘启动执行。

7.1.2 实验内容

使用 GNU as 汇编语言编写引导启动程序 boot.s，该程序的主要功能为启动时在屏幕上显示字符串“Loading system ...”。

7.1.3 实验原理

PC 机的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 处开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后 BIOS 会把启动盘上第一个扇区（磁盘引导扇区）加载到物理内存 0x7c00 位置开始处，并把执行权转移到 0x7c00 处开始运行引导程序代码。引导启动程序通常负责把操作系统内核程序加载到物理内存合适的位置中，最后转移至这个合适的位置开始执行操作系统代码。

7.1.4 实验步骤

本实现采用 GNU as 汇编语言，GNU as 汇编的基础知识请参考相关资料。

步骤一：编写 boot.s 程序。

boot.s 程序的代码如下：

```
/* GNU as 汇编语言编制的 512B 引导扇区代码
 * 运行在 16 位实地址模式下
 */
BOOTSEG = 0x07c0 # BIOS 加载引导扇区代码的原始段地址
.code16
.section .text
.globl _start
_start:
    ljmp $BOOTSEG, $go # 段间跳转，执行之后 cs=0x07C0
go:    movw %cs, %ax
        movw %ax, %ds
        movw %ax, %es
        movb %ah, load_msg+17 # 替换字符串中最后一个点符号，响铃一次
        movw $20, %cx # 共显示 20 个字符，包括回车和换行
```

```

    movw $0x1004, %dx  # 字符串将显示在屏幕的第 17 行，第 5 列处
    movw $0x000c, %bx  # 字符显示属性(红色)
    movw $load_msg, %bp # 指向要显示的字符串
    movw $0x1301, %ax
    int $0x10  # BIOS 中断调用 0x10, 功能 0x13, 子功能 01
loop:    jmp loop  # 无限循环
load_msg:    .ascii "Loading system ..."
            .byte 13,10
.org 510
boot_flag:
    .word 0xAA55  # 有效引导扇区标志，供 BIOS 加载引导扇区使用

```

步骤二：编写 Makefile 文件。

Makefile 文件内容如下：

```

# Makefile for the boot
AS =as
LD =ld

all: Image

Image: boot
    dd bs=512 if=boot of=Image
    sync

boot:    boot.s
    $(AS) -o boot.o boot.s
    $(LD) --oformat binary -Ttext 0x0 -o boot boot.o

clean:
    rm -f Image boot *.o

```

步骤三：编译源程序。

编译环境：gcc 4.4.3。

执行命令：

```
make
```

即可生成引导启动程序映像 Image，如图 7-1 所示。

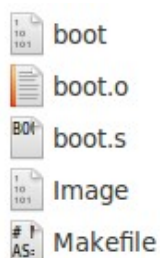


图 7-1 编译生成的 Image 文件

步骤四：将 Image 写入 U 盘的第一个物理扇区。

这里使用已注册版本的 winHex 工具。注意：请先备份 U 盘上的重要资料，采用这种方法向 U 盘的物理扇区写数据会破坏 U 盘扇区中原来的内容。

1) 选择 Tools->Open Disk 选择物理 U 盘设备，如图 7-2 所示。

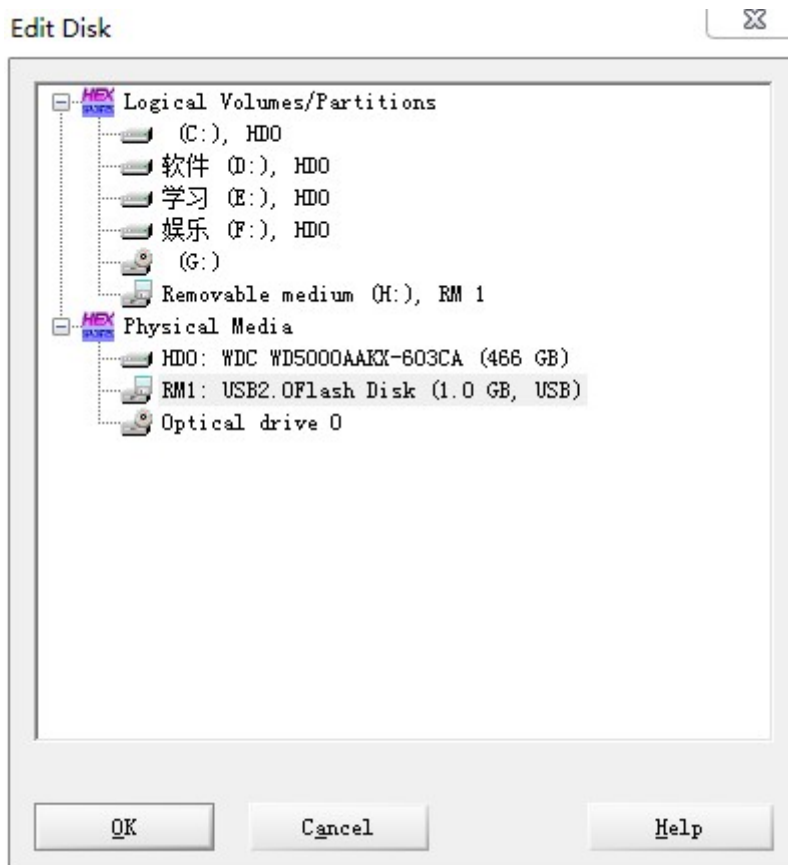


图 7-2 打开物理 U 盘设备

2) 选择 File->Open，浏览到 Image 文件打开。

3) 将 Image 文件的值复制到 U 盘的起始位置后保存。这样 U 盘的第一个扇区的值即修改为 Image 中的值。

步骤五：从 U 盘启动并进入引导启动程序。

首先要确保 PC 主板支持从 U 盘启动，具体可以进入 PC BIOS 中查看。一些老式 PC 主板可能不支持从 U 盘启动，请换一台 PC 机重试。

从 U 盘启动后既可以看到屏幕上打印出的字符串“Loading system ..”，并可以听到响铃一次。最后一个点号已经被响铃替代。如图 7-3 所示。

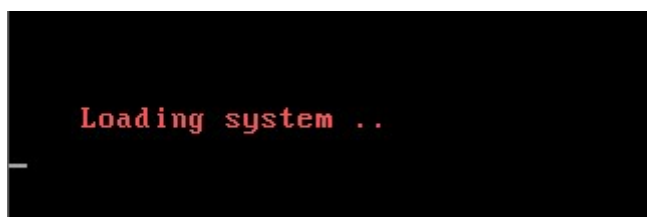


图 7-3 引导启动界面

7.2 实现一个简单的多任务操作系统

7.2.1 实验目的

实现一个运行在保护模式下的简单的多任务的操作系统。

7.2.2 实验内容

编写一个引导启动程序boot.s和简单多任务内核程序head.s。

boot.s用于在计算机加电时从启动盘上把多任务内核代码加载到内存中。

head.s实现两个运行在特权级3上的任务在时钟中断控制下相互切换运行，还实现一个在屏幕上显示字符的系统调用。

7.2.3 实验原理

PC机加电后BIOS会把启动盘上第一个扇区（磁盘引导扇区）加载到物理内存0x7c00位置开始处，并把执行权转移到0x7c00处开始运行引导程序代码。boot程序的主要功能是把引导盘中的内核代码head加载到内存的某个指定位置处，并在设置好临时GDT表等信息后，把处理器设置成运行在保护模式下，然后跳转到head代码处去运行内核代码。

boot.s程序编译出的代码共512B，将放在引导盘映像的第一个扇区中。

head.s程序编译的代码放在引导盘从第二个扇区开始的扇区。boot代码和head代码在启动盘上的分布如图7-4所示。

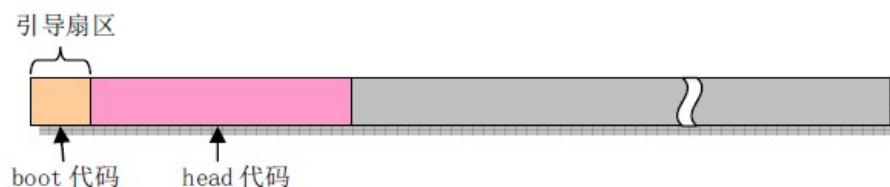


图7-4 启动盘上代码分布图

实际上boot.s程序首先会利用BIOS中断int 0x13把启动盘中的head代码读入到内存0x10000位置开始处，然后把这段head代码移动到内存0开始处。最后设置控制寄存器CR0中的开启保护运行模式标志，并跳转到内存0处开始执行head代码。boot程序代码在内存中移

动head代码的示意图如图7-5所示。

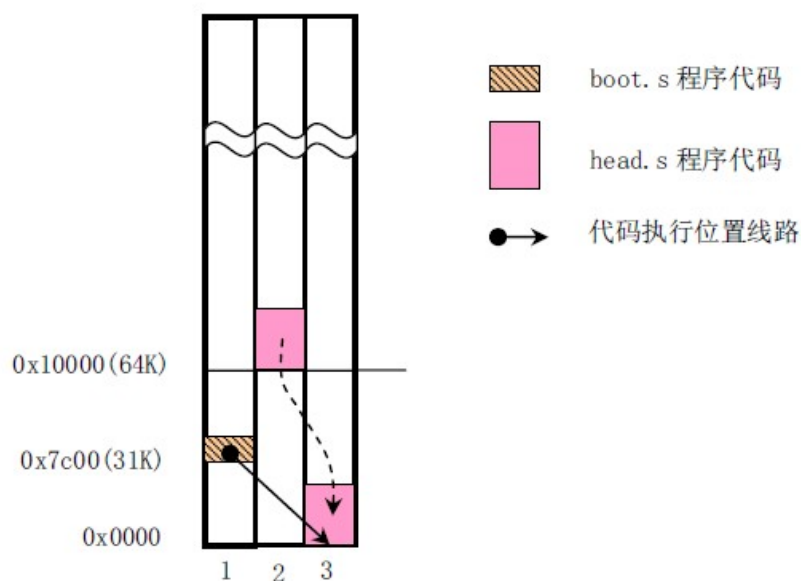


图7-5 内核代码在物理内存中的移动图

把head内核代码移动到物理内存0开始处的主要原因是为了设置GDT表时可以简单一些，也能让head.s程序尽量短一些。但是我们不能让boot程序把head代码从启动盘直接加载到内存0处，因为加载操作需要使用BIOS提供的中断过程，而BIOS使用的中断向量表正处在内存0开始的地方，并且在内存1KB开始处是BIOS程序使用的数据区，所以若直接把head代码加载到内存0处将使得BIOS中断过程不能正常运行。当然，我们也可以把head代码加载到内存0x10000处后就直接跳转到该处运行head代码，你可以尝试修改boot.s程序代码来达到这种目的。

7.2.4 实验步骤

本实验 boot.s 和 head.s 均采用 GNU as 汇编语言。

步骤一：编写引导启动程序 boot.s。

boot.s 程序的代码如下：

/* GNU as 格式的 boot.s 程序

* 首先利用 BIOS 中断把内核代码(head.s 编译后的代码)加载到内存 0x10000 处，

* 然后移动到内存 0 处，最后进入保护模式，并跳转到内存 0 处继续运行

*/

BOOTSEG = 0x07c0

SYSSEG = 0x1000

内核先加载到 0x10000 处，然后移动到 0x0 处

SYSLEN = 17

内核占用的最大磁盘扇区数

.code16

```

.section .text
.globl _start
_start:
    ljmp    $BOOTSEG, $go    # 段间跳转，当本程序刚开始运行时，所有段寄存器值
go: movw    %cs, %ax         # 均为 0。该跳转语句会把 cs 寄存器加载为 0x07c0
    movw    %ax, %ds
    movw    %ax, %ss
    movw    $0x400, %sp      # 临时栈指针

```

加载内核代码到内存 0x10000 处

```

load_system:
    movw    $0x0080, %dx    # 如果从软盘启动 0x0000
    movw    $0x0002, %cx
    movw    $SYSSEG, %ax
    movw    %ax, %es
    xorw    %bx, %bx
    movw    $0x200+SYSLN, %ax
    int     $0x13           # 利用 BIOS 中断 int 0x13 功能 2 从启动盘读取 head 代码
    jnc ok_load
die:      jmp die

```

把内核代码移动到内存 0 开始处，共移动了 16 个扇区

```

ok_load:
    cli          # 关闭中断
    movw    $SYSSEG, %ax
    movw    %ax, %ds
    xorw    %ax, %ax
    movw    %ax, %es
    movw    $0x2000, %cx
    subw    %si, %si
    subw    %di, %di
    rep
    movsw

```

加载 IDT 和 GDT 基址寄存器 IDTR 和 GDTR

```

    movw    $BOOTSEG, %ax
    movw    %ax, %ds
    lidt    idt_48          # 加载 IDTR。6 字节操作数
    lgdt    gdt_48          # 加载 GDTR。6 字节操作数

```

设置控制寄存器 CR0 (机器状态字)，进入保护模式

```

    movw    $0x0001, %ax
    lmsw    %ax
    ljmp    $8, $0          # 此时段值已经是段选择子。注意与实模式的区别

```

下面是全局描述符表 GDT 的内容。包含 3 个段描述符

第一个不用，另外两个是代码段和数据段描述符

```
gdt:    .word    0,0,0,0 # 段描述符 0，不用
```

```
    .word    0x07FF      # 段描述符 1。8Mb - 段限长=2047
```

```
    .word    0x0000      # 段基地址
```

```
    .word    0x9A00      # 代码段，可读/可执行
```

```
    .word    0x00C0      # 段属性颗粒度=4KB
```

```
    .word    0x07FF      # 段描述符 2。8Mb - 段限长=2047
```

```
    .word    0x0000      # 段基地址
```

```
    .word    0x9200      # 数据段，可读/可写
```

```
    .word    0x00C0      # 段属性颗粒度=4KB
```

LIDT 和 LGDT 指令的 6 字节操作数

```
idt_48: .word    0
```

```
    .word    0,0
```

```
gdt_48: .word    0x7ff
```

```
    .word    0x7c00+gdt,0
```

```
.org 510
```

```
boot_flag:
```

```
    .word    0xAA55 # 引导扇区有效标志，必须处于引导扇区最后两个字节
```

步骤二：编写多任务内核程序 head.s。

head.s 程序的代码入下：

```
/* GNU as 格式的多任务内核程序 head.s
```

```
* 包含 32 位保护模式初始化设置代码，时钟中断代码，系统调用中断代码和
```

```
* 两个运行在特权级 3 上任务的代码。
```

```
* 初始化完成之后移动到任务 0 开始执行，并在时钟中断控制下进行任务 0
```

```
* 和任务 1 之间的切换
```

```
*/
```

```
LATCH      = 11930 # 定时器初始计数值，即每隔 10ms 发送一次中断请求
```

```
SCRN_SEL   = 0x18  # 屏幕显示内存段选择子
```

```
TSS0_SEL   = 0x20  # 任务 0 的 TSS 段选择子
```

```
LDT0_SEL   = 0x28  # 任务 0 的 LDT 段选择子
```

```
TSS1_SEL   = 0x30  # 任务 1 的 TSS 段选择子
```

```
LDT1_SEL   = 0x38  # 任务 1 的 LDT 段选择子
```

```
.global startup_32
```

```
.text
```

```
startup_32:
```



```
# 加载 DS, SS, ESP
    movl $0x10, %eax
    mov %ax, %ds
    lss init_stack, %esp

# 在新的位置重新设置 IDT 和 GDT 表
    call setup_idt    # 设置 IDT
    call setup_gdt    # 设置 GDT
    movl $0x10, %eax    # 重新加载所有段寄存器
    mov %ax, %ds
    mov %ax, %es
    mov %ax, %fs
    mov %ax, %gs
    lss init_stack, %esp

# 设置 8253 定时芯片
    movb $0x36, %al
    movl $0x43, %edx
    outb %al, %dx
    movl $LATCH, %eax    # 频率 100HZ
    movl $0x40, %edx
    outb %al, %dx
    movb %ah, %al
    outb %al, %dx

# 在 IDT 表的第 8 和第 128(0x80) 项处分别设置定时中断门描述符
# 和系统调用陷阱门描述符
    movl $0x00080000, %eax
    movw $timer_interrupt, %ax
    movw $0x8E00, %dx    # 中断门类型是 14, 特权级 0
    movl $0x08, %ecx
    lea idt(, %ecx, 8), %esi
    movl %eax, (%esi)
    movl %edx, 4(%esi)
    movw $system_interrupt, %ax
    movw $0xef00, %dx    # 陷阱门类型是 15, 特权级 3 的程序可执行
    movl $0x80, %ecx
    lea idt(, %ecx, 8), %esi
    movl %eax, (%esi)
    movl %edx, 4(%esi)

# 为移动到任务 0 中执行来操作堆栈内容, 在堆栈中人工建立中断返回时的现场
    pushfl
```

```

    andl $0xffffbfff, (%esp)    # 复位 EFLAGS 中的嵌套任务标志
    popfl
    movl $TSS0_SEL, %eax
    ltr %ax                     # 把任务 0 的 TSS 段选择子加载到任务寄存器 TR
    movl $LDT0_SEL, %eax
    lldt %ax                    # 把任务 0 的 LDT 段选择子加载到 LDTR
    movl $0, current            # 把当前任务号 0 保存在 current 变量中
    sti                          # 开启中断，并在栈中营造中断返回时的现场
    pushl $0x17                 # 把任务 0 的堆栈段选择子入栈(SS)
    pushl $init_stack           # 把堆栈指针入栈(ESP)
    pushfl                      # 标志寄存器入栈(EFLAGS)
    pushl $0x0f                 # 当前代码段选择子入栈(CS)
    pushl $task0                # 代码指针入栈(EIP)
    iret                        # 执行中断返回指令，切换到特权级 3 的任务 0 中执行

/*****设置 GDT 和 IDT 中描述符项的子程序*****/
setup_gdt:
    lgdt lgdt_opcode
    ret

setup_idt:
    lea ignore_int, %edx
    movl $0x00080000, %eax
    movw %dx, %ax
    movw $0x8E00, %dx
    lea idt, %edi
    mov $256, %ecx
rp_sidt:
    movl %eax, (%edi)
    movl %edx, 4(%edi)
    addl $8, %edi
    dec %ecx
    jne rp_sidt
    lidt lidt_opcode
    ret

# 显示字符子程序。取当前光标位置并把 AL 中的字符显示在屏幕上。整屏可显示 80x25
# 个字符
write_char:
    push %gs
    pushl %ebx
    mov $SCRN_SEL, %ebx        # GS 指向显示内存段
    mov %bx, %gs
    movl scr_loc, %ebx         # 从变量 scr_loc 中取目前字符显示位置值

```

```

    shl $1, %ebx
    movb %al, %gs:(%ebx)
    shr $1, %ebx
    incl %ebx
    cmpl $2000, %ebx      # 下一个显示位置若大于 2000，复位成 0
    jb 1f
    movl $0, %ebx
1:  movl %ebx, scr_loc
    popl %ebx
    pop %gs
    ret

/*****以下是三个中断处理程序*****/
/* ignore_int 是默认的中断处理程序，若系统产生了其他中断，
 * 会在屏幕上显示字符'C'
 */
.align 2
ignore_int:
    push %ds
    pushl %eax
    movl $0x10, %eax
    mov %ax, %ds
    movl $67, %eax        /* 显示字符 'C' */
    call write_char
    popl %eax
    pop %ds
    iret

/* 定时中断处理程序，主要任务是执行任务切换操作 */
.align 2
timer_interrupt:
    push %ds
    pushl %eax
    movl $0x10, %eax
    mov %ax, %ds
    movb $0x20, %al
    outb %al, $0x20
    movl $1, %eax        # 判断当前任务，若是任务 1 则去执行任务 0，或反之
    cmpl %eax, current
    je 1f
    movl %eax, current    # 若当前任务是 0，则 current=1，并跳转到任务 1 去执行
    ljmp $TSS1_SEL, $0
    jmp 2f
1:  movl $0, current

```

```

        ljmp $TSS0_SEL, $0
2:  popl %eax
    pop %ds
    iret

/* 系统调用中断 int 0x80 处理程序，该示例只有一个显示字符的功能 */
.align 2
system_interrupt:
    push %ds
    pushl %edx
    pushl %ecx
    pushl %ebx
    pushl %eax
    movl $0x10, %edx
    mov %dx, %ds
    call write_char
    popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    pop %ds
    iret

/*****/
current:.long 0    # 当前任务号(0 或 1)
scr_loc:.long 0    # 屏幕当前显示位置。按左上角到右下角顺序显示

.align 2
lidt_opcode:
    .word 256*8-1
    .long idt
lgdt_opcode:
    .word (end_gdt-gdt)-1
    .long gdt

    .align 8
idt:    .fill 256, 8, 0    # IDT 表空间

gdt:    .quad 0x0000000000000000    # GDT 表。第 1 个描述符不用
        .quad 0x00c09a00000007ff    # 第 2 个是内核代码段描述符。选择子是 0x08
        .quad 0x00c09200000007ff    # 第 3 个是内核数据段描述符。选择子是 0x10
        .quad 0x00c0920b80000002    # 第 4 个是显示内存段描述符。选择子是 0x18

        .word 0x0068, tss0, 0xe900, 0x0 # 第 5 个是 TSS0 段描述符。选择子是 0x20

```

```

        .word 0x0040, ldt0, 0xe200, 0x0 # 第 6 个是 LDT0 段描述符。选择子是 0x28
        .word 0x0068, tss1, 0xe900, 0x0 # 第 7 个是 TSS1 段描述符。选择子是 0x30
        .word 0x0040, ldt1, 0xe200, 0x0 # 第 8 个是 LDT1 段描述符。选择子是 0x38
end_gdt:
        .fill 128, 4, 0                # 初始内核堆栈空间
init_stack:
        .long init_stack
        .word 0x10

# 下面是任务 0 的 LDT 表段中的局部描述符
.align 8
ldt0:    .quad 0x0000000000000000 # 第 1 个描述符不用
        .quad 0x00c0fa00000003ff # 第 2 个是局部代码段描述符，选择子 0x0f
        .quad 0x00c0f200000003ff # 第 3 个是局部数据段描述符，选择子 0x17

# 下面是任务 0 的 TSS 段内容
tss0:    .long 0                    /* 前一任务链 */
        .long krn_stk0, 0x10        /* esp0, ss0 */
        .long 0, 0, 0, 0, 0        /* esp1, ss1, esp2, ss2, cr3 */
        .long 0, 0, 0, 0, 0        /* eip, eflags, eax, ecx, edx */
        .long 0, 0, 0, 0, 0        /* ebx esp, ebp, esi, edi */
        .long 0, 0, 0, 0, 0, 0     /* es, cs, ss, ds, fs, gs */
        .long LDT0_SEL, 0x8000000  /* ldt, I/O 位图基地址 */

        .fill 128, 4, 0            # 任务 0 的内核空间
krn_stk0:

# 下面是任务 1 的 LDT 表段内容和 TSS 段内容
.align 8
ldt1:    .quad 0x0000000000000000
        .quad 0x00c0fa00000003ff
        .quad 0x00c0f200000003ff

tss1:    .long 0
        .long krn_stk1, 0x10
        .long 0, 0, 0, 0, 0
        .long task1, 0x200
        .long 0, 0, 0, 0
        .long usr_stk1, 0, 0, 0
        .long 0x17, 0x0f, 0x17, 0x17, 0x17, 0x17
        .long LDT1_SEL, 0x8000000

        .fill 128, 4, 0            # 任务 1 的内核空间，其用户栈直接使用初始栈空间
krn_stk1:

```

下面是任务 0 和任务 1 的子程序，分别循环显示字符 A 和 B

task0:

```
    movl $0x17, %eax
    movw %ax, %ds
    movb $65, %al          /* 显示字符 'A' */
    int $0x80
    movl $0xffff, %ecx
```

```
1:  loop lb
    jmp task0
```

task1:

```
    movl $0x17, %eax
    movw %ax, %ds
    movb $66, %al          /* 显示字符 'B' */
    int $0x80
    movl $0xffff, %ecx
```

```
1:  loop lb
    jmp task1
```

```
.fill 128, 4, 0      # 任务 1 的用户栈空间
```

usr_stk1:

步骤三：编写 Makefile 文件。

Makefile 文件的内容如下：

Makefile for the simple example kernel.

AS =as

LD =ld

LDFLAGS =--oformat binary -Ttext 0x0

all: Image

Image: boot system

```
    dd bs=512 if=boot of=Image
```

```
    dd bs=1 if=system of=Image seek=512
```

```
    sync
```

system: head.s

```
    $(AS) -o head.o head.s
```

```
    $(LD) $(LDFLAGS) -o system head.o
```

boot: boot.s

```
    $(AS) -o boot.o boot.s
```

```
    $(LD) $(LDFLAGS) -o boot boot.o
```

clean:

```
rm -f Image core boot head *.o system
```

步骤四：编译源程序。

编译环境：gcc 4.4.3。

执行命令：

```
make
```

即可生成引导启动程序映像 Image，如图 7-6 所示。

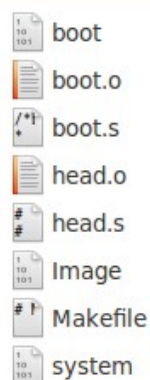


图 7-6 编译生成的文件

步骤五：将 Image 写入 U 盘的第一个物理扇区开始处。注意：请先备份 U 盘上的重要资料，采用这种方法向 U 盘的物理扇区写数据会破坏 U 盘扇区中原来的内容。

写入步骤同 7.1.4 小节步骤四。

步骤六：从 U 盘启动并进入引导启动程序。

步骤同 7.1.4 小节步骤五。

从 U 盘启动后既可以看到一连串的字符“A”和一连串的字符“B”间隔地连续不断地显示在屏幕上，如图 7-7 所示。

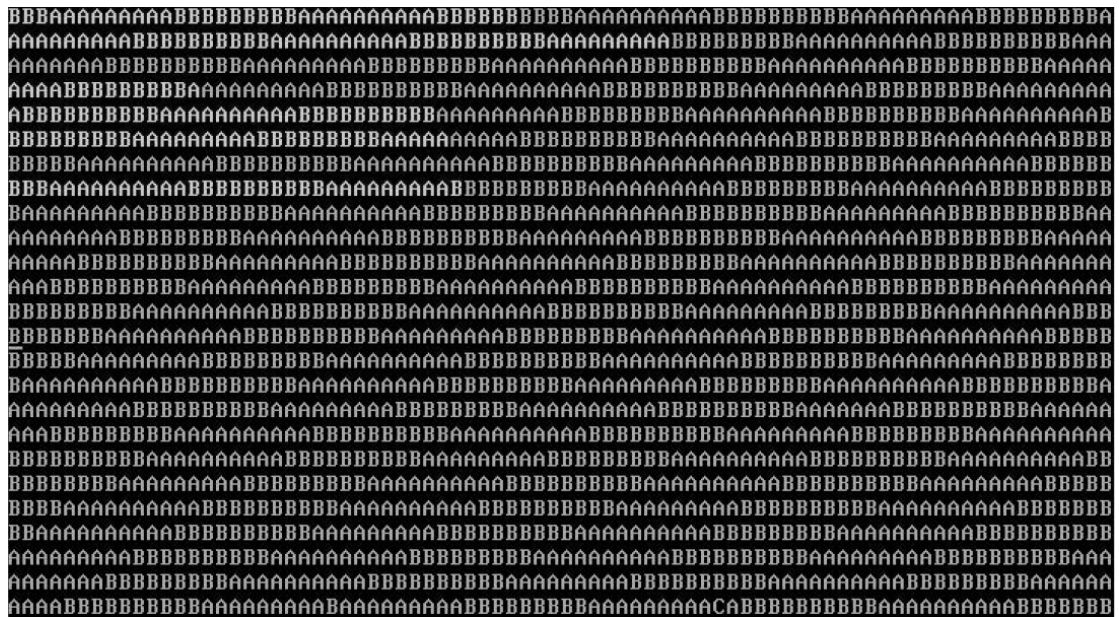


图 7-7 简单多任务内核运行屏幕显示情况

实验八 内核空间的内存映射实验

8.1 实验目的

通过将内核空间中的物理地址空间映射到用户虚拟空间,了解内核中 `vmalloc` 和 `kmalloc` 两种内存申请方法中内核虚拟区域与实际物理区域映射方式的不同。

8.2 实验内容

设计一个内存映射模块,模块初始化时需在内核空间中申请一块内存(使用 `vmalloc` 或者 `kmalloc` 函数),并在此位置写入内容,随后利用文件操作集中的 `mmap` 将其映射到用户空间的文件中;在用户空间程序中,再通过读取此文件将内核空间的内容显示出来。

8.3 实验原理

1) 内存地址空间的分配

(1) 进程虚拟空间

Linux 内核将 4G 的空间分为两部分。最高的 1G 字节(从虚拟地址 `0xC0000000` 到 `0xFFFFFFFF`),供内核使用,称为“内核空间”。而将较低的 3G 字节(从虚拟地址 `0x00000000` 到 `0xBFFFFFFF`),供各个进程使用,称为“用户空间”。因为每个进程可以通过系统调用进入内核,因此, Linux 内核由系统内的所有进程共享。于是,从具体进程的角度来看,每个进程可以拥有 4G 字节的虚拟空间。图 8.1 给出了进程虚拟空间示意图。

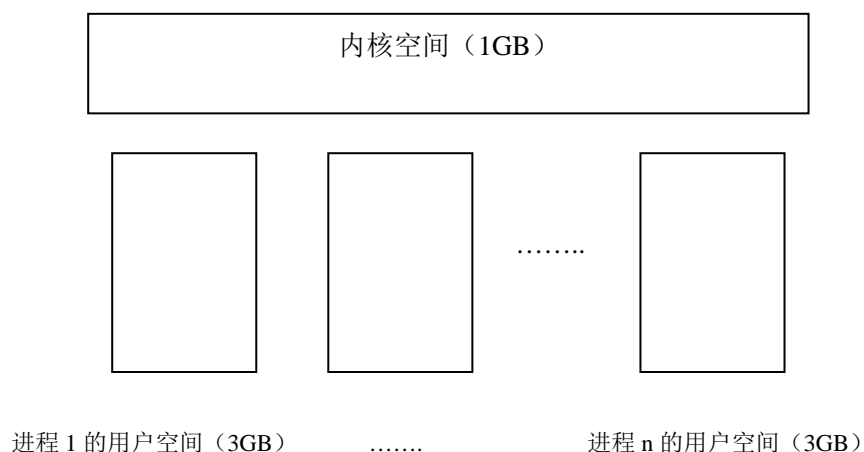


图 8.1 Linux 进程虚拟空间示意图

（2）内核空间的分配

4G 的虚拟地址中有 1G 是分配给内核来使用的（从虚拟地址 0xC0000000 到 0xFFFFFFFF），图 8.2 便是此 1G 空间的具体分配情况。

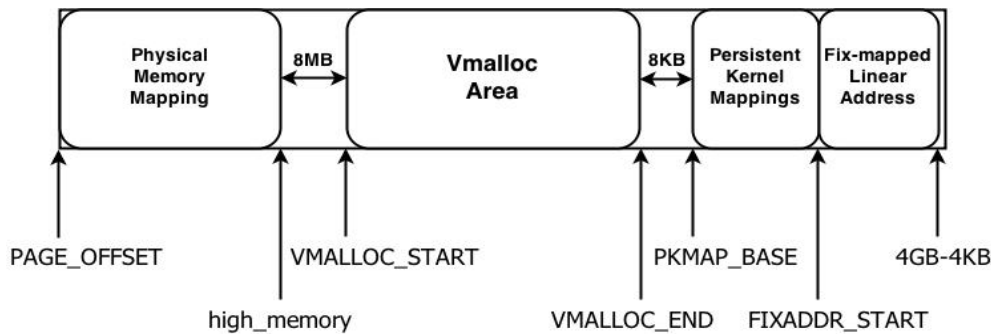


图 8.2 内核虚拟空间划分说明图

其中 `PAGE_OFFSET` 被定义为 0xC0000000，即内核虚拟地址空间的起始位置，从此位置到 `high_memory` 的空间叫做物理内存直接映射区，只有这一段，物理地址与虚拟地址之间是简单的线性关系，即： $\text{physical address} = \text{virtual address} - \text{PAGE_OFFSET}$ ，要在这段内存分配内存，则调用 `kmalloc()` 函数。反过来说，通过 `kmalloc()` 分配的内存，其物理页是连续的，对于内核物理内存映射区的虚拟内存，使用 `virt_to_phys()` 和 `phys_to_virt()` 来实现物理地址和内核虚拟地址之间的互相转换（实际上仅仅做了 3G 的地址移位）。`high_memory` 是内核定义的一个变量，用来保存内核所映射的实际内存最大物理地址对应的内核逻辑地址。内核物理内存映射区的大小在本实验环境下大约为 891MB（很多资料中此值为 896MB）。

其后的内存区域，即从 `high_memory` 到 `4GB-4KB` 的空间可作为另一区域，至于为什么要这样来分这 1GB 的内存区域，会在随后的内核虚拟空间到物理空间的映射部分进行说明。

下面继续分析此图，`high_memory` 和 `VMALLOC_START` 中间有一个 8MB 的安全区，实际为了“捕获”对内存的越界访问，避免内存越界等安全问题。`VMALLOC_START` ~ `VMALLOC_END` 之间是非连续内存区，称为 `vmalloc_area` 区域，大小为 120MB，出于同样的原因，每个非连续内存区之间有 4KB 的安全区来隔离非连续区。`vmalloc_area` 区域的分配由 `vmalloc()` 函数执行。其后 `PKMAP_BASE` 位置处开始的 2M 空间（本实验机器中是如此）称为永久内存映射区域；最后的 `FIXADDR_BASE` 到 `FIXADDR_TOP` 的 932KB 空间为固定内存映射区域。

（3）内核虚拟空间到物理地址空间的转换

x86 架构中物理内存被划分为三个区来管理，它们是 `ZONE_DMA`、`ZONE_NORMAL` 和

ZONE_HIGHMEM。

		包含 ISA/PCI 设备需要
ZONE_DMA	0-16 MB	的低端物理内存区域中的
		内存范围。
		由内核直接映射到高端范
		围的物理内存的内存范
ZONE_NORMAL	16-891 MB	围。所有的内核操作都只
		能使用这个内存区域来进
		行，因此这是对性能至关
		重要的区域。
ZONE_HIGHMEM	891MB到更高的内存	系统中内核不能映像到的
		其他可用内存。

首先需要介绍下**高端内存**的概念，linux中内核使用 3G-4G的线性地址空间，如果实际的物理内存大于 1G，则将 1G的线性地址空间划分为两部分：小于 891M物理地址空间的称之为低端内存，这部分内存的物理地址和 3G开始的线性地址是一一对应映射的,也就是说内核使用的线性地址空间 3G--(3G+891M)和物理地址空间 0-891M一一对应；剩下的 133M的线性空间用来映射剩下的大于 891M的物理地址空间，这也就是我们通常说的高端内存区，在图 8.2 中显示的PAGE_OFFSET到high_memory是用来映射低端内存，high_memory到 4GB 则映射高端内存区域。至于这样划分的原因，就是为了解决实际物理内存大于内核虚拟空间的映射访问问题。但请注意，高端内存的概念仅存在于 32 位机器中，64 位机器中由于内核虚拟地址空间足够去映射物理内存，所以不需要留出这 133MB的空间来映射多出来的空间（除非你的物理内存超过 512GB）。

低端内存到内核虚拟地址的映射是线性的，即内核空间虚拟地址减去PAGE_OFFSET即是物理地址。比如一般物理地址的 0MB—16MB作为DMA区域，该区域的物理页面专门供I/O设备的DMA使用，映射到内核虚拟区域就是 3G—3G+16MB；内核镜像（内核的代码和数据）从 3G+16MB地址处开始映射（此处 16 即为CONFIG_PHYSICAL_START的值 0x1000000）。**高端内存区域到物理地址区域的映射不是线性的**，需要建立页表，对应的物理内存，可以是low memory对应的内存区，也可以是high memory对应的内存区。

2) 内存分配方式kmalloc和vmalloc的区别

`kmalloc` 保证分配的内存存在物理上是连续的，所以能分配的大小有限，最多只能开辟大小为 $32 * \text{PAGE_SIZE}$ 的内存, $\text{PAGE_SIZE}=4\text{kB}$, 也就是 128kB 的大小的内存（但测试了下可以申请大于此数的内存）。`kmalloc` 在申请空间的时候，一般分配给其的虚拟空间位于物理内存直接映射区，所以求对应的物理地址时，只需要减去一个偏移量即可得到真正的物理地址。在内核文件 `arch/x86/include/asm/io.h` 第 112 行有这样一个内联函数

```
static inline phys_addr_t virt_to_phys(volatile void *address)
{
    return __pa(address);
}
```

而其中 `__pa(x)` 的宏定义也很简单, 其作用就是让逻辑地址减去位移量 `PAGE_OFFSET` (默认值为 `0xC0000000`)

```
#define __pa(x) ((unsigned long)(x)-PAGE_OFFSET)

(定义在 arch/x86/include/asm/page.h, line 40)
```

相反地给定一个物理地址也可计算 `kmalloc` 虚拟地址。

`vmalloc` 申请的内存地址（我们暂且称其为 `vmalloc` 虚拟地址）位于高端内存区中的 `VMALLOC_START` 到 `VMALLOC_END`，与实际物理地址不成线性映射关系，在分配物理内存时需要通过页表项来进行逻辑页面和物理页面的对应，获得的页必须一个一个的进行映射（因为它们物理上不是连续的），至于实际物理页面的分配管理，有伙伴关系和更复杂的 `slab` 技术进行支撑，这里不去进行介绍。在求一个内存地址所对应的物理页时可以直接利用 `vmalloc_to_page` 函数求出该 `vmalloc` 地址对应的实际物理页。下面以 `vmalloc_to_page`（位于 `linux/mm/vmalloc.c`）中实现 `vmalloc` 地址到对应物理页的映射进行说明。

//返回 `vmalloc` 内存首地址对应的物理页面描述符，输入参数为 `vmalloc` 地址

```
struct page *vmalloc_to_page(const void *vmalloc_addr)
{
    unsigned long addr = (unsigned long) vmalloc_addr;
    struct page *page = NULL;
    //获取页全局目录的描述符
    pgd_t *pgd = pgd_offset_k(addr);
    VIRTUAL_BUG_ON(!is_vmalloc_or_module_addr(vmalloc_addr));

    if (!pgd_none(*pgd)) {
        //求出在第二级页目录表表项的描述符
```

```

pud_t *pud = pud_offset(pgd, addr);
if (!pud_none(*pud)) {
    //第三级页目录表项的描述符
    pmd_t *pmd = pmd_offset(pud, addr);
    if (!pmd_none(*pmd)) {
        pte_t *ptep, pte;
        //求出对应的页表项
        ptep = pte_offset_map(pmd, addr);
        pte = *ptep;
        if (pte_present(pte))
            //该页表项对应的物理页描述符
            page = pte_page(pte);
        pte_unmap(ptep);
    }
}
}
return page;
}

```

为了更好地理解该函数的实现过程，可参考下图 8.3。

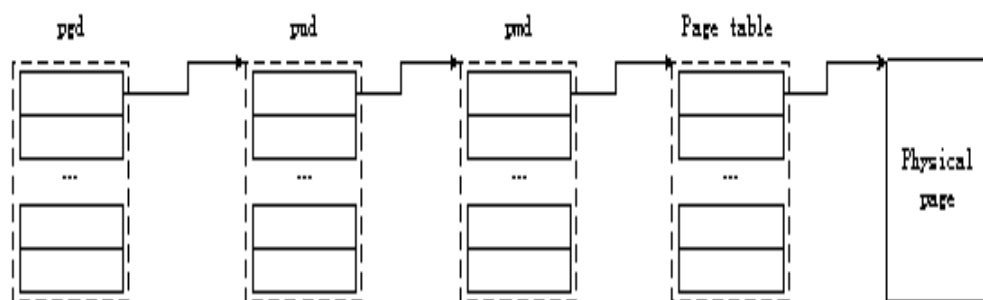


图 8.3 内核空间 vmalloc 地址到实际物理页描述符的转换

从上文可以看出，实验机器所使用内核 3.8.13 的内存是四级页表管理，分别为 pgd、pud、pmd 和 pte。

再通过对 vmalloc 函数的分析，可以看出 vmalloc 优先使用 HIGHMEM 内存。

```

void *vmalloc(unsigned long size)
{
    return __vmalloc_node_flags(size, -1, GFP_KERNEL|__GFP_HIGHMEM);
}

```

下面通过下图 8.4 简单总结 vmalloc 逻辑地址和 kmalloc 逻辑地址转换到物理地址的过程。

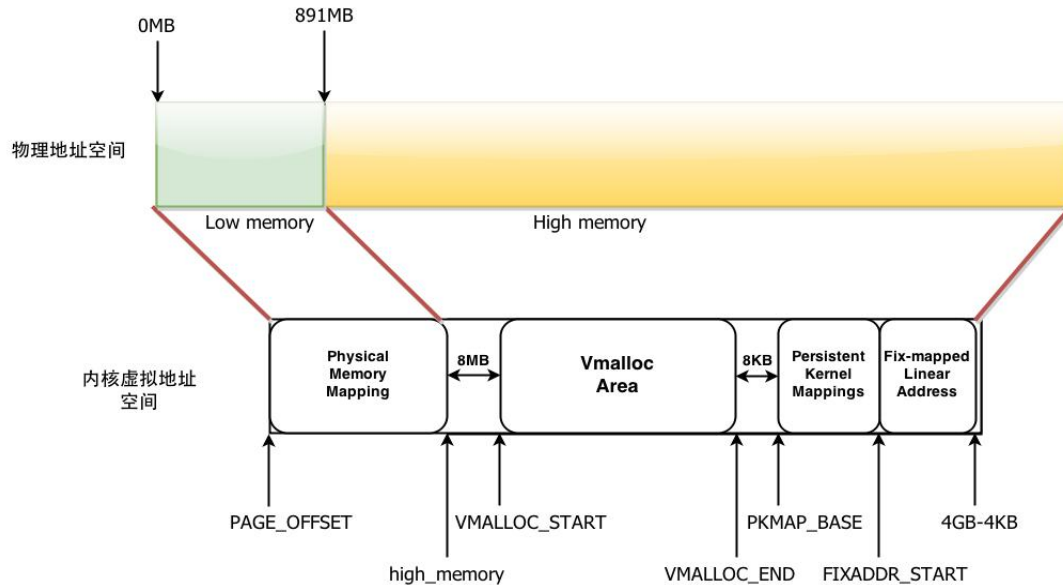


图 8.4 内核地址区域到物理地址区域的对应

3) mmap系统调用

(1) 基础介绍

`mmap` 将一个文件或者其它对象映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会清零。`munmap` 执行相反的操作，删除特定地址区域的对象映射。

当使用 `mmap` 映射文件到进程后,就可以直接操作这段虚拟地址进行文件的读写等操作,不必再调用 `read,write` 等系统调用.但需注意,直接对该段内存写时不会写入超过当前文件大小内容.

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

`mmap` 函数在用户空间的函数原型为 `void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)`;各参数解释如下：

start: 映射区的开始地址。

length: 映射区的长度。

prot: 期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过 or 运算合理地组合在一起

flags: 指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

fd: 有效的文件描述词。如果 MAP_ANONYMOUS 被设定，为了兼容问题，其值应为 -1。

offset: 被映射对象内容的起点。

而内核空间对应的 mmap 函数原型为 `int mmap(struct file *flip, struct vm_area_struct *vma);`其作为设备文件的一个操作函数。

(2) 映射过程

mmap 系统调用的最终目的是将设备或文件映射到用户进程的虚拟地址空间,实现用户进程对文件的直接读写,这个任务可以分为以下两步:

1. 在用户虚拟地址空间中寻找空闲的满足要求的一段连续的虚拟地址空间,为映射做准备(由内核 mmap 系统调用完成)

在 Linux 内核中对应进程内存区域的数据结构。在内核中，用结构 `struct vm_area_struct` 来表示.这一段连续的、具有相同访问属性的虚存空间，该虚存空间的大小为物理内存页面的整数倍。可以用 `cat /proc/<pid>/maps` 来查看一个进程的内存使用情况,pid 是进程号,其中显示的每一行对应进程的一个 `vm_area_struct` 结构。

下面是 `struct vm_area_struct` 结构体的定义：

```
#include <linux/mm_types.h>

/* This struct defines a memory VMM memory area. */

struct vm_area_struct {

    struct mm_struct * vm_mm; /* VM area parameters */

    unsigned long vm_start;

    unsigned long vm_end;

    /* linked list of VM areas per task, sorted by address */

    struct vm_area_struct *vm_next;

    pgprot_t vm_page_prot;

    unsigned long vm_flags;

    /* AVL tree of VM areas per task, sorted by address */
```

```

short vm_avl_height;

struct vm_area_struct * vm_avl_left;

struct vm_area_struct * vm_avl_right;

/* For areas with an address space and backing store,
vm_area_struct *vm_next_share;

struct vm_area_struct **vm_pprev_share;

struct vm_operations_struct * vm_ops;

unsigned long vm_pgoff; /*offset in PAGE_SIZE units, not PAGE_CACHE_SIZE*/

struct file * vm_file;

unsigned long vm_raend;

void * vm_private_data; /* was vm_pte (shared mem) */

};

```

为了说明打开的文件和此结构的关系，请见下图 8.5。

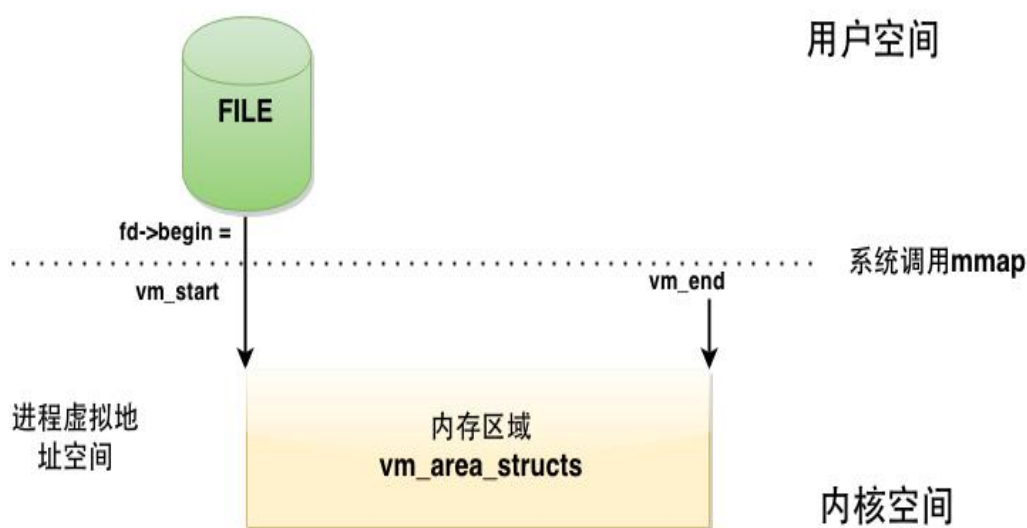


图 8.5 用户态文件到vm_area_struct结构映射

以上图示中 $fd->begin == vm_start$ 仅当用户空间的 `mmap` 函数的最后一个参数 `offset` 为 0 时才是此，如果 `offset` 为 4096，文件的映射位置则需要向后偏移一个页的大小，此时 `vm_area_struct->vm_pgoff` 的值也变为 1。

如果在初始化函数中，使用的是 `vmalloc` 申请的内核空间，则需要在 `mmap` 函数中为 `vm_area_struct` 绑定一个 `fault` 方法，完成 `vma` 虚拟内存区和实际物理区域的页映射，建立页表，此方法属于 `vm_ops` 集。如果使用的是 `kmalloc`，可以使用前面这种方式，也可以不使

用 `fault` 方法，直接在初始化函数中使用 `remap_pfn_range` 将用户空间的一个 `vma` 虚拟内存区映射到一段连续物理页面上，具体用法，详见下文。

因此, `mmap` 系统调用所完成的工作就是准备这样一段虚存空间,并建立 `vm_area_struct` 结构体,将其传给具体的设备驱动程序。

2. 虚拟地址空间和文件或设备的物理地址之间的映射(设备驱动完成)

建立文件映射的第二步就是建立虚拟地址和具体的物理地址之间的映射,这是通过修改进程页表来实现的。实现这一步主要依赖上文提到的 `fault` 方法，是 `vm_ops` 中的一个操作方法，当用户虚拟空间并没有实际映射到物理页面时，就会发生缺页，然后调用 `fault` 方法进行物理页面和虚拟空间页面的映射。这也是此实验中最重要的一步。`fault` 的函数原型为 `int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf)`;其中 `vma` 表示的是进程虚拟地址空间，`vmf` 则是进行用户虚拟空间和物理地址空间连接的一个结构体。`vm_fault` 结构体描述如下：

```
struct vm_fault {
    unsigned int flags; /* FAULT_FLAG_xxx flags */
    pgoff_t pgoff; /* Logical page offset based on vma */
    void __user *virtual_address; /* Faulting virtual address */

    struct page *page; /* ->fault handlers should return a
*   page here, unless VM_FAULT_NOPAGE
*   is set (which is also implied by
*   VM_FAULT_ERROR).
*/
};
```

`flags` 是该结构体的描述参数；`pgoff` 比较重要，表示的 `vma` 空间的偏移量，单位为页；`virtual_address` 记录的是发生缺页的用户虚拟地址；`page` 则是最后指向对应的物理页。需要注意的是 `fault` 方法并不是一次完成映射，而是一页一页地进行页表项的建立。代码中的

具体做法就是先得到发生缺页的用户虚拟地址（按页对齐，一页一页地递增），随后将 vmalloc 地址区域中每一页的虚拟地址对应的物理页描述符计算出来（使用上文提到的 vmalloc_to_page 函数），最后赋值到 vm_fault 结构体中的 page 项即可。这样就将用户空间地址和内核物理地址连接起来了。以上过程如下图 8.6 所示。

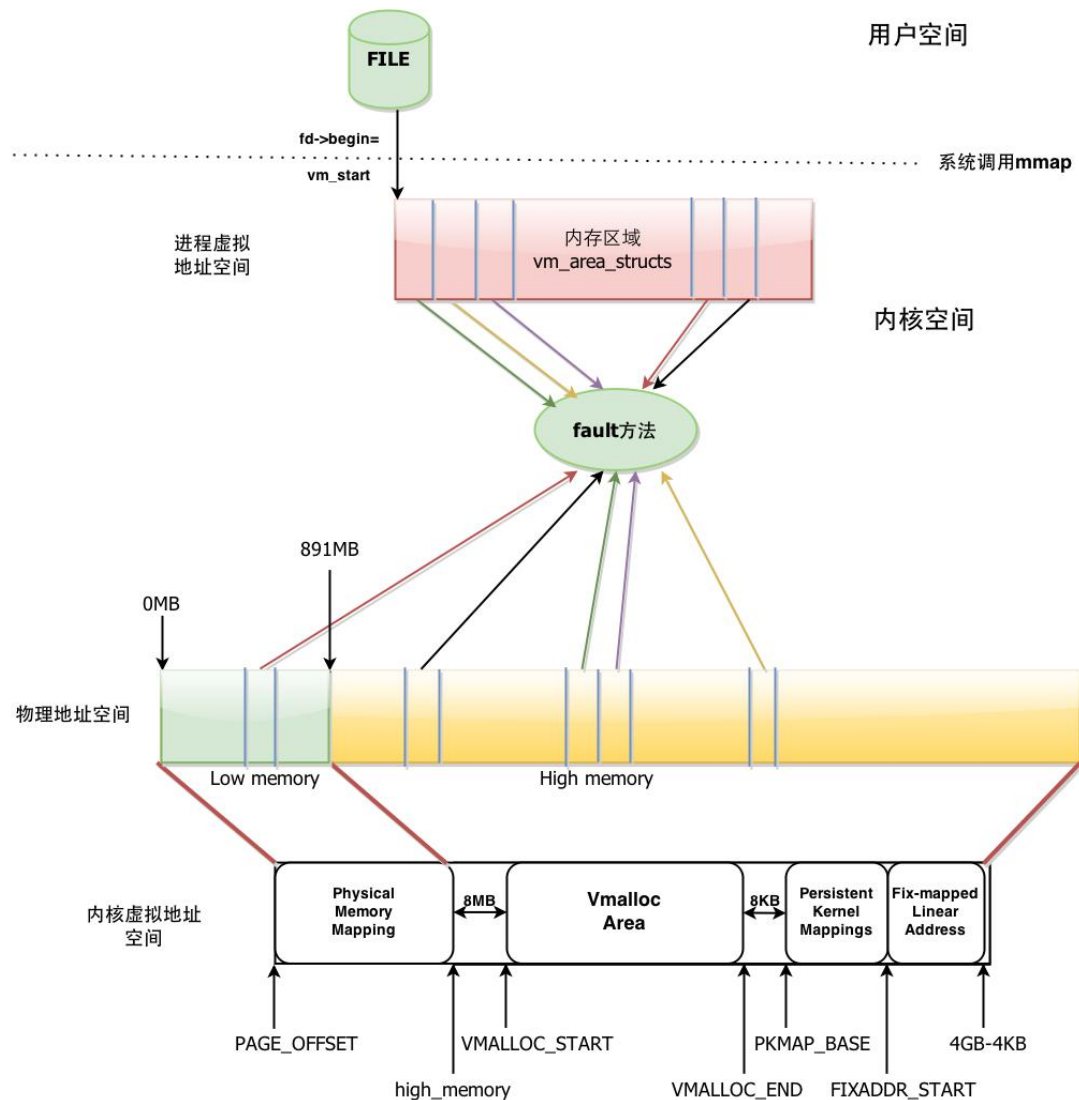


图 8.6 vma结构到物理空间的映射

对于kmalloc申请的虚拟内存区，除了上述方法外，还可以使用remap_pfn_range直接在初始化函数中实现区域的连续映射，相比较上面的方法，由于是一次性映射，效率会更高。

remap_pfn_range 函数原型为 `int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn, unsigned long size, pgprot_t prot)`；其中参数vma是内核根据用户的请求自己填写的，而参数addr表示内存映射开始处的虚拟地址，因此，该函数为addr~addr+size之间的虚拟地址构造页表。另外，pfn (Page Fram Number)

是虚拟地址应该映射到的物理地址的页面号，实际上就是物理地址右移PAGE_SHIFT位。如果PAGE_SHIFT为 4KB，则PAGE_SHIFT为 12。最后一个参数prot是新页所要求的保护属性。值得注意的是函数中的第三个参数pfn一般可由virt_to_phys得到物理地址，再右移PAGE_SHIFT得到pfn的值，而不是直接使用virt_to_page来得到。

这一部分介绍了两种方式下将用户虚拟区域映射到实际物理内存区域的具体实现方法，也是整个实验的重点。

8.4 实验步骤

下面将分别给出 kmalloc 和 vmalloc 映射的源代码，读者可以比较其实现差别。

(1) 利用 Linux 的动态模块，编写的设备驱动程序，其代码如下：

//kmalloc_map.c 源代码

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include <asm/page.h>
#include <linux/mm.h>
#include <linux/slab.h>

#define MMAPNOPAGE_DEV_NAME "mmapnopcode"
#define MMAPNOPAGE_DEV_MAJOR 240
#define SHARE_MEM_PAGE_COUNT 4
#define SHARE_MEM_SIZE (PAGE_SIZE*SHARE_MEM_PAGE_COUNT)

char *share_memory=NULL;

int mmapnopcode_mmap(struct file *filp,struct vm_area_struct *vma)
{
    unsigned long page;
    unsigned char i;
    unsigned long start = (unsigned long)vma->vm_start;
    unsigned long size = (unsigned long)(vma->vm_end - vma->vm_start);
    page = virt_to_phys(share_memory);
```

```

        if(remap_pfn_range(vma,start,page>>PAGE_SHIFT,size,PAGE_SHARED))
            return -1;
    }

    struct file_operations mmapnopcode_fops={
        .owner=THIS_MODULE,
        .mmap=mmapnopcode_mmap,
    };

    int mmapnopcode_init(void)
    {
        int lp;
        int result;
        result=register_chrdev(MMAPNOPAGE_DEV_MAJOR,
                               MMAPNOPAGE_DEV_NAME,
                               &mmapnopcode_fops);
        if(result<0){
            return result;
        }

        share_memory=kmalloc(SHARE_MEM_SIZE, GFP_KERNEL);
        for(lp=0;lp<SHARE_MEM_PAGE_COUNT;lp++)
        {
            sprintf(share_memory+PAGE_SIZE*lp,"LELETEST %d",lp);
        }
        return 0;
    }

    void mmapnopcode_exit(void)
    {
        if(share_memory!=NULL)
        {
            kfree(share_memory);
        }
        unregister_chrdev(MMAPNOPAGE_DEV_MAJOR, MMAPNOPAGE_DEV_NAME);
    }

    module_init(mmapnopcode_init);
    module_exit(mmapnopcode_exit);
    MODULE_LICENSE("Dual BSD/GPL");

//vmalloc_map.c 源代码
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

```

```
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/vmalloc.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include <asm/page.h>
#include <linux/mm.h>

#define MMAPNOPAGE_DEV_NAME "mmapnopcode"
#define MMAPNOPAGE_DEV_MAJOR 240
#define SHARE_MEM_PAGE_COUNT 4
#define SHARE_MEM_SIZE (PAGE_SIZE*SHARE_MEM_PAGE_COUNT)

char *share_memory=NULL;

int mmapnopcode_vm_fault(struct vm_area_struct *vma,
                        struct vm_fault *vmf)
{
    struct page *page;
    unsigned long offset;
    void *page_ptr;

    printk("\n");
    printk("%-25s %08x\n", "1)vma->flags", vmf->flags);
    printk("%-25s %08x\n", "2)vmf->pgoff", vmf->pgoff);
    printk("%-25s %08x\n", "3)vmf->virtual_address", vmf->virtual_address);
    printk("%-25s %08x\n", "4)vma->vm_start", vma->vm_start);
    printk("%-25s %08x\n", "5)vma->vm_end", vma->vm_end);
    printk("%-25s %08x\n", "6)vma->vm_pgoff", vma->vm_pgoff);
    printk("%-25s %d\n", "7)PAGE_SHIFT", PAGE_SHIFT);

    page_ptr=NULL;
    if((NULL==vma)|| (NULL==share_memory)){
        printk("return VM_FAULT_SIGBUS!\n");
        return VM_FAULT_SIGBUS;
    }

    offset=vmf->virtual_address-vma->vm_start;

    if(offset>=SHARE_MEM_SIZE){
        printk("return VM_FAULT_SIGBUS!");
        return VM_FAULT_SIGBUS;
    }
}
```

```
    }
    printk("offset    %d\n", offset);
    page_ptr=share_memory+offset;
    page=vmalloc_to_page(page_ptr);
    get_page(page);
    vmf->page=page;
    return 0;
}

struct vm_operations_struct mmapnopcode_vm_ops={
    .fault=mmapnopcode_vm_fault,
};

int mmapnopcode_mmap(struct file *filp,struct vm_area_struct *vma)
{
    vma->vm_flags |= VM_RESERVED;
    vma->vm_ops=&mmapnopcode_vm_ops;
    return 0;
}

struct file_operations mmapnopcode_fops={
    .owner=THIS_MODULE,
    .mmap=mmapnopcode_mmap,
};

int mmapnopcode_init(void)
{
    int lp;
    int result;

    result=register_chrdev(MMAPNOPAGE_DEV_MAJOR,
                          MMAPNOPAGE_DEV_NAME,
                          &mmapnopcode_fops);
    if(result<0){
        return result;
    }

    share_memory=vmalloc(SHARE_MEM_SIZE);
    for(lp=0;lp<SHARE_MEM_PAGE_COUNT;lp++){
        sprintf(share_memory+PAGE_SIZE*lp,"LELETEST %d",lp);
    }

    return 0;
}
```

```

void mmapnpage_exit(void)
{
    if(share_memory!=NULL){
        vfree(share_memory);
    }
    unregister_chrdev(MMAPNOPAGE_DEV_MAJOR,
        MMAPNOPAGE_DEV_NAME);
}

module_init(mmapnpage_init);
module_exit(mmapnpage_exit);
MODULE_LICENSE("Dual BSD/GPL");

```

(2) 为了便于对上述设备驱动程序 **kmalloc_map.c** (模块源程序) 进行编译, 建立 **Makefile** 文件, 代码如下: (以下部分均以 **kmalloc_map.c** 为例进行叙述, 测试 **vmalloc_map.c** 读者可自行修改参数)

```

ifneq ($(KERNELRELEASE),)
obj-m := kmalloc_map.o
else
KERNELDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
clean:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) clean

```

(3) 编译模块程序

利用 **Makefile** 对字符设备驱动程序 **kmalloc_map.c** 进行编译。

```
#make
```

编译后生成 **kmalloc_map.ko**

(4) 加载模块

```
#insmod kmalloc_map.ko
```

(5) 创建设备节点

在 **/dev** 目录下创建 **mmapnpage** 文件 (也可参照实验六中的自动创建文件的方法)

```
#mknod /dev/mmapnpage c 240 0
```

(6) 编写读写测试程序 **test.c**

为了验证物理地址空间真正映射到了用户细腻地址空间, 编写 **test.c** 函数, 测试是否可

通过用户空间的文件读取内核空间的数据。

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define DEVICE_FILENAME "/dev/mmapnopcode"
#define SHARE_MEM_PAGE_COUNT 4
#define SHARE_MEM_SIZE (4096*SHARE_MEM_PAGE_COUNT)

int main()
{
    int dev;
    int loop;
    char *ptrdata;

    dev=open(DEVICE_FILENAME,O_RDWR|O_NDELAY);

    if(dev>=0)
    {
        printf("open file success!\n");
        ptrdata=(char*)mmap(0,
            SHARE_MEM_SIZE,
            PROT_READ|PROT_WRITE,
            MAP_SHARED,
            dev,
            0);

        for(loop=0;loop<SHARE_MEM_PAGE_COUNT;loop++)
        {
            printf("[%s%x]\n",ptrdata+4096*loop,ptrdata+4096*loop);
        }
        munmap(ptrdata,SHARE_MEM_SIZE);
        close(dev);
    }
    else
    {
        printf("open fail!\n");
    }
    return 0;
}
```

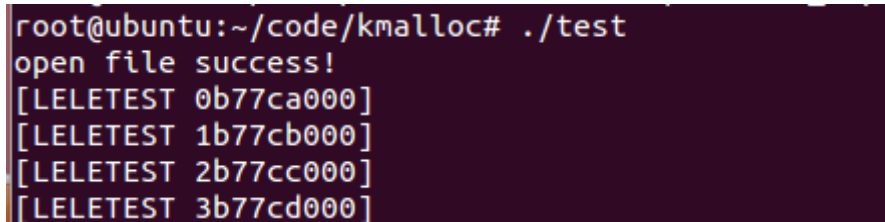

(7) 编译测试程序

```
#gcc -o test test.c
```

(8) 运行测试程序

```
#./test
```

运行效果如图 8-6 所示。



```
root@ubuntu:~/code/kmalloc# ./test
open file success!
[LELETEST 0b77ca000]
[LELETEST 1b77cb000]
[LELETEST 2b77cc000]
[LELETEST 3b77cd000]
```

图 8.6 读取测试效果图

(9) 删除设备节点

```
#rm /dev/ mmapnopcode
```

(10) 卸载模块

```
#rmmod kmalloc_map
```

至于实际的物理分配页面采用的方法与函数，读者可在实验课后自己查阅相关内容。