

第一章 linux 内核编译及添加系统调用

1.1 设计目的和内容要求

1. 设计目的

Linux 是开源操作系统，用户可以根据自身系统需要裁剪、修改内核，定制出功能更加合适、运行效率更高的系统，因此，编译 linux 内核是进行内核开发的必要基本功。

在系统中根据需要添加新的系统调用是修改内核的一种常用手段，通过本次实验，学习者应理解 linux 系统处理系统调用的流程以及增加系统调用的方法。

2. 内容要求

- (1) 编译一个干净的 linux 内核并加载成功，不需要对内核进行修改。
- (2) 在 (1) 中新编译的内核中，添加一个系统调用，实现对指定进程的 nice 值的修改或读取功能。建议调用原型为：

```
Int mysetnice(pid_t pid, int flag, int nicevalue);
```

参数含义：

pid: 进程 ID

flag: 若值为 0，功能是读取 nice 值；若值为 1，功能是设置 nice 值。

系统调用成功时返回 0，失败时返回错误码 EFAULT。

3. 学时安排（共 4 学时）

4. 开发平台

Linux 环境，gcc，gdb，vim 或 gedit 等。

1.2 linux 系统调用基本概念

系统调用的实质是调用系统函数，于内核态中运行。Linux 系统中用户（或封装例程）通过执行一条访管指令“int \$0x80”来实现系统调用，该指令会产生一个访管中断，从而让系统暂停当前进程的执行，而转去执行系统调用处理程序，通过用户态传入的系统调用号从系统调用表中找到相应服务例程的入口并执行，完成后返回。下面介绍相关的基本概念。

1. 系统调用号

Linux 系统提供了多达几百种的系统调用，为了唯一的标识每一个系统调用，linux 为每个系统调用都设置了一个唯一的编号，称为系统调用号，在 4.4.19 内核版本中，定义在文件/usr/include/asm-generic/unistd.h 中（注意，不同的内核版本，该头文件的位置会稍有不同），比如大家比较熟悉的几个系统调用的调用号是：

```
#define __NR_close 57
#define __NR_read 63
#define __NR_write 64
#define __NR_exit 93
```

```
#define __NR_open 1024
```

```
#define __NR_vfork 1071
```

```
#define __NR_fork 1079
```

系统调用号非常关键，一旦分配就不能再有任何变更，否则之前编译好的应用程序就会崩溃。在 x86 中，系统调用号是通过 `eax` 寄存器传递给内核的。在陷入内核之前，用户空间将系统调用号存入 `eax` 中，这样系统调用处理程序一旦运行，就可以从 `eax` 中得到调用号了。

2. 系统调用表

系统调用表是 linux 内核中用于关联系统调用号及其相对应的服务例程入口地址的一张表，存放在 `/usr/src/linux-4.4.19/arch/x86/entry/syscalls/syscall_64.tbl` 文件中，每个表项记录了某个系统调用的服务例程的入口地址，以系统调用号为索引可快速找到其服务例程。如：

0	common read	<code>sys_read</code>
1	common write	<code>sys_write</code>
2	common open	<code>sys_open</code>
3	common close	<code>sys_close</code>

3. 系统调用服务例程

每个系统调用都对应一个内核服务例程来实现该系统调用的具体功能，其命名格式都是以 “`sys_`” 开头，如 `sys_read` 等，通常存放在 `/usr/src/linux-source-4.4.19/kernel/sys.c` 文件中。服务例程的原型声明通常都有固定的格式，如 `sys_open` 的原型为：

```
asmlinkage long sys_open(const char __user *filename,int flags, int mode);
```

其中 “`asmlinkage`” 是一个必须的限定词，用于通知编译器仅从堆栈中提取该函数的参数，而不是从寄存器中，因为在执行服务例程之前系统已经将通过寄存器传递过来的参数值压入内核堆栈了。不过在新版本的内核中，引入了宏 “`SYSCALL_DEFINE3(sname)`” 对服务例程原型进行了封装，其中的 “`N`” 是该系统调用所需要参数的个数，如上述 `sys_open` 调用的原型声明格式为：

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, int, mode)
```

如后面添加系统调用示例程序中，服务例程的声明格式为：

```
SYSCALL_DEFINE0(mysyscall)
```

本知识点的详细介绍大家可以参考网页：

<http://blog.csdn.net/adc0809608/article/details/7417180>

4. 系统调用参数传递

与普通函数一样，系统调用通常也需要输入/输出参数。在 x86 上，linux 通过 6 个寄存器来传入参数值，其中 `eax` 传递系统调用号，后面 5 个寄存器 `ebx`, `ecx`, `edx`, `esi` 和 `edi` 按照顺序存放前五个参数，需要六个或六个以上参数的情况不多见，此时，应该用一个单独的寄存器存放指向所有这些参数在用户空间地址的指针。服务例程的返回值通过 `eax` 寄存器传递，这是在执行 `ret` 指令时由 C 编译器自动完成的。

当系统调用执行成功时，将返回服务例程的返回值，如 0。但如果执行失败，为防止和正常的返回值混淆，系统调用并不直接返回错误码，而是将错误码放入一个名为 `errno` 的全局变量中，通常是一个负值，通过调用 `perror()` 库函数，可以把 `errno` 翻译成用户可以理解的错误信息描述。

5. 系统调用参数验证

系统调用必须仔细检查用户传入的参数是否合法有效。比如与进程相关的调用必须检查用户提供的 PID 等是否有效。

最重要的是要检查用户提供的指针是否有效，以防止用户进程非法访问数据。内核提供了两个函数来完成必须的检查以及内核空间与用户空间之间数据的来回拷贝：`copy_to_user()`和 `copy_from_user()`，定义在 `/usr/src/linux-4.4.19/include/asm-generic/uaccess.h` 文件中。

(1) `copy_to_user()`:

函数定义原型:

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

功能: 将数据从内核空间复制到用户空间

参数含义:

to: 用户进程空间中的目的内存地址;

from: 内核空间内的源地址

count: 需要拷贝的数据长度(字节数)。

返回值: 函数执行成功返回 0; 如果失败, 则返回没有拷贝成功的字节数。

(2) `copy_from_user()`

函数定义原型:

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

功能: 将数据从用户空间复制到内核空间

参数含义:

to: 用户进程空间中的目的内存地址;

from: 内核空间内的源地址

count: 需要拷贝的数据长度(字节数)。

返回值: 函数执行成功返回 0; 如果失败, 则返回没有拷贝成功的字节数。

1.3 添加系统调用的步骤:

这里以一个很简单的例子说明 linux 中添加一个新的系统调用的步骤, 这个调用没有输入参数和返回值, 只是简单的输出一行信息, 名字叫 “zwhsyscall”。我的内核版本是 4.4.19, x86 平台、64 位。

1. 分配系统调用号

Linux 系统的系统调用号设置保存在 `/usr/include/asm-generic/unistd.h` 文件中, 这里并不容易确定未使用的系统调用号, 可以通过查找系统调用表 (`/usr/src/linux-4.4.19/arch/x86/entry/syscalls/syscall_64.tbl`) 来确定, 如图 1-1 所示, 比如当前系统使用到 325 号, 则新添加的系统调用使用 326 号。

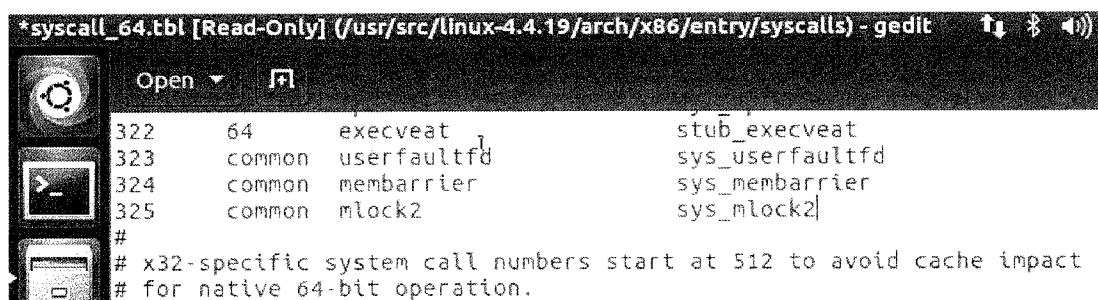


图 1-1 系统调用表部分内容

确定调用号后，修改 `unistd.h` 文件：为新系统调用 `zwsyscall` 设置系统调用号，并将 `__NR_syscalls` 的值加 1（原来是 285），修改前后 `unistd.h` 文件的内容分别如图 1-2 和 1-3 所示：

```
__SYSCTL(__NR_mlock2, sys_mlock2),
#define __NR_mlock2 284
__SYSCALL(__NR_mlock2, sys_mlock2)

#undef __NR_syscalls
#define __NR_syscalls 285
```

图 1-2 修改前 `unistd.h` 文件的部分内容

```
#define __NR_mlock2 326
__SYSCALL(__NR_zwsyscall, sys_zwsyscall)

#undef __NR_syscalls
#define __NR_syscalls 286
```

图 1-3 修改后 `unistd.h` 文件的部分内容

2. 修改系统调用表

确定新调用的系统调用号后，应在系统调用表中关联调用号与新调用的服务例程的入口地址，即修改 `/usr/src/linux-4.4.19/arch/x86/entry/syscalls/syscall_64.tbl` 文件，为新调用添加一条记录，其格式为：

<系统调用号> <common/64/x32> <系统调用名> <服务例程入口地址>

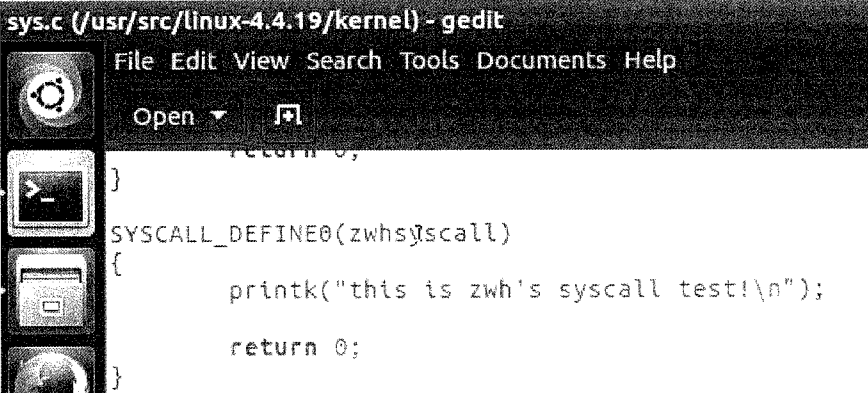
修改结果如图 1-4 所示：

```
325      common  mlock2          sys_mlock2
326      64      zwsyscall       sys_zwsyscall
#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
```

图 1-4 修改系统调用表

3. 实现系统调用服务例程

下面为新调用 `zwsyscall` 编写服务例程 `sys_zwsyscall`，通常添加在 `sys.c` 文件中，其完整路径为：`/usr/src/linux-4.4.19/kernel/sys.c`（不同内核版本会稍有不同），如图 1-5 所示：



```
sys.c (/usr/src/linux-4.4.19/kernel) - gedit
File Edit View Search Tools Documents Help
Open
}
SYSCALL_DEFINE0(zwsyscall)
{
    printk("this is zw's syscall test!\n");
    return 0;
}
```

图 1-5 编写系统调用服务例程

4. 重新编译内核

上面三个步骤已经完成添加一个新系统调用的所有工作，但是要让这个系统调用真正在内核中运行起来，还需要重新编译内核。有关内核编译的知识，见 1.4 节的介绍。

5. 编写用户态程序测试新系统调用

可编写一个用户态程序来调用上面新添加的系统调用：

```
1  #include <linux/unistd.h>
2  #include <sys/syscall.h>
3  #define __NR_mysyscall 326 /*系统调用号根据实验具体数字而定*/
4  int main()
5  {
6      syscall(__NR_mysyscall); /*或 syscall(326) */
7  }
```

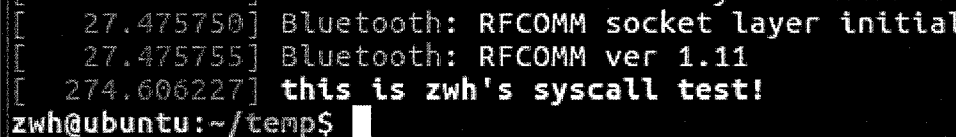
程序说明：

程序第 6 行使用了 `syscall()` 宏调用新添加的系统调用，它是 linux 提供给用户态程序直接调用系统调用的一种方法，其格式为：

```
int syscall(int number, ...);
```

其中 `number` 是系统调用号，`number` 后面应顺序接上该系统调用的所有参数。

编译该程序并运行后，使用 `dmesg` 命令查看输出内容，如图 1-6 所示：



```
[ 27.475750] Bluetooth: RFCOMM socket layer initial
[ 27.475755] Bluetooth: RFCOMM ver 1.11
[ 274.606227] this is zwh's syscall test!
zwh@ubuntu:~/temp$
```

图 1-6 系统调用测试结果

1.4 linux 内核编译步骤

作为自由软件，linux 内核版本不断更新，新内核会修订旧内核的 bug，并增加若干新特性，如支持更多的硬件、具备更好的系统管理能力、运行速度更快、更稳定等。用户若想要使用这些新特性，或希望根据自身系统需求定制一个更高效、更稳定的内核，就需要重新编译内核。下面以 linux 初学者喜欢使用的 ubuntu 系统为例，介绍内核编译步骤。

1. 实验环境

Ubuntu 64 位：ubuntu-16.04-desktop-amd64.iso，待编译的新内核是 linux-4.4.19.tar.xz。

虚拟机：VMware-player-12.1.1-3770994.exe

2. 下载内核源码

Linux 的内核源代码是完全公开的，有很多网站都课堂源码下载，推荐使用 linux 的官方网站：<http://www.kernel.org>，在这里可以找到所有的内核版本，如图 1-7 所示：

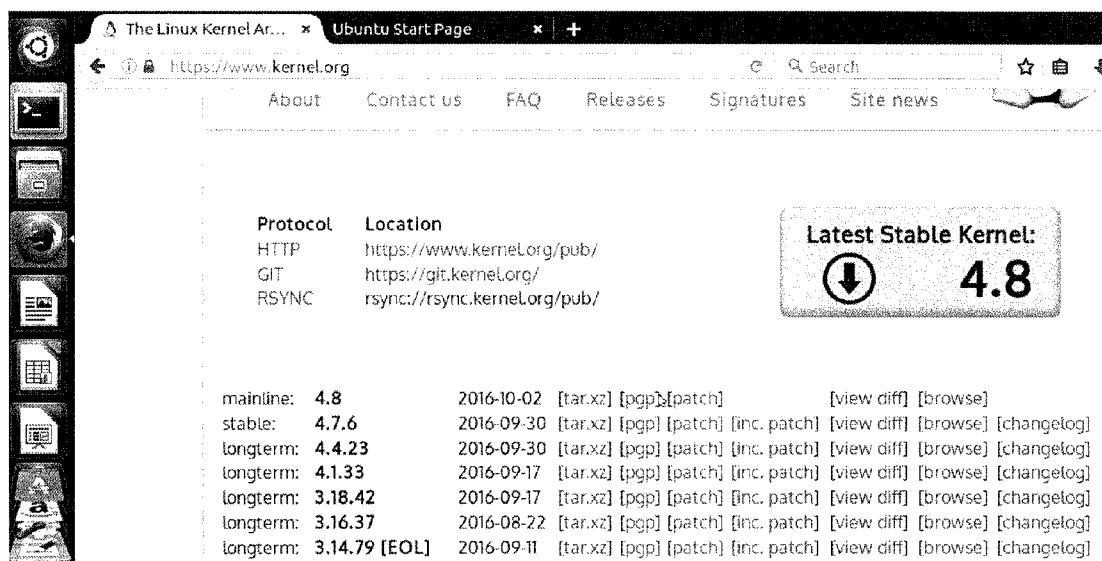


图 1-7 linux 官方网站

3. 解压缩内核源码文件

首先切换到 root 用户，将下载的新内核压缩文件复制到/usr/src 下面，然后进入/usr/src 子目录中，分两步解压缩：

- (1) `# xz -d linux-4.4.19.tar.xz` 大概执行 1 分钟左右，中间没有任何信息显示
- (2) `# tar -xvf linux-4.4.19.tar`

4. 清楚残留的.config 和.o 文件

进入 linux-4.4.19 子目录，清除残留的.config 和.o 文件：

```
# make mrproper
```

这里可能会提醒安装 ncurses 包，在 ubuntu 中 ncurses 库的名字是：libncurses5-dev，所以安装命令是：

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

5. 执行：make mrproper

当编译出错需要重新编译或不是第一次编译，都需要执行该命令清理编译历史。

6. 配置内核

运行命令：`# make menuconfig`

在显示的对话框中：对于每一个配置选项，用户可以回答“y”、“m”或“n”。其中“y”表示将相应特性的支持或设备驱动程序编译进内核；“m”表示将相应特性的支持或设备驱动程序编译成可加载模块，在需要时，可由系统或用户自行加入到内核中去；“n”表示内核不提供相应特性或驱动程序的支持。

一般采用默认值即可，但要检查设备驱动配置情况，如图 1-8 所示：

```
Bus options (PCI etc.) --->
Executable file formats / Emulations --->
[*] Networking support --->
Device Drivers --->
Firmware Drivers --->
File systems --->
Kernel hacking --->
```

图 1-8 内核配置对话框

(1) 进入 scsi 设备配置界面，如图 1-9 和 1-10，将下面几项全部设置成 “*”：

```
Misc devices --->
< > ATA/ATAPI/MFM/RLL support (DEPRECATED) ----
[*] SCSI device support --->
<*> Serial ATA and Parallel ATA drivers (libata) ---
[*] Multiple devices driver support (RAID and LVM)
<M> Generic Target Core Mod (TCM) and ConfigFS Infra:
```

图 1-9 设备配置界面 1

```
| | SCSI: use blk-mq I/O path by default
[*] legacy /proc/scsi/ support
*** SCSI support type (disk, tape, CD-ROM) ***
<*> SCSI disk support
<*> SCSI tape support
<*> SCSI OnStream SC-x0 tape support
<*> SCSI CDROM support
[ ] Enable vendor-specific extensions (for SCSI CDROM)
<*> SCSI generic support
```

图 1-10 设备配置界面 2

(2) 退回到上一个界面，设置 “Fusion MPT device support”：设置为 “*”，如图 1-11 和 1-12 所示：

```
<*> Serial ATA and Parallel ATA drivers (lib)
[*] Multiple devices driver support (RAID an
<M> Generic Target Core Mod (TCM) and Config
[*] Fusion MPT device support --->
IEEE 1394 (FireWire) support --->
```

图 1-11 设备配置界面 3

```
[-] Fusion MPT device support
<*> Fusion MPT ScsiHost drivers for SPI
<M> Fusion MPT ScsiHost drivers for FC
<*> Fusion MPT ScsiHost drivers for SAS
(128) Maximum number of scatter gather entries (16 - 128)
<*> Fusion MPT misc device (ioctl) driver
<M> Fusion MPT LAN driver
[*] Fusion MPT logging facility
```

图 1-12 设备配置界面 4

(3) 直接选择<exit>退出并按照提示保存配置文件。

7. 编译内核，生成启动映像文件

内核配置完成后，编译内核，生成启动映像文件：

执行命令：`# make bzImage`

结果可能出现图 1-13 所示的错误：

```
CALL scripts/checksyscalls.sh
HOSTCC scripts/sign-file
scripts/sign-file.c:23:30: fatal error: openssl/opensslv.h: No such file or directory
```

图 1-13 编译错误

这是因为没有安装 openssl 的原因，需要先安装 openssl：

执行命令：`# apt-get install libssl-dev`

openssl 安装完成后，再执行 `make bzImage` 即可，需要很长时间。

编译完成后将在 `/usr/src/linux-4.4.19/arch/x86/boot` 目录下生成一个名为 `bzImage` 的文件。

8. 编译模块

执行命令：`# make modules`

9. 安装内核

(1) 安装模块：`# make modules_install`

(2) 建立要载入 `ramdisk` 的映像文件：如果你的系统是跑在虚拟机里，那么这一步一定需要：

```
# mkinitramfs 4.4.19 -o /boot/initrd-4.4.19.img
```

其中 `4.4.19` 是 `/lib/modules` 下的目录名称，即内核版本号；`initrd-4.4.19.img` 为所生成的内核镜像文件名，可根据自己需要进行修改，上面命令也可以写成：

```
# mkinitramfs -o /boot/initrd-4.4.19.img -v 4.4.19
```

(3) 安装内核：`make install`

10. 配置 grub 引导程序

只需要执行命令：`# update-grub2`，该命令会自动修改 `grub`。

11. 重启系统

执行命令：`# reboot`，将使用新内核启动 `linux`。

启动完成后进入终端查看内核版本，如图 1-14 所示：

```
zwh@ubuntu:~$ uname -a
Linux ubuntu 4.4.19 #1 SMP Tue Sep 27 06:38:17 PDT 2016 x86_64 x86_64 x86_64 GNU
/Linux
zwh@ubuntu:~$
```

图 1-14 查看内核版本

第二章 linux 内核模块编程

2.1 设计目的和内容要求

1. 设计目的

Linux 提供的模块机制能动态扩充 linux 功能而无需重新编译内核，已经广泛应用在 linux 内核的许多功能的实现中。在本实验中将学习模块的基本概念、原理及实现技术，然后利用内核模块编程访问进程的基本信息，从而加深对进程概念的理解、对模块编程技术的掌握。

2. 内容要求

(1) 设计一个模块，要求列出系统中所有内核线程的程序名、PID 号、进程状态及进程优先级。

(2) 设计一个带参数的模块，其参数为某个进程的 PID 号，该模块的功能是列出该进程的家族信息，包括父进程、兄弟进程和子进程的程序名、PID 号。

3. 学时安排（共 6 学时）

4. 开发平台

Linux 环境, gcc, gdb, vim 或 gedit 等。

2.2 linux 内核模块简介

2.2.1 线程基本概念

Linux 内核是单体式结构, 相对于微内核结构而言, 其运行效率高, 但系统的可维护性及可扩展性较差。为此, linux 提供了内核模块 (module) 机制, 它不仅弥补了单体式内核相对于微内核的一些不足, 而且对系统性能没有影响。内核模块的全称是动态可加载内核模块 (Loadable Kernel Module, KLM), 简称为模块。模块是一个目标文件, 能完成某种独立的功能, 但其自身不是一个独立的进程, 不能单独运行, 可以动态载入内核, 使其成为内核代码的一部分, 与其他内核代码的地位完全相同。当不需要某模块功能时, 可以动态卸载。实际上, linux 中大多数设备驱动程序或文件系统都以模块方式实现, 因为它们数目繁多, 体积庞大, 不适合直接编译在内核中, 而是通过模块机制, 需要时临时加载。使用模块机制的另一个好处是, 修改模块代码后只需重新编译和加载模块, 不必重新编译内核和引导系统, 降低了系统功能的更新难度。

一个模块通常由一组函数和数据结构组成, 用来实现某种功能, 如实现一种文件系统、一个驱动程序或其他内核上层的功能。模块自身不是一个独立的进程, 当前进程运行是调用到模块代码时, 可以认为该段代码就代表当前进程在核心态运行。

2.2.2 内核符号表

模块编程可以使用内核的一些全局变量和函数, 内核符号表就是用来存放所有模块都可以访问的符号及相应地址的表, 其存放位置在 `/proc/kallsyms` 文件中, 我们可以使用 “`cat /proc/kallsyms`” 命令查看当前环境下导出的内核符号。

通常情况下, 一个模块只需实现自己的功能, 而无需导出任何符号; 但如果其他模块需要调用这个模块的函数或数据结构时, 该模块也可以导出符号。这样, 其他模块可以使用由该模块导出的符号, 利用现成的代码实现更加复杂的功能, 这种技术也称作模块层叠技术, 当前已经使用在很多主流的内核源代码中。

如果一个模块需要向其他模块导出符号, 可使用下面的宏:

```
EXPORT_SYMBOL(symbol_name);
```

```
EXPORT_SYMBOL_GPL(symbol_name);
```

这两个宏均用于将给定的符号导出到模块外部。_GPL 版本使得要导出的符号只能被 GPL 许可证下的模块使用。符号必须在模块文件的全局部分导出, 不能在模块中的某个函数中导出。

2.3 内核模块编程基础

我们以一个简单的 “hello world” 模块的实现为例, 来说明内核模块的编写结构、编译及加载过程。

2.3.1 模块代码结构

“hello world” 的示例代码如下：

```

1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4
5  static int  hello_init(void)
6  {
7      printk(KERN_ALERT"hello,world\n");
8      return 0;
9  }
10 static void  hello_exit(void)
11 {
12     printk(KERN_ALERT"goodbye\n");
13 }
14
15 module_init(hello_init);
16 module_exit(hello_exit);
17 MODULE_LICENSE("GPL");

```

上面的代码是一个内核模块的典型结构。该模块被载入内核时会向系统日志文件中写入“hello, world”；当被卸载时，也会向系统日志中写入“goodbye”。下面说明该模块代码的结构组成：

(1) 头文件声明：

第 1、2 行是模块编程的必需头文件。module.h 包含了大量加载模块所需要的函数和符号的定义；init.h 包含了模块初始化和清理函数的定义。如果模块在加载时允许用户传递参数，模块还应该包含 moduleparam.h 头文件。

(2) 模块许可申明：

第 17 行是模块许可声明。Linux 内核从 2.4.10 版本内核开始，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，会收到内核被污染“kernel tainted”的警告。从 linux/module.h 文件中可以看到，被内核接受的有意义的许可证有“GPL”，“GPL v2”，“GPL and additional rights”，“Dual BSD/GPL”，“Dual MPL/GPL”，“Proprietary”，其中“GPL”是指明这是 GNU General Public License 的任意版本，其他许可证大家可以查阅资料进一步了解。MODULE_LICENSE 宏声明可以写在模块的任何地方（但必须在函数外面），不过惯例是写在模块最后。

(3) 初始化与清理函数的注册：

内核模块程序中没有 main 函数，每个模块必须定义两个函数：一个函数用来初始化（示例第 5 行），主要完成模块注册和申请资源，该函数返回 0，表示初始化成功，其他值表示失败；另一个函数用来退出（示例第 10 行），主要完成注销和释放资源。Linux 调用宏 module_init 和 module_exit 来注册这两个函数，如示例中第 15、16 两行代码，module_init 宏标记的函数在加载模块时调用，module_exit 宏标记的函数在卸载模块时调用。需要注意的是，初始化与清理函数必须在宏 module_init 和 module_exit 使用前定义，否则会出现编译错误。

初始化函数通常定义为：

```
static int __init init_func(void)
{
    //初始化代码
}
Module_init(init_func);
```

一般情况下，初始化函数应当申明为 `static`，以便它们不会在特定文件之外可见。如果该函数只是在初始化使用一次，可在声明语句中加 `__init` 标识，则模块在加载后会丢弃这个初始化函数，释放其内存空间。

清理函数通常定义为：

```
static void __exit exit_func(void)
{
    //清理代码
}
Module_exit(exit_func);
```

清理函数没有返回值，因此被声明为 `void`。声明语句中的 `__exit` 的含义与初始化函数中的 `__init` 类似，不再重述。

一个基本的内核模块只要包含上述三个部分就可以正常工作了。

(4) `printk()` 函数说明：

大家可能已经发现在代码第 3 行还有一个头文件 “`<linux/kernel.h>`”，这不是模块编程必须的，而是因为在代码中使用了 `printk()` 函数（第 7、12 行），在该头文件中包含了 `printk()` 的定义。

`Printk()` 会依据日志级别将指定信息输出到控制台或日志文件中，其格式为：

```
printk(日志级别 "消息文本");
如 printk(KERN_ALERT"hello,world\n");
```

一般情况下，优先级高于控制台日志级别的消息将被打印到控制台，优先级低于控制台日志级别的消息将被打印到 `messages` 日志文件中，而在伪终端下不打印任何的信息。有关其更详细的使用说明请大家自行查阅资料学习。

加载模块后，用户可使用 `dmesg` 命令查看模块初始化函数中的输出信息，如使用 “`dmesg | tail -20`” 来输出 “`dmesg`” 命令的最后 20 行日志。

2.3.2 模块编译和加载

1. 模块编译的 Makefile 文件：

在 `linux2.6` 及之后的内核中，模块的编译需要配置过的内核源代码，否则无法进行模块的编译工作；编译、链接后生成的内核模块后缀为 `.ko`；编译过程首先会到内核源目录下读取顶层的 `Makefile` 文件，然后返回模块源代码所在的目录继续编译。

在使用 `make` 命令编译模块代码时，应先书写 `Makefile` 文件，且应放在模块源代码文件所在目录中。针对上面的 “`hello world`” 模块，编写的一个简单 `Makefile`：

```
obj-m :=hello.o           //生成的模块名称是：hello.ko
KDIR :=/lib/modules/$(shell uname -r)/build
PWD :=$(shell pwd)       // PWD 是当前目录
default:
```

`make -C $(KDIR) M=$(PWD) modules` // -C 指定内核源码目录，M 指定模块源码目录

clean:

```
make -C $(KDIR) M=$(PWD) clean
```

其中: KDIR 是内核源码目录, 该目录通过当前运行内核使用的模块目录中的 build 符号链接指定。或者直接给出源码目录也可以, 如: KDIR := /usr/src/linux-headers-4.4.0-36-generic.

2. 由多个文件构成的内核模块的 Makefile 文件:

当模块的功能较多时, 把模块的源代码分成几个文件是一个明智的选择, 如下面的示例:

hello.c:

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

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

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

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

```
void hello2print(void); //来源于第二个.c 文件: hello2print.c
```

```
static int __init hello_init(void)
```

```
{
```

```
    printk(KERN_ALERT"hello,world\n");
```

```
    hello2print();
```

```
    return 0;
```

```
}
```

```
static void __exit hello_exit(void)
```

```
{
```

```
    printk(KERN_ALERT"goodbye\n");
```

```
}
```

```
module_init(hello_init);
```

```
module_exit(hello_exit);
```

hello2.c

```
#include "linux/kernel.h"
```

```
void hello2print(void)
```

```
{
```

```
    printk(KERN_ALERT"this is hello2 print\n");
```

```
}
```

则 Makefile 相应的行改为: obj-m :=hello.o

```
hello-objs :=hello1.o hello2.o
```

其中含有 module_init(hello_init);module_exit(hello_exit);这两个宏的.o 模块应放在开始。
完整的 Makefile 文件内容为:

```
obj-m +=hello.o //生成的模块名称是: hello5.ko
```

```
hello-objs :=hello.o hello2.o
```

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

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

```
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

3. 相关操作命令:

以下命令除 make 命令外, 其他都应以 root 用户执行:

(1) 模块编译命令 make:

命令格式: `$make`

不带参数的 make 命令将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。

(2) 加载模块命令 insmod 或 modprobe:

insmod 命令把需要载入的模块以目标代码的形式加载到内核中, 将自动调用 init_module 宏。其格式为:

```
# insmod [filename] [module options...]
```

Modprobe 命令的功能与 insmod 一样, 区别在于 modprobe 能够处理 module 载入的相依问题, 其格式为:

```
# modprobe [module options...] [modulename] [module parameters...]
```

如本示例中加载模块的命令为: `# insmod hello.ko`

(3) 查看模块命令 lsmod:

列出当前所有已载入系统的模块信息, 包括模块名、大小、其他模块的引用计数等信息。命令格式: `# lsmod`

通常会配合 grep 来查看指定模块是否已经加载: `# lsmod | grep 模块名`

(5) 卸载模块命令 rmmod:

卸载已经载入内核的指定模块, 格式为:

```
# rmmod 模块名
```

4. 带参数的模块编程

有时候用户需要向模块传递一些参数, 如使用模块机制实现设备驱动程序时, 用户可能希望在不同条件下让设备在不同状态下工作。

(1) 头文件:

模块要带参数, 则头文件必须包括: `#include <linux/moduleparam.h>`。

(2) module_param () 宏

module_param () 宏的功能是在加载模块时或者模块加载以后传递参数给模块, 其格式为: `module_param(name,type,perm);`

其中: name: 模块参数的名称

type: 模块参数的数据类型

perm: 模块参数的访问权限

更加详细的内容请大家参考其他资料自学。

程序中首先将所有需要获取参数值的变量声明为全局变量; 然后使用宏 module_param () 对所有参数进行说明, 这个宏定义应当放在任何函数之外, 典型地是出现在源文件的前面, 如下面示例程序中的第 8、9 两行。

然后在加载模块的命令后面跟上参数值即可。注意, 必须明确指出哪一个变量的值是

多少，否则系统不能判断。如对本示例，可用如下命令加载模块并传递参数：

```
#insmod module_para.ko who=zwh times=4
```

示例程序：

```
1 #include<linux/init.h>
2 #include<linux/module.h>
3 #include<linux/kernel.h>
4 #include <linux/moduleparam.h>
5 MODULE_LICENSE("GPL");
6 static char *who;    //参数申明
7 static int  times;
8 module_param(who,charp,0644);    //参数说明
9 module_param(times,int,0644);
10 static int __init hello_init(void)
11 {
12     int i;
13     for(i = 1;i <= times;i++)
14         printk("%d  %s!\n",i,who);
15     return 0;
16 }
17 static void __exit hello_exit(void)
18 {
19     printk("Goodbye,%s!\n",who);
20 }
21 module_init(hello_init);
22 module_exit(hello_exit);
```

2.4 实验指南

2.4.1 linux 内核链表结构及操作

链表是 linux 内核中最简单、最常用的一种数据结构。Linux 内核对链表的实现方式与众不同，它给出了一种抽象链表定义，实际使用中可将其嵌入到其他数据结构中，从而演化出所需要的复杂数据结构。

1. 链表的定义：

Linux 中链表的定义为：

```
struct list_head {
    struct list_head *next, *prev;
}
```

这个不含数据域的链表，可以嵌入到任何结构中，形成结构复杂的链表，之后就以此 struct list_head 为基本对象，进行链表的插入、删除、合并、遍历等各种操作。如：

```
struct numlist {
```

```

    int num;
    struct list_head list;
};

```

2. 链表的操作

Linux 内核为抽象链表定义了若干操作，如申明及初始化链表、插入节点、删除节点、合并链表、遍历链表等。本实验只涉及读取内核已有链表，所以这里只介绍链表遍历操作，感兴趣的同学可以查看 `/usr/src/linux4.4.19/include/linux/list.h` 文件学习。

`list.h` 中定义了遍历链表的宏：

```

/* list_for_each - iterate over a list
 * @pos: the &struct list_head to use as a loop cursor.
 * @head: the head for your list.
 */

```

```

#define list_for_each(pos, head) \

```

```

    for (pos = (head)->next; pos != (head); pos = pos->next)

```

这个宏仅仅是找到一个个节点在链表中的偏移位置 `pos`，如图 2-1 所示：

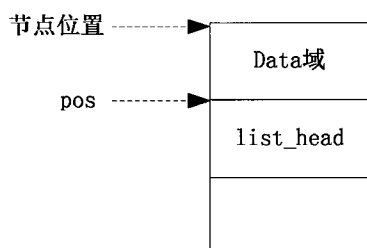


图2-1 `list_for_each()` 宏

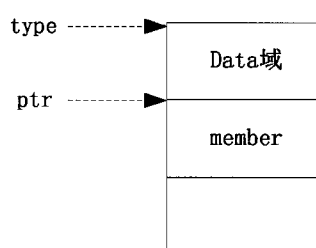


图2-2 `list_entry()` 宏

问题是，如何通过 `pos` 获得节点的起始地址，以便引用节点中的其他域？在 `list.h` 中定义了 `list_entry()` 宏：

```

/* list_entry - get the struct for this entry
 * @ptr: the &struct list_head pointer.
 * @type: the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */

```

```

#define list_entry(ptr, type, member) \

```

```

    container_of(ptr, type, member)

```

其中宏 `container_of()` 定义在 `/usr/src/linux4.4.19/include/linux/kernel.h` 中：

```

#define container_of(ptr, type, member) ({\
    const typeof( ((type *)0)->member ) *__mptr = (ptr);\
    (type *) ( (char *)__mptr - offsetof(type,member) );})

```

该宏的功能是计算返回包含 `ptr` 指向的成员所在的 `type` 类型数据结构的指针，如图 2-2 所示。其实现思路是：计算 `type` 结构体成员 `member` 在结构体中的偏移量，然后用 `ptr` 的值减去这个偏移量，就得出 `type` 数据结构的首地址。

例如前面定义的链表结构：

```

struct numlist {
    int num;
    struct list_head list;
};

```

```

};
可通过如下方式遍历链表的各节点:
struct numlist  numhead  //链表头节点
struct list_head *pos;
struct numlist *p;
list_for_each(pos, &numhead.list) {
    p=list_entry(pos,struct numlist,list);
    //下面可以对 p 指向的 numlist 节点进行相关操作
}

```

2.4.2 进程的 task_struct 结构及家族关系

Linux 进程描述符 task_struct 结构定义在/usr/src/linux4.4.19/include/linux/sched.h 中, 包含众多的成员项, 部分与本实验相关的成员项有: (相关含义自行查阅资料学习)

```

#define TASK_RUNNING      0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define __TASK_STOPPED    4
#define __TASK_TRACED     8
/* in tsk->exit_state */
#define EXIT_DEAD         16
#define EXIT_ZOMBIE       32
/* in tsk->state again */
#define TASK_DEAD         64
#define TASK_WAKEKILL     128
#define TASK_WAKING       256
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    int prio, static_prio, normal_prio;
    unsigned int policy;
    struct list_head tasks; /*线程组长链表, 是节点*/
    struct mm_struct *mm, *active_mm;
    pid_t pid;
    pid_t tgid;
    struct task_struct __rcu *real_parent; /* real parent process */
    struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    cputime_t utime, stime, utimescaled, stimescaled;
    char comm[TASK_COMM_LEN]; /* executable name excluding path */
    .....
};

```

Linux 的进程和轻量级进程/线程均有相应的 task_struct 结构和 PID 号, 而 POSIX 要求同一组线程有统一的 PID, 为此 linux 引入了 tgid (thread group identifier), tgid 实际上是线程

组第一个线程的 pid 值，该线程称为线程组长。对于普通进程，其 pid 与 tgid 是相同的。此外，linux 系统的进程包含一种特殊的类型——内核线程（kernel thread），完成内核的一些特定任务，并始终在核心态运行，没有用户态地址空间，其 task_struct 结构的 mm 成员项为 NULL，如交换进程。

实验内容（1）可以利用内核的线程组长链表实现，每个线程组长通过 task_struct 结构的 tasks 成员加入该链表。Linux 内核提供了宏 for_each_process() 访问链表中的每个进程：

```
#define for_each_process(p) \
    for (p = &init_task; (p = next_task(p)) != &init_task; )
```

实验内容（2）需要了解 linux 进程家族的组织情况。所以的进程都是 PID 为 1 的 init 进程的后代，内核在系统启动的最后阶段创建 init 进程，并由其完成后续启动工作。系统中的每个进程必有一个父进程，相应的，每个进程也可以拥有零个或多个子进程。父进程（task_struct 中的 parent 成员）相同的所有进程称为兄弟进程，由 task_struct 中的 sibling 成员链接成父进程的 children 链表，它们间的关系如图 2-3 所示：

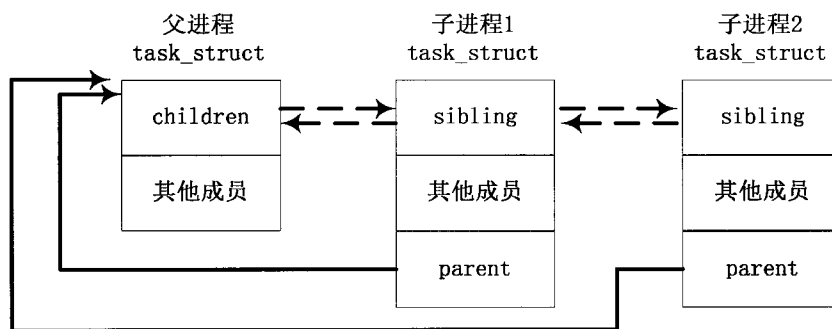


图2-3 进程家族关系

对子进程链表和兄弟进程链表的访问，都可以通过宏 list_for_each 和 list_entry 来实现。对于指定的 pid，可以通过函数 find_task_by_vpid() 找到其相应的 task_struct 结构：

```
struct task_struct *find_task_by_vpid(pid_t vnr)
{
    return find_task_by_pid_ns(vnr, current->nsproxy->pid_ns);
}
```

宏 for_each_process 和函数 find_task_by_vpid() 都定义在文件 linux/sched.h 中。

第三章 设备驱动程序开发（选做）

3.1 设计目的和内容要求

1. 设计目的

Linux 驱动程序占了内核代码的一半以上，开发设计驱动程序是 linux 内核编程的一项很重要的工作。通过本次实验，学习者应了解 linux 的设备管理机制及驱动程序的组织结构；掌握 linux 设备驱动程序的编写流程及加载方法，为从事具体的硬件设备驱动程序开发打下基础。

2. 内容要求

- (1) 编写一个字符设备驱动程序，要求实现对该字符设备的打开、读、写、I/O 控制和关闭 5 个基本操作。为了避免牵涉到汇编语言，这个字符设备并非一个真实的字符设备，而是用一段内存空间来模拟的。以模块方式加载该驱动程序。
- (2) 编写一个应用程序，测试 (1) 中实现的驱动程序的正确性。
- (3) 有兴趣的同学还可以编写一个块设备的驱动程序。

3. 学时安排 (共 6 学时)

4. 开发平台

Linux 环境，gcc，gdb，vim 或 gedit 等。

3.2 linux 设备管理概述

3.2.1 设备文件的概念

Linux 沿用了 Unix 的设备管理思想，将所有设备看成是一类特殊文件，即为每个设备建立一个设备文件，一般保存在 `/dev` 目录下，如 `/dev/hda1` 标识第一个硬盘的第一个逻辑分区。Linux 将系统中的设备分成三类：

- ① 块设备：一次 I/O 操作是固定大小的数据块，可随机存取，其设备文件的属性字段中以“b”进行标识；
- ② 字符设备：只能按字节访问的设备，一次 I/O 操作存取数据量不固定，只能顺序存取，其设备文件的属性字段中以“c”进行标识；
- ③ 网络设备：网卡是特殊处理的，没有对应的设备文件。

3.2.2 设备号的概念

1. 什么是设备号

与普通文件一样，每个设备文件都有文件名和一个唯一的索引节点，在索引节点中记录了与特定设备建立连接所需的信息，其中最主要的三个信息是：

- ① 类型：表明是字符设备还是块设备；
- ② 主设备号：主设备号相同的设备，由同一个驱动程序控制；
- ③ 次设备号：说明该设备是同类设备中的第几个，即表示具体的某个设备。

如查看 `/dev` 目录，可看到如下一些信息：

```
brw-rw---- 1 root disk 8, 0 Oct 7 06:31 sda
brw-rw---- 1 root disk 8, 1 Oct 7 06:31 sda1
brw-rw---- 1 root disk 8, 2 Oct 7 06:31 sda2
brw-rw---- 1 root disk 8, 5 Oct 7 06:31 sda5
```

可见，`sda1`、`sda2`、`sda5` 是同一类块设备，它们的主设备号都是 8，次设备号分别是 1、2 和 5。

由主设备号和次设备号组成了设备的唯一编号：设备号，其类型为 `dev_t`，是一个 32 位的无符号整数，定义在 `/usr/src/linux-4.4.19/include/linux/types.h` 文件中：

```
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t dev_t;
```

2. 与设备号相关的操作函数

(1) 定义在/usr/include/linux/kdev_t.h 中的三个宏：

- ① #define MAJOR(dev) ((dev)>>8) //从 dev (dev_t 类型) 中获得主设备号
- ② #define MINOR(dev) ((dev) & 0xff) //从 dev (dev_t 类型) 中获得次设备号
- ③ #define MKDEV(ma,mi) ((ma)<<8 | (mi)) //将主、次设备号组合成 dev_t 类型的设备号

(2) 为字符设备静态分配设备号：

如果驱动程序开发者清楚了解系统中尚未被使用的设备号，则可直接指定主设备号，然后再申请若干个连续的次设备号。这种方法可能会造成系统中设备号冲突，而使驱动程序无法注册。函数原型定义在/usr/src/linux-4.4.19/include/linux/fs.h 文件中，为：

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

函数功能：为一个指定主设备号的字符驱动程序申请一个或一组连续的次设备号。

输入参数：

first : dev_t 类型的起始设备号（可通过 MKDEV(major,0)获得）；

count: 需要申请的次设备号数量；

name: 设备名，会出现在 /proc/devices 和 sysfs 中；

返回值：分配成功返回 0，失败返回一个负的错误码

(3) 为字符设备动态分配设备号：

如果没有提前指定主设备号，则采用动态申请方式，它不会出现设备号冲突的问题，但是无法在安装驱动前创建设备文件（因为安装前还没有分配到主设备号）。

函数原型定义在/usr/src/linux-4.4.19/include/linux/fs.h 文件中：

```
int alloc_chrdev_region(dev_t *dev,unsigned firstminor,unsigned count,char *name);
```

函数功能：动态分配一个主设备号及一个或一组连续次设备号。

输入参数：

firstminor: 起始次设备号，一般从 0 开始；

count: 需要分配的次设备号数量；

name: 设备名，会出现在 /proc/devices 和 sysfs 中；

输出参数：系统自动分配的 dev_t 类型的设备号；

返回值：分配成功返回 0，失败返回一个负的错误码。

(4) 释放设备号：

采用上面两种方式申请到的设备号，在设备不使用时，比如在调用 cdev_del()函数从系统中注销字符设备之后，应该及时释放掉。释放设备号使用的函数原型定义在/usr/src/linux-4.4.19/include/linux/fs.h 文件中：

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

其中参数含义跟上面（2）中的一样。

(5) 查看设备号使用情况：

当静态分配设备号时，需要查看系统中已经使用掉的设备号，从而决定使用哪个新设备号。可使用命令“cat /proc/devices”查看，显示的字符设备的最后一行信息如下所示：

```
Character devices:
```

```
254      mdp
```

则新添加设备驱动时可以选择主设备号 255。

3.2.3 字符设备管理相关数据结构

1. struct cdev 结构

Linux 内核中使用 struct cdev 来描述一个字符设备，定义在/usr/src/linux-4.4.19/include/linux/cdev.h 文件中：

```
struct cdev {
    struct kobject kobj; /*内嵌的内核对象，包括引用计数、名称、父指针等*/
    struct module *owner; /*所属内核模块，一般设置为 THIS_MODULE */
    const struct file_operations *ops; /*设备操作集合 */
    struct list_head list; /*设备的 inode 链表头 */
    dev_t dev; /*设备号 */
    unsigned int count; /*分配的设备号数目 */
};
```

cdev 结构是内核对字符设备的标准描述，在实际的设备驱动开发中，通常使用自定义的结构体来描述一个特定的字符设备：内嵌 cdev 结构，同时包含其他描述该具体设备特性的字段。比如本实验中，用一段内存来模拟字符设备：

```
struct mymem_dev
{
    Struct cdev cdev;
    Unsigned char mem[512];
};
```

2. struct char_device_struct 结构

内核为主设备号相同的一组设备设置一个 char_device_struct 结构，描述这个主设备号下已经被分配的次设备号区间：

```
static struct char_device_struct {
    struct char_device_struct *next; /* 指向散列链表中的下一个元素的指针*/
    unsigned int major; /* 主设备号*/
    unsigned int baseminor; /* 起始次设备号*/
    int minorct; /* 次设备号区间大小*/
    char name[64]; /* 设备名*/
    struct file_operations *fops; /* 未使用*/
    struct cdev *cdev; /* 指向字符设备描述符的指针*/
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

3. file_operations 结构

file_operations 结构体是字符设备中最重要的数据结构之一。其中的成员是一组函数指针，用于实现相应的系统调用，如 open()、read()、write()、close()、seek()、ioctl()等系统调用最终就是这组函数实现的，是字符设备驱动程序设计的主体内容。

file_operations 结构体中对字符设备比较重要的成员主要有：

```
struct file_operations {
    struct module *owner; /*拥有该结构的模块，一般为 THIS_MODULE*/
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); /*从设备中读取数据*/
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *); /*向设备中写数据*/
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long); /*执行设备的 I/O
```

```

控制命令*/
    int (*open) (struct inode *, struct file *); /*打开设备文件*/
    int (*release) (struct inode *, struct file *); /*关闭设备文件*/
    .....
};

```

4. file 结构

file 结构代表一个打开的文件，内核每执行一次 open 操作就会建立一个 file 结构，因此一个文件可以对应多个 file 结构。其中几个重要的成员有：

```

struct file{
    mode_t fmode; /*文件模式，如 FMODE_READ, FMODE_WRITE*/
    loff_t f_pos; /*当前读写指针*/
    struct file_operations *f_op; /*文件操作函数表指针*/
    void *private_data; /*非常重要，用于存放转换后的设备描述结构指针*/
    .....
};

```

5. inode 结构

磁盘上每个文件都有一个 inode，对于设备文件来说，有两个很重要的成员：

```

struct inode{
    dev_t i_rdev; /*设备号*/
    struct cdev *i_cdev; /*该设备的 cdev 结构*/
    .....
};

```

根据其设备号可以得到其主设备号和次设备号。

3.3 linux 字符设备驱动程序的设计

3.3.1 linux 字符设备驱动程序框架

Linux 字符设备驱动程序框架如图 3-1 所示：

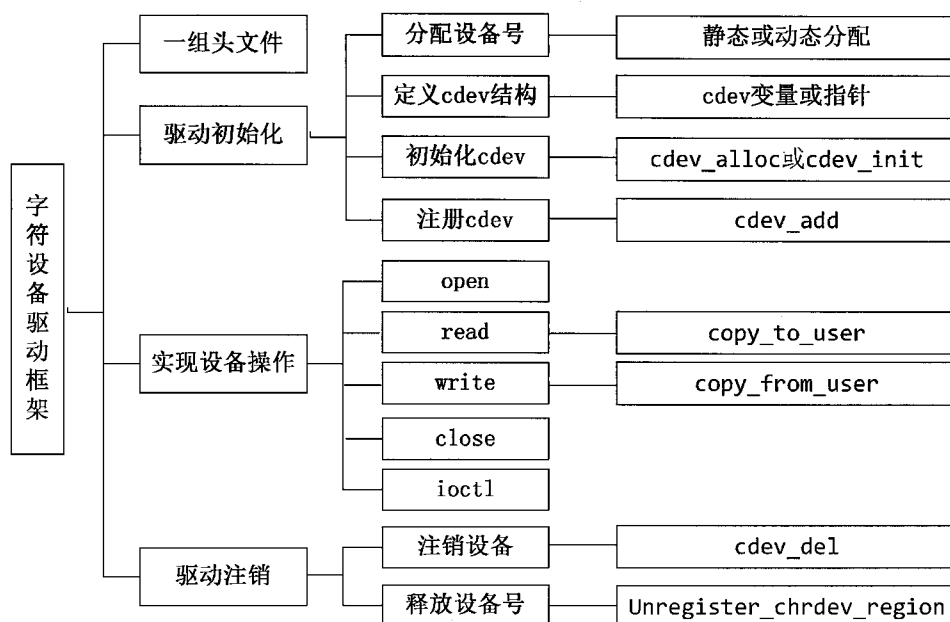


图 3-1 字符设备驱动程序框架图

3.3.2 linux 字符设备驱动程序中需要的一组头文件

在编写 linux 字符设备驱动程序时，可能要用到的头文件包括：

```

#include <linux/fs.h> //定义文件表结构（file 结构,buffer_head,m_inode 等）
#include <linux/types.h> //对一些特殊的系统数据类型的定义，例如 dev_t, off_t, pid_t.其实这些类型大部分都是 unsigned int 型通过一连串的 typedef 变过来的，只是为了方便阅读。
#include <linux/cdev.h> //包含了 cdev 结构及相关函数的定义。
#include <asm/uaccess.h> //包含 copy_to_user(),copy_from_user()的定义
#include <linux/module> //模块编程相关函数
#include <linux/init.h> //模块编程相关函数
#include <linux/kernel>
#include <linux/slab.h> //包含内核的内存分配相关函数，如 kmalloc()/kfree()等
  
```

3.3.3 字符设备驱动程序的初始化

1. 分配设备号

如 3.2.2 节所述，为一个新字符设备分配设备号可以有静态和动态两种方式，如果提前指定主设备号，则使用静态方式：

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

否则使用动态分配方式：

```
int alloc_chrdev_region(dev_t *dev,unsigned firstminor,unsigned count,char *name);
```

2. 定义 cdev 结构并初始化

linux 内核必须为每个字符设备都建立一个 cdev 结构，定义时采用 cdev 结构体指针或变量均可，只不过两种定义方式的初始化操作会有所不同：

(1) 定义 cdev 结构体及初始化：

```
struct cdev my_cdev;
```

```

    cdev_init(&my_cdev, &fops);
    my_cdev.owner = THIS_MODULE;
(2) 定义 cdev 结构指针及初始化:
    struct cdev *my_cdev = cdev_alloc();
    my_cdev->ops = &fops;
    my_cdev->owner = THIS_MODULE;

```

其实, `cdev_init()` 和 `cdev_alloc()` 的功能是差不多的, 只是前者多了一个 `ops` 的赋值操作, 具体区别参看下面两个函数的实现代码:

```

struct cdev *cdev_alloc(void)
{
    struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
    if (p) {
        INIT_LIST_HEAD(&p->list);
        kobject_init(&p->kobj, &ktype_cdev_dynamic);
    }
    return p;
}

void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
    memset(cdev, 0, sizeof *cdev);
    INIT_LIST_HEAD(&cdev->list);
    kobject_init(&cdev->kobj, &ktype_cdev_default);
    cdev->ops = fops;
}

```

两个函数的原型定义在 `/usr/src/linux-4.4.19/include/linux/cdev.h` 文件中。

3. 注册 cdev 结构

`cdev` 初始化完成后, 应将其注册到系统中, 一般在模块加载时完成该操作。设备注册函数是 `cdev_add()`, 其原型定义在 `/usr/src/linux-4.4.19/include/linux/cdev.h` 文件中:

```

int cdev_add(struct cdev *p, dev_t dev, unsigned count)
{
    p->dev = dev;
    p->count = count;
    return kobj_map(cdev_map, dev, count, NULL, exact_match, exact_lock, p);
}

```

其中的输入参数分别是 `cdev` 结构指针、起始设备号、次设备号数量

linux 内核中所有字符设备都记录在一个 `kobj_map` 结构的 `cdev_map` 散列表里。`cdev_add()` 函数中的 `kobj_map()` 函数就是用来把设备号及 `cdev` 结构一起保存到 `cdev_map` 散列表里。当以后要打开这个字符设备文件时, 通过调用 `kobj_lookup()` 函数, 根据设备号就可以找到 `cdev` 结构变量, 从而取出其中的 `ops` 字段。

执行 `cdev_add()` 操作后, 意味着一个字符设备对象已经加入了系统, 以后用户程序可以通过文件系统接口找到对应的驱动程序。

3.3.4 实现字符设备驱动程序的操作函数

1. 实现 file_operations 结构中要用到的函数

这些函数具体实现设备的相关操作，如打开设备、读设备等，部分函数的大致结构可参看下面的描述：

- (1) 打开设备函数 open:

```
static int char_dev_open(struct inode *inode, struct file *filp)
{
    // 这里可以进行一些初始化
    printk("char_dev device open.\n");
    return 0;
}
```

- (2) 读设备函数

```
ssize_t char_dev_read(struct file *file, char __user *buff, size_t count, loff_t *offp)
{
    ...
    copy_to_user();
    ...
}
```

- (3) 写设备函数

```
ssize_t char_dev_write(struct file *file, const char __user *buff, size_t count, loff_t *offp)
{
    ...
    copy_from_user();
    ...
}
```

- (4) I/O 控制函数

```
static int char_dev_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
unsigned long arg)
{
    ...
    switch(cmd)
    {
        case xxx_cmd1:
            ...
            break;
        case xxx_cmd2:
            ...
            break;
        ...
    }
}
```

- (5) 关闭设备函数 release，对应用户空间的 close 系统调用

```
static int char_dev_release(struct inode *inode, struct file *file)
{
    ...
}
```



```
// 这里可以进行一些资源的释放
printk("char_dev device release.\n");
return 0;
}
```

2. 添加 file_operations 成员

file_operations 结构体中包含很多函数指针，是驱动程序与内核的接口，下面列出最常用的几种操作：

```
static struct file_operations char_dev_fops =
{
    .owner = THIS_MODULE,
    .open = char_dev_open,    // 打开设备
    .release = char_dev_release, // 关闭设备
    .read = char_dev_read,    // 实现设备读功能
    .write = char_dev_write,  // 实现设备写功能
    .ioctl = char_dev_ioctl,  // 实现设备控制功能
};
```

3.3.5 注销设备

当不使用某个设备时，应及时从系统注销，以节省系统资源。注销设备主要包括两个操作：撤销cdev结构和释放设备号，此项工作通常放在模块卸载过程中完成。

1. 撤销 cdev 结构

Linux内核使用cdev_del()函数向系统删除一个cdev，完成字符设备的注销：

```
void cdev_del(struct cdev *p)
{
    cdev_unmap(p->dev, p->count); //调用 kobj_unmap()释放 cdev_map散列表中的对象
    kobject_put(&p->kobj); //释放 cdev结构本身
}
```

2. 释放设备号

调用cdev_del()函数从系统注销字符设备之后，应调用unregister_chrdev_region()释放原先申请的设备号，其函数原型为：

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

3.4 linux 字符设备驱动程序的编译及加载

当以模块方式实现一个字符设备的驱动程序后，可从按以下步骤对驱动程序进行编译和加载：

1. 编译模块

在驱动程序源码文件所在目录中建立Makefile文件，参考内容如下：

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

```
KDIR :=/usr/src/linux-headers-4.4.0-36-generic
PWD :=$(shell pwd)
```

default:

```
make -C $(KDIR) M=$(PWD) modules
```

clean:

```
make -C $(KDIR) M=$(PWD) clean
```

然后使用make命令编译模块，得到.ko文件。

2. 使用 insmod 命令加载模块（需要 root 权限）

加载后可使用“cat /proc/devices”查看所加载的设备

3. 建立设备节点（即设备文件）

根据设备号在文件系统中建立对应的设备节点（即设备文件），使用命令 mknod，如：

```
#mknod /dev/mycdev c 145 0
```

从而建立了/dev/mycdev 文件与（145,0）号设备的连接

4. 可根据需要修改设备文件的权限

如：#chmod 777 /dev/mycdev

至此，一个新设备建立完毕，以后应用程序就可以使用文件操作函数如“open”等操作/dev/mycdev 设备了。

第四章：linux 进程管理

4.1 设计目的和内容要求

1. 设计目的

（1）熟悉 linux 的命令接口。

（2）通过对 linux 进程控制的相关系统调用的编程应用，进一步加深对进程概念的理解，明确进程和程序的联系和区别，理解进程并发执行的具体含义。

（3）通过 Linux 管道通信机制、消息队列通信机制、共享内存通信机制的使用，加深对不同类型的进程通信方式的理解。

（4）通过对 linux 的 Posix 信号量的应用，加深对信号量同步机制的理解。

2. 设计内容

（1）熟悉 linux 常用命令：pwd, passwd, who, ps, pstree, kill, top, ls, cd, mkdir, rmdir, cp, rm, mv, cat, more, grep。

（2）编写三个不同的程序 cmd1.c, cmd2.c, cmd3.c, 每个程序输出一句话，分别编译成可执行文件 cmd1, cmd2, cmd3。然后再编写一个程序，模拟 shell 程序的功能，能根据用户输入的字符串（表示相应的命令名），去为相应的命令创建子进程并让它去执行相应的程序，而父进程则等待子进程结束，然后再等待接收下一条命令。如果接收到的命令为 exit，则父进程结束；如果接收到的命令是无效命令，则显示“Command not found”，继续

等待。

(3) 由父进程创建一个管道，然后再创建 3 个子进程，并由这三个子进程利用管道与父进程之间进行通信：子进程发送信息，父进程等三个子进程全部发完消息后再接收信息。通信的具体内容可根据自己的需要随意设计，要求能试验阻塞型读写过程中的各种情况，并且要实现进程间对管道的互斥访问。运行程序，观察各种情况下，进程实际读写的字节数以及进程阻塞唤醒的情况。

(4) 编写程序创建两个线程：sender 线程和 receive 线程，其中 sender 线程运行函数 sender()，它创建一个消息队列，然后，循环等待用户通过终端输入一串字符，将这串字符通过消息队列发送给 receiver 线程，直到用户输入“exit”为止；最后，它向 receiver 线程发送消息“end”，并且等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，删除相关消息队列，结束程序的运行。Receiver 线程运行 receive()，它通过消息队列接收来自 sender 的消息，将消息显示在终端屏幕上，直至收到内容为“end”的消息为止，此时，它向 sender 发送一个应答消息“over”，结束程序的运行。使用无名信号量实现两个线程之间的同步与互斥。

(5) 编写程序 sender，它创建一个共享内存，然后等待用户通过终端输入一串字符，并将这串字符通过共享内存发送给 receiver；最后，它等待 receiver 的应答，等到应答消息后，将接收到的应答信息显示在终端屏幕上，删除共享内存，结束程序的运行。编写 receiver 程序，它通过共享内存接收来自 sender 的消息，将消息显示在终端屏幕上，然后再通过该共享内存向 sender 发送一个应答消息“over”，结束程序的运行。使用有名信号量或 System V 信号量实现两个进程对共享内存的互斥使用。

3. 学时安排（共 6 学时）

4. 开发平台

Linux 环境，gcc，gdb，vim 或 gedit 等。

5. 思考

- (1) OS 向用户提供的命令接口、图形接口和程序接口分别适用于哪些场合？
- (2) 系统调用和用户自己编制的子函数有什么区别？通常操作系统提供的 API 与系统调用有什么联系和区别？
- (3) 进程和程序有何联系，又有哪些区别？
- (4) 一个进程从出生到终止，其状态会经历哪些变化？
- (5) 用户可如何取得进程的控制信息？
- (6) 当首次将 CPU 调度给子进程时，它将从哪里开始执行指令？
- (7) 虽然父子进程可以完全并发执行，但在 Linux 中，创建子进程成功之后，通常让子进程优先获得 CPU，这种做法有什么好处？
- (8) 僵尸进程通常是如何形成的？
- (9) 对一个应用，如果用多个进程的并发执行来实现，与单个进程来实现有什么不同？
- (10) 有名管道和无名管道之间有什么不同？
- (11) 管道的读写与文件的读写有什么异同？
- (12) Linux 消息队列通信机制中与教材中的消息缓冲队列通信机制存在哪些异同？
- (13) linux 中 posix 信号量与 System V 信号量有什么区别？

4.2 linux 编程基础知识

4.2.1 Linux 的在线帮助 man

Linux 提供了丰富的帮助手册，当你需要查看某个命令的参数时不必到处上网查找，只要 man (man 为 manual 的简写) 一下即可。

1. 简单范例

例如，如果你不清楚 `pwd` 命令的用法，你可以在命令提示符下直接输入命令：`man pwd`，马上就会有 `pwd` 的详细资料提供给你：

PWD (1)

User commands

Date(1)

NAME

`pwd` - print name of current/working directory

SYNOPSIS

`pwd [OPTION]`

DESCRIPTION

NOTE: your shell may have its own version of `pwd` which will supercede the version described here. Please refer to your shell's documentation for details about the options it supports.

Print the full filename of the current working directory.

-help

display this help and exit

-version

output version information and exit

AUTHOR

Written by Jim Meyering.

REPORTING BUGS

Report bugs to <bug-coreutils@gnu.org>.

COPYRIGHT

Copyright © 2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

SEE ALSO

The full documentation for `pwd` is maintained as a Texinfo manual. If the `info` and `pwd` programs are properly installed at your site, the command `info coreutils pwd`

should give you access to the complete manual.

2. man page 说明

前面的范例中，第一行名字后的数字：

"1"表示用户命令

"2"表示系统调用

"3"表示 C 语言库函数

"4"表示设备或特殊文件

"5"表示文件格式和规则

"6"表示游戏及其他

"7"表示宏、包及其他杂项

"8"表示系统管理员相关的命令

可见，man 不仅可以查询命令，还可以查询系统调用、C 语言库函数、配置文件的格式、系统管理员可用的管理命令等。值得注意的是 man 是按照手册的章节号的顺序进行搜索的，比如：

man sleep

只会显示 sleep 命令的手册，如果想查看库函数 sleep，就要输入使用 man 3 sleep。

若想知道 sleep 系统调用需要哪些头文件，则要输入 man 2 sleep

通常，man page 大致分几个部分：

NAME	简短的命令、数据名称说明
SYNOPSIS	简短的命令语法简介
DESCRIPTION	较为完整的说明，这部分最好仔细看看、
OPTIONS	针对 SYNOPSIS 部分中，列举说明所有可用的参数
COMMANDS	当这个程序（软件）在执行的时候，可以在此程序（软件）中发出的命令
FILES	这个程序或数据所使用、参考或连接的某些参考说明
SEE ALSO	与这个命令或数据相关的其他参考说明、
EXAMPLE	一些可以参考的范例
BUGS	是否有相关的错误

3. man page 中可以使用的常用按键

在 man 中的按键使用：

空格键	向下翻一页
[Page Down]	向下翻一页
[Page Up]	向上翻一页
[Home]	到第一页
[End]	到最后一页
/word	向下搜索 word 字符串，如果要搜索 date 的话，就输入/date
?word	向上搜索 word 字符串
n,N	使用/或?来搜索字符串时，可以用 n 来继续下一个搜索（不论是/

还是?), 可以使用 N 来进行“反向”搜索。

举例来说, 我以 /date 搜索 date 字符串, 那么可以用 n 继续往下查询, 用 N 往上查询。若以 ?date 向上查询 date 字符串, 可以用 n 继续 “向上”查询, 用 N 反向查询

q 结束并退出 man page

4. 互联网上的在线 Linux man 手册

即使不在 Linux 下, 也可以通过某些网站在线查询某个 Linux 的命令, 如:

<http://www.linuxmanpages.com/>

在这里有非常全的 Linux 的 man 信息, 你可以分 1—8 来查看相应的 manual 。

4.2.2 vi 和 vim 编辑器

在计算机系统中, 编辑文本文件是用户经常要进行的操作。所谓文本文件指的是由 ASCII 码字符构成的文件。vi 编辑器是 Unix/Linux 系统提供的文本编辑器, 用于创建和修改文本文件。vi 编辑器与其他字处理软件不同, 它不包含任何格式方面的信息, 如粗体、居中或者下划线等。

vi 编辑器是一个全屏编辑器, 用户可以在整个文档范围内自由移动光标进行编辑操作。vi 编辑器中有 100 多个命令可供用户使用, 提供了丰富的编辑功能, 当然对于学习使用者来说也是个挑战。但是不必灰心, 因为只有少数一些命令是必须使用, 或者使用频繁的, 所以只要熟练掌握这些常用命令就可以完成大部分文本文件的编辑任务了。

vim 可以当做 vi 的升级版, vi 的命令几乎都可以在 vim 上使用。Vim 会依据文件扩展名或文件的开头信息来判断文件内容, 而自动执行该程序的语法判断, 再以颜色来显示程序代码和一般信息。因此 vim 用于程序编辑更加方便。

在系统提示符 (\$、#) 下, 输入: vi <文件名>, vi 可以自动载入所要编辑的文件或创建一个新文件 (若该文件不存在)。

vi 的三种工作模式

vi 编辑器有三种工作模式: 一般模式、编辑模式和命令模式:

(1) **一般模式**。以 vi 打开一个文件就直接进入一般模式了(这是默认的模式)。在这个模式中, 你可以使用『上下左右』按键来移动光标, 你可以使用『删除字符』或『删除整行』来处理文件内容, 也可以使用『复制、贴贴』来处理你的文件数据。

即在一般模式中可以进行删除、复制、粘贴等动作, 但却无法编辑文件的内容。

(2) **编辑模式**。在一般模式下, 按下『i, l, o, O, a, A, r, R』等任何一个字母后就会进入编辑模式。通常, 在按下这些字母后, 在画面的左下方会出现『INSERT 或 REPLACE』的字样, 如图 5-1 所示, 此时才可以进行编辑。而要回到一般模式, 则必须按下『ESC』按键, 即可退出编辑模式, 返回一般模式。

而读取、保存、大量字符替换、离开 vi、显示行号等操作也是在这模式下完成的。

使用范例

● 范例 1:

(1) 在 Linux 命令行界面下输入命令: vi ccc.c 后便进入 vi 的一般模式。如图 5-1 所示, vi 的界面分为上下两部分, 上半部分显示的是文件的实际内容, 而下半部, 即最下面的一行则显示一些状态信息。如果 ccc.c 是一个原来不存在的文件, 状态行中会显示『“ccc.c” [New File]』表示它是一个新文件, 否则, 会显示出被编辑文件的行数、字符数等信息。

(2) 按 “i” 进入编辑模式, 此时, 状态行中会出现『INSERT』的字样, 便可以开始编辑文字了。此时, 你输入的除了“ESC”以外的所有信息都被视为文件的内容, 比如, 如图

5.1 那样，你可以输入：

```
#include <stdio.h>
main(){
    printf("Hello,World!\n");
}
```

(3) 按“ESC”回到一般模式，此时，状态行中的『 INSERT』不见了。

(4) 在一般模式中输入“: wq”保存文件的内容并离开 vi。

● 范例 2

(1) 在 Linux 命令行界面下输入命令： vi file1.txt

(2) 按“i”进入编辑模式，输入下列字符，并保存文件。

You raise me up, so I can stand on mountains;

You raise me up, to walk on stormy seas;

I am strong, when I am on your shoulders;

You raise me up: To more than I can be.

● 范例 3

通过 vi 编辑器编辑一个 syscall.c 文件，其内容如下：

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

int main(){

    int fd=0, i;
    char buf[10];

    fd=open("file1.txt",O_RDONLY);
    if(fd == -1) printf("Cannot open file!\n");

    while((i=read(fd,buf,sizeof(buf)-1))>0){
        buf[i]='\0';
        printf("%s",buf);
    }
}
```

(3) 按“ESC”回到一般模式。

(4) 在一般模式中输入“: wq”保存文件的内容并离开 vi。

一般模式下的常用按键

(1) 光标移动

vi 可以直接用键盘上的光标键来上下左右移动，但正规的 vi 是用小写英文字母。

h、j、k、l，分别控制光标左、下、上、右移一格。

按 Ctrl+B：屏幕往后移动一页。[常用]

按 Ctrl+F：屏幕往前移动一页。[常用]

按 Ctrl+U：屏幕往后移动半页。

按 Ctrl+D：屏幕往前移动半页。

按 0 (数字零): 移动文章的开头。[常用]
 按 G: 移动到文章的最后。[常用]
 按 w: 光标跳到下个单词的开头。[常用]
 按 e: 光标跳到下个单词的字尾。
 按 b: 光标回到上个单词的开头。
 按 \$: 移到光标所在行的行尾。[常用]
 按 ^: 移到该行第一个非空白的字符。
 按 O: 移到该行的开头位置。[常用]
 按 #: 移到该行的第#个位置, 例: 51、121。[常用]

(2) 删除

x: 每按一次删除光标所在位置的后面一个字符。[超常用]
 #x: 例如, 6x 表删除光标所在位置的后面 6 个字符。[常用]
 X: 大写的 X, 每按一次删除光标所在位置的前面一个字符。
 #X: 例如, 20X 表删除光标所在位置的前面 20 个字符。
 dd: 删除光标所在行。[超常用]
 #dd: 例如, 6dd 表删除从光标所在的该行往下数 6 行之文字。[常用]

(3) 复制 与 粘贴

yw: 将光标所在处到字尾的字符复制到缓冲区中。
 yy: 复制光标所在行。[超常用]
 #yy: 如: 6yy 表示拷贝从光标所在的该行往下数 6 行之文字。[常用]
 p: 将已复制的内容粘贴在光标后的位置
 P: 将已复制的内容粘贴在光标前的位置

(4) 替换

r: 取代光标所在处的字符: [常用]
 R: 取代字符直到按 Esc 为止。

(5) 撤销和重做

u: 假如您误操作一个指令, 可以马上按 u, 可撤销前一个操作。[超常用]
 U: 撤销当前行上最近的所有操作。
 ctrl+r: 重做上一个操作。
 .: 点号可以重复执行上一次的指令。

(6) 更改

cw: 更改光标所在处的字到字尾\$处。
 c#w: 例如, c3w 代表更改 3 个字。

3. 退出 vi 编辑器

(7) 切换到编辑模式

i: 进入插入模式, 在当前光标前插入
 I: 进入插入模式, 在当前行首插入
 a: 进入插入模式, 在当前光标后插入
 A: 进入插入模式, 在当前行尾插入
 o: 进入插入模式, 在当前行之下新开一行
 O: 进入插入模式, 在当前行之上新开一行
 r: 进入替换模式, 替换当前字符
 R: 进入替换模式, 替换当前字符及其后的字符, 直至按 ESC 键

命令模式下的常用命令

在一般模式下，按“:”便可切换到命令模式，然后可以使用下述命令模式下的命令：

q 命令：在没有任何修改操作发生的情况下，该命令可以退出 vi 编辑器。

q!命令：不保存文件，强制退出 vi 编辑器。

w 命令：保存文件。

wq 命令：保存文件，然后退出 vi 编辑器

w[文件名] 命令：将编辑后的文件另存到指定文件中。

r[文件名] 命令：将指定文件的内容读入，并添加到当前文件光标所在的位置后面。

4.2.3 Linux 环境下 C 编程

1. GCC 概述

GCC 是 linux 下最常用的编译器，也能运行在 unix、solaris、windows 等下，支持多种语言的编译，如 C、C++、Object C 等语言编写的程序。

gcc 对文件的处理需要经过预处理->编译->汇编->链接的步骤，从而产生一个可执行文件。预处理阶段主要是在库中寻找头文件，包含到待编译的文件中；编译阶段检查程序的语法；汇编阶段将源代码翻译成机器语言；链接阶段将所有的目标代码连接成一个可执行程序。各阶段对应不同的文件类型，具体如下：

file.c	c 程序源文件
file.i	c 程序预处理后文件
file.cxx	c++程序源文件，也可以是 file.cc / file.cpp / file.c++
file.ii	c++程序预处理后文件
file.h	c/c++头文件
file.s	汇编程序文件
file.o	目标代码文件

2. GCC 使用格式：

gcc [参数选项] [文件名]

其中文件名是要编译的文件名称。

3. GCC 遵循的部分后缀越大规则：

当调用gcc时，gcc根据待编译文件的扩展名（后缀）自动识别文件的类别，并调用对应的编译器。Gcc遵循的部分后缀约定规则如表所示：

后缀	约定规则
.c	C语言源代码文件
.a	由目标文件构成的档案库文件
.C .cc .cxx	C++源代码文件
.h	程序包含的头文件
.i	已经预处理过的C源代码文件
.ii	已经预处理过的C++源代码文件
.m	Objective-C源代码文件
.o	编译后的目标文件
.s	汇编语言源代码文件

.S	经过预编译的汇编语言源代码文件
----	-----------------

1. GCC的参数说明:

GCC 参数很多,最常用的参数如下:

-c 仅编译或汇编,生成目标代码文件,将.c、.i、.s 等文件生成.o 文件,其余文件被忽略

-S 仅编译,不进行汇编和链接,将.c、.i 等文件生成.s 文件,其余文件被忽略

-E 仅预处理,并发送预处理后的.i 文件到标准输出,其余文件被忽略

-o file 创建可执行文件并保存在 file 中,而不是默认文件 a.out

-g 产生用于调试和排错的扩展符号表,用于 GDB 调试,切记-g 和-O 通常不能一起使用

-w 取消所有警告

-W 给出更详细的警告

-O [num] 优化,可以指定 0-3 作为优化级别,级别 0 表示没有优化

-x language 默认为-x none,即依靠后缀名确定文件类型,加上-x lan 确定后面所有文件类型,直到下一个-x 出现为止

-I dir 将 dir 目录加到搜寻头文件的目录中去,并优先于 gcc 中缺省的搜索目录,有多个-I 选项时,按照出现顺序搜索

-L dir 将 dir 目录加到搜索-lname 选项指定的函数库文件的目录列表中去,并优先于 gcc 缺省的搜索目录,有多个-L 选项时,按照出现顺序搜索

-lname 在链接时使用函数库 libname.a,链接程序在-L dir 指定的目录和/lib、/usr/lib 目录下寻找该库文件,在没有使用-static 选项时,如果发现共享函数库 libname.so,则使用 libname.so 进行动态链接

-fPIC 产生位置无关的目标代码,可用于构造共享函数库

-static 禁止与共享函数库链接

-shared 尽量与共享函数库链接(默认)

4. 使用范例介绍

(1) 简单范例 1

(1) 按“<5>vi 编辑器”的简单范例 1,先通过 vi 编辑器编辑好文件 ccc.c。

(2) 使用 Linux 命令行界面,在命令提示符后,输入命令:

```
gcc ccc.c
```

功能:对源程序 ccc.c 进行编译链接,产生对应的可执行文件,文件名缺省为: a.out。

也可以通过选项-o 在编译命令中指定可执行文件名,如:

```
gcc -o ccc ccc.c
```

将产生可执行文件 ccc,而不是默认的 a.out。

(3) 在命令提示符后,输入命令: ./a.out 或 ./ccc 运行当前目录下可执行文件 a.out (或 ccc)。

上述范例的运行结果如图 6-1 所示。

```

user@hd-vmpr:~
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[user@hd-vmpr ~]$ gcc ccc.c
[user@hd-vmpr ~]$ gcc -o ccc ccc.c
[user@hd-vmpr ~]$ ./a.out
Hello, World!
[user@hd-vmpr ~]$ ./ccc
Hello, World!
[user@hd-vmpr ~]$

```

(2) 简单范例 2

(1) 先按“<五>vi 编辑器”的简单范例 2 和简单范例 3 通过 vi 编辑器编辑好文件 file1.txt, 和 systemcall.c; 然后用 Gcc 编译器对 systemcall.c 进行编译, 生成可执行文件, 运行之。

(2) 阅读上述 C 源程序, 分析它到底是完成什么功能的; 再观察程序的运行结果来验证你的理解是否正确。

(3) 思考上述程序中文件打开操作和读文件操作到底是谁具体完成的, 并用 man 查阅 open, read, printf, 了解它们的具体功能、函数参数和返回值等信息, 并注意它们属于系统调用, 还是库函数, 以加速对系统调用概念的理解。

(3) 简单范例 3

编译多个源文件。

(1) 使用 vi/vim 编辑器编写两个文件: message.c, main.c:

vi message.c

vi main.c

(2) 使用 gcc 编译:

gcc -c message.c //输出 message.o 文件, 是一个已编译的目标代码文件

gcc -c main.c //输出 main.o 文件

gcc -o all main.o message.o //执行连接阶段的工作, 然后生成可执行文件 all

(3) 执行可执行文件:

./all

注意: : gcc 对如何将多个源文件编译成一个可执行文件有内置的规则, 所以前面的多个单独步骤可以简化为一个命令:

gcc -o all message.c main.c

4.2.4 gdb 调试工具

gdb 是一个 GNU 调试工具, 可以调试 C 和 C++ 程序, 其主要功能有: 1) 监视程序中变量的值; 2) 设置断点; 3) 单步执行程序。

为了能够使用 gdb 调试程序, 必须在编译时包含调试信息, 即使用 gcc 编译时需要加上 “-g” 选项, 如 gcc -g -o test test.c。

gdb 命令很多, 下面列出一些常用的调试命令:

1、gdb 启动及退出:

(1) 启动gdb命令:

gdb exefilename

其中 exefilename 是可执行文件名, 如果没有指定运行程序, 也可进入 gdb 后再用 file 命令装入文件。

gdb 启动成功后, 提示符为: (gdb), 随后可以输入 gdb 命令对程序进行调试。

(2) 退出 gdb 命令: (gdb)quit

2、断点管理命令:

(1) 设置断点:

break 命令 (可简写为 b) 可以用来在调试的程序中设置断点, 该命令有如下四种

形式:

(gdb)break line-number 使程序在执行给定行之前停止。

(gdb)break function-name 使程序在进入指定的函数之前停止。

(gdb)break line-or-function if condition 如果 condition (条件) 是真, 程序到达指定行或函数时停止。

(gdb) break routine-name 在指定例程的入口处设置断点

如果该程序是由很多原文件构成的, 你可以在各个原文件中设置断点, 而不是在当前的原文件中设置断点, 其方法如下:

(gdb) break filename:line-number

(gdb) break filename:function-name

(2) 从断点处继续执行:

(gdb) continue

(3) 显示当前 gdb 的断点信息:

(gdb) info break

(4) 删除指定的某个断点:

(gdb) delete breakpoint 1

该命令将会删除编号为 1 的断点, 如果不带编号参数, 将删除所有的断点:

(gdb) delete breakpoint

(5) 禁止使用某个断点

(gdb) disable breakpoint 1

该命令将禁止断点 1

(4) 允许使用某个断点

(gdb) enable breakpoint 1

该命令将允许断点 1

(5) 清除原文件中某一代码行上的所有断点

(gdb)clean number

注: number 为原文件的某个代码行的行号

3、显示信息命令:

(1) print 命令:

利用 print 命令可以检查各个变量的值。

(gdb) print p // (p 为变量名或表达式)

(gdb) print 开始表达式@连续内存空间大小

打印内存中某一连续内存空间的值, 主要用于打印数组类型的表达式的值。

(gdb) print 程序中的某一函数调用

如: (gdb) print find_entry(1,0)

(2) whatis 命令: 可以显示某个变量的类型

(gdb) whatis p // (p 为变量)

type = int *

(3) display 命令：设置要显示的表达式

(gdb)display 表达式

当程序运行到断点时，显示该表达式的值。

(gdb)info display: 显示所有要显示表达式的值。

(gdb)undisplay 表达式: 结束已设置的表达式。

(4) awatch 命令：设置要监视的表达式

(gdb)awatch 表达式

当表达式的值变化或被读取时，程序暂停，显示表达式的值。

4、程序运行控制命令：

(1) run: 运行程序

(2) kill: 结束程序的调试运行

(3) cont: 继续执行程序

(4) next: 单步运行，不进入子程序

(5) step: 单步运行，进入子程序

5、文件命令：

(1) file 命令：加载调试文件

file 文件名

(2) list 命令：列出文件内容

list [参数]

参数说明：

参数为空：从上次显示的最后一行或附近开始，显示 10 行

<行号>: 从当前文件的该行开始显示

<文件名> <行号>: 从指定文件的指定行开始显示

<函数名>: 显示指定的函数

<文件名> <函数名>: 指定文件的指定函数

<行号 1><行号 2>: 从行号 1 显示到行号 2

6、堆栈相关命令：

(1) backtrace 命令：可简写为 bt，显示栈中内容。

命令: bt: 显示当前函数调用栈的所有信息。

命令: bt <n>: n 是一个正整数，表示只打印栈顶上 n 层的栈信息。

n 是一个负整数，表示只打印栈底下 n 层的栈信息。

(2) frame 命令：可简写为 f，显示当前栈层的信息：

显示的内容有：栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。

info frame (或 info f): 出更为详细的当前栈层的信息，只不过，大多数都是运行时的内存地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等。

第五章 简单文件系统的实现

5.1 设计目的和内容要求

1. 设计目的

通过具体的文件存储空间的管理、文件的物理结构、目录结构和文件操作的实现，加深对文件系统内部数据结构、功能以及实现过程的理解。

2. 内容要求

(1) 在内存中开辟一个虚拟磁盘空间作为文件存储分区，在其上实现一个简单的基于多级目录的单用户单任务系统中的文件系统。在退出该文件系统的使用时，应将该虚拟文件系统以一个 linux 文件的方式保存到磁盘上，以便下次可以再将它恢复到内存的虚拟磁盘空间中。

(2) 文件存储空间的分配可采用显式链接分配或其他办法。

(3) 空闲磁盘空间的管理可选择位示图或其他办法。如果采用位示图来管理文件存储空间，并采用显式链接分配方式，那么可以将位示图合并到 FAT 中。

(4) 文件目录结构采用多级目录结构。为了简单起见，可以不使用索引结点，其中的每个目录项应包含文件名、物理地址、长度等信息，还可以通过目录项实现对文件的读和写的保护。

(5) 要求提供以下操作命令：

- my_format: 对文件存储器进行格式化，即按照文件系统的结构对虚拟磁盘空间进行布局，并在其上创建根目录以及用于管理文件存储空间等的数据结构。

- my_mkdir: 用于创建子目录。
- my_rmdir: 用于删除子目录。
- my_ls: 用于显示目录中的内容。
- my_cd: 用于更改当前目录。
- my_create: 用于创建文件。
- my_open: 用于打开文件。
- my_close: 用于关闭文件。
- my_write: 用于写文件。
- my_read: 用于读文件。
- my_rm: 用于删除文件。
- my_exitsys: 用于退出文件系统。

3. 学时安排 (12 学时)

4. 开发平台

Linux 环境, gcc, gdb, vim 或 gedit 等。

5. 思考

(1) 我们的数据结构中的文件物理地址信息是使用 C 语言的指针类型、还是整型，为什么

么？

(2) 如果引入磁盘索引结点，上述实现过程需要作哪些修改？

(3) 如果设计的是一个单用户多任务文件系统，则系统需要进行哪些扩充（尤其要考虑读写指针问题）？如果设计的是一个多用户文件系统，则又要进行哪些扩充？