

# 操作系统课程设计指导书

赵伟华 刘真 张梅

2017 年 08 月

计算机学院

## 第六章 linux 常用工具介绍

### 6.1 查看 linux 源码内容工具

学习 linux 的实现原理时,经常需要查看某些数据结构的定义格式、某些内核函数的具体实现过程等,相关工具比较多,这里介绍 ctags 和 source insight 两种。

#### 6.1.1 通过相关网站查询

1. 网址 1: <http://lxr.linux.no/>
2. 网址 2: <http://lxr.free-electrons.com/>

#### 6.1.2 vim + ctags

在 vim 中安装插件 ctags 后,就可以在终端方便地使用 vi 命令查看 linux 源码内容了。

##### 1. 安装 ctags 插件

Ctags 插件的功能是遍历 linux 源码文件,为源码的变量/对象、结构体/类、函数/接口、宏等产生索引文件:tags 文件,以便快速定位这些内容.tags 文件也是 Taglist 和 OmniCppComplete 工作的基础。

有两种方法安装 ctags 插件,一种是源码安装,一种是在 ubuntu 中使用 apt-get 安装。

- (1) ubuntu 中使用 apt-get 安装:

```
sudo apt-get install ctags
```

- (2) 下载源码安装:

- 1) 从 <http://prdownloads.sourceforge.net/ctags/ctags-5.8.tar.gz> 下载 ctags 源码包 ctags-5.8.tar.gz;

或者从 <http://ctags.sourceforge.net/> 下载 ctags 源代码包

- 2) 解压缩生成源代码目录;
- 3) 进入源代码目录,依次执行如下命令安装 ctags:

```
$ ./configure
$ make
$ sudo make install
```

##### 1. 生成索引文件 tags

从终端进入 linux 源码目录,执行:

```
$ sudo ctags -R *
```

“-R”标识递归创建,即包括 linux 源代码根目录(当前目录)下的所有子目录;“\*”表示所有文件。执行该命令后会在当前目录下生成一个“tags”文件,如图所示:

```
zwh@ubuntu:/usr/src/linux-4.4.19$ sudo ctags -R *
[sudo] password for zwh:
zwh@ubuntu:/usr/src/linux-4.4.19$ ls
arch      crypto    include  kernel    net        security  virt
block     Documentation  init     lib        README     sound
certs     drivers   ipc      MAINTAINERS  REPORTING-BUGS  tags
COPYING   firmware  Kbuild   Makefile    samples     tools
CREDITS   fs        Kconfig  mm          scripts     usr
zwh@ubuntu:/usr/src/linux-4.4.19$
```

如果只是在 linux 源代码目录下查询源码信息，则不需要修改 vim 的配置文件；否则如果希望在其他路径下也能查看，则必须使用 `sudo vim /etc/vim/vimrc` 编辑这个文件，方法是在 vimrc 文件中添加如下内容：

```
set tags=/usr/src/linux-4.4.19/tags;
set autochdir
```

## 2. 利用 ctags 文件查看 linux 源码信息

最常用的命令有下面几个：

(1) `$ vi -t tag:`

命令中的“tag”是要查看的变量名、数据结构名、函数名等，如执行：

```
$ vi -t effective_prio
```

则会显示 `effective_prio()` 的实现源码：

```
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    /*
     * If we are RT tasks or we were boosted to RT priority,
     * keep the priority unchanged. Otherwise, update priority
     * to the normal priority:
     */
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```

(2) `Ctrl+]命令`

在 vim 中，将光标移动到要查看的变量名、数据结构名、函数名处，同时按下“`Ctrl+]`”键，则立即跳转到光标所在变量名、数据结构名、函数名的定义处。

(3) `Ctrl+T 命令`

在 vim 中同时按下“`Ctrl+T`”键，则返回查找或跳转到前一次界面处。

(4) `ta 命令`

在 vim 中命令行下使用 `ta` 命令，也能显示变量名、数据结构名、函数名的定义，如：

```
:ta normal_prio
```

则显示 `normal_prio()` 的定义：

```
static inline int normal_prio(struct task_struct *p)
{
    int prio;

    if (task_has_dl_policy(p))
        prio = MAX_DL_PRIO-1;
    else if (task_has_rt_policy(p))
        prio = MAX_RT_PRIO-1 - p->rt_priority;
    else
        prio = __normal_prio(p);
    return prio;
}
```

更多功能可通过命令 `man ctags` 或在 Vim 命令行下运行 `help ctags` 查询。

### 6.1.3 安装使用 Taglist

Taglist 是 vim 的一个插件，提供源码的结构化浏览功能，可将源码中定义的函数、变量、结构体等以树结构显示，层次关系一目了然，便于快速定位查看。

#### 1. 安装 Taglist 插件

- 1) 从 [http://www.vim.org/scripts/script.php?script\\_id=273](http://www.vim.org/scripts/script.php?script_id=273) 下载安装包，也可以从 <http://vim-taglist.sourceforge.net/index.html> 下载。
- 2) 以 root 用户进入 `/etc/vim` 目录,将 Taglist 安装包复制到该目录下并解压，解压后会在当前目录下生成两个子目录：plugin 和 doc，如图所示：

```
root@ubuntu:/etc/vim# unzip taglist_46.zip
Archive:  taglist_46.zip
  inflating: plugin/taglist.vim
  inflating: doc/taglist.txt
root@ubuntu:/etc/vim# ls
doc  plugin  taglist_46.zip  vimrc  vimrc.tiny
```

- 3) 进入 `/etc/vim/doc` 目录，运行 vim，执行“`helptags`”命令。该命令是将 doc 下的帮助文档加入到 Vim 的帮助主题中，这样我们就可以通过在 Vim 中运行“`help taglist.txt`”查看 taglist 帮助。
- 4) 打开 vim 的配置文件 `/etc/vim/vimrc`，加入以下两行内容，则安装工作就完成了：
 

```
let Tlist_Show_One_File=1    //不同时显示多个文件的 tag，只显示一个
let Tlist_Exit_OnlyWindow=1  //taglist 为最后一个窗口时，退出 vim

Tlist_Sort_Type=name         //使 taglist 以 tag 名字进行排序
```

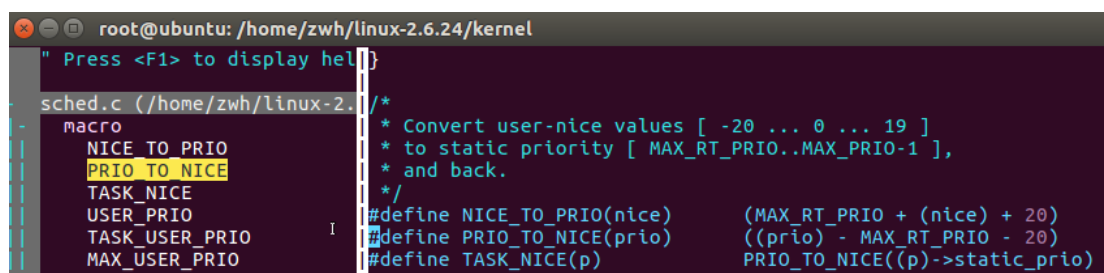
 还有许多其他的设置选项，请参考帮助文档：`help taglist.txt`。

#### 2. 使用 Taglist 插件

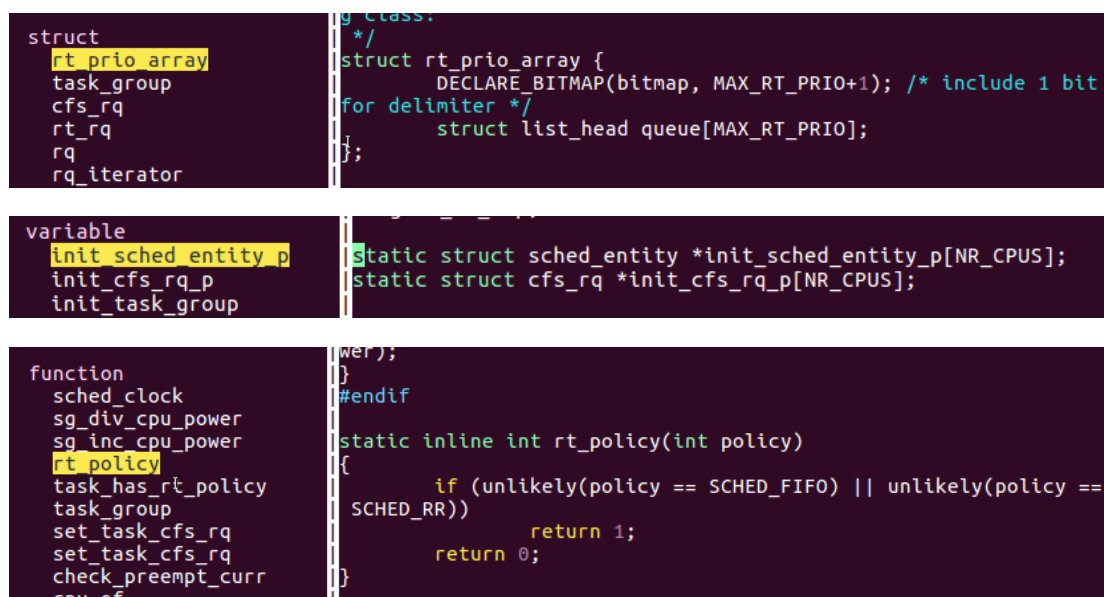
- (1) 在 vim 中打开 taglist 窗口：

在 vim 命令方式下运行以下三个命令之一都可以打开 taglist 窗口，如图所示：

```
:Tlist
:TlistOpen
:TlistToggle
```



图中右边窗口是文件编辑窗口，左边是 taglist 窗口。在 taglist 窗口中分类显示右边文件中所有的 tag（分类依次为宏定义、数据结构、变量、函数），并且每类 tag 都按各 tag 在文件中出现的先后顺序排序，如图所示：



(2) 关闭 taglist 窗口：

在 vim 命令方式下再次运行上述 (1) 中三个命令之一，则关闭 taglist 窗口。

(3) 在 taglist 窗口中常用的快捷键：

- 1) Ctrl+ww 在 taglist 窗口和文件编辑窗口之间切换焦点
- 2) <Ctrl>+] 跳转到光标所在 tag 的定义位置；用鼠标双击此 tag 功能相同
- 3) o 在一个新窗口中显示光标所在 tag 的定义位置；
- 4) <Space> 显示光标下 tag 的原型定义；
- 5) u 更新 taglist 窗口中的 tag
- 6) s 更改排序方式，在按名字排序和按出现顺序排序间切换
- 7) x taglist 窗口放大和缩小，方便查看较长的 tag
- 8) [[ 跳到前一个文件
- 9) ]] 跳到后一个文件
- 9) q 关闭 taglist 窗口
- 10) <F1> 显示帮助

#### 6.1.4 安装使用 Cscope

通过 Cscope 可以很方便地找到某个函数或变量的定义位置、被调用的位置等信息。Cscope 已经是 Vim 的标准特性，默认都有支持，官方网址为 <http://cscope.sourceforge.net/>。

## 1. 安装 Cscope 插件

在 ubuntu 下安装 Cscope:

直接运行命令: `apt-get install cscope`

使用源码安装:

从 <http://cscope.sourceforge.net/> 下载 Cscope 源代码包编译安装，步骤同 Ctags 安装过程。

## 2. 以 root 身份配置 Cscope 插件

1) 下载 [http://cscope.sourceforge.net/cscope\\_maps.vim](http://cscope.sourceforge.net/cscope_maps.vim) 文件到 `/etc/vim/plugin` 目录中;

2) 生成 cscope 数据库文件:

进入 linux 源码根目录，执行下面命令:

```
#cscope -Rbkq
```

该命令将生成三个文件: `cscope.out`, `cscope.in.out`, `cscope.po.out`, 如图所示:

```
root@ubuntu:/home/zwh/linux-2.6.24# cscope -Rbkq
root@ubuntu:/home/zwh/linux-2.6.24# ls
arch      cscope.in.out  fs           kernel      net         security
block     cscope.out    include     lib         README     sound
COPYING   cscope.po.out init         MAINTAINERS REPORTING-BUGS tags
```

其中 `cscope.out` 是基本的符号索引，后两个文件是使用“-q”选项生成的，可以加快 cscope 的索引速度。cscope 命令的参数含义如下:

- R: 在生成索引文件时，搜索子目录树中的代码
- b: 只生成索引文件，不进入 cscope 的界面
- d: 只调出 cscope gui 界面，不跟新 cscope.out
- k: 在生成索引文件时，不搜索/usr/include 目录
- q: 生成 cscope.in.out 和 cscope.po.out 文件，加快 cscope 的索引速度
- i: 如果保存文件列表的文件名不是 cscope.files 时，需要加此选项告诉 cscope 到哪儿去找源文件列表。可以使用“-”，表示由标准输入获得文件列表。
- I dir: 在-I 选项指出的目录中查找头文件
- u: 扫描所有文件，重新生成交叉索引文件
- C: 在搜索时忽略大小写
- P path: 在以相对路径表示的文件前加上 path，这样，你不用切换到你数据库文件所在的目录也可以使用它了。

3) 载入 cscope 数据库文件

使用 cscope 之前，需将前面生成的数据库文件载入到 vim 中。如果之前已经下载过 `cscope_maps.vim` 文件到 `/etc/vim/plugin` 目录下，可忽略这一步。

注意：生成的 cscope.out 和 tags 文件要在打开 vim 所在的文件夹，否则 vim 无法找到相关符号信息。

进入 linux 源码根目录，在 vim 中运行以下命令即可：

```
: cscope add cscope.out
```

到这里，就可以开始使用 Cscope 了。

### 3. 使用 Cscope 插件

Cscope 的查看命令都是在 vim 的命令方式下运行的。

cscope find 命令：

在 vim 中添加 cscope 数据库文件后，就可以调用“cscope find”命令查找源码相关内容，如函数、变量、数据结构的定义及调用情况。vim 支持 8 种 cscope 的查询功能，描述如下：

s: 查找 C 语言符号，即查找函数名、宏、枚举值等出现的地方

g: 查找函数、宏、枚举等定义的位置，类似 ctags 所提供的功能

d: 查找本函数调用的函数

c: 查找调用本函数的函数

t: 查找指定的字符串

e: 查找 egrep 模式，相当于 egrep 功能，但查找速度快多了

f: 查找并打开文件，类似 vim 的 find 功能

i: 查找包含本文件的文件

命令运用举例：

查看函数、宏、枚举等的定义位置：

```
:cscope find g 函数（宏、枚举）名
```

```
或：:cs find g 函数（宏、枚举）名
```

如执行：cscope find g effective\_prio，将显示该函数的实现源码。

查看函数、宏、枚举等的调用位置：

```
:cscope find c 函数（宏、枚举）名
```

```
或：:cs find c 函数（宏、枚举）名
```

如执行：cscope find c effective\_prio，将显示所有调用该函数（宏、枚举）的位置，如图所示：

```
Cscope tag: effective_prio
# line filename / context / line
1 1761 kernel/sched.c <<wake_up_new_task
    p->prio = effective_prio(p);
2 4098 kernel/sched.c <<set_user_nice>>
    p->prio = effective_prio(p);
Type number and <Enter> (empty cancels):
```

其他命令选项的功能请读者自行实践。

#### 3) Cscope 中常用快捷键

在 cscope\_maps.vim 文件中定义了一组快捷键执行前面的命令，具体使用方法请读者自行上机实践。

#### 4) 返回上一次界面：

快捷键“ctrl+t”返回上一次界面。

更多功能可通过命令 `man cscope` 或在 Vim 命令行下运行 `help cscope` 查询。

### 6.1.5 source insight

Source insight 是一款 Windows 下的面向项目开发的代码编辑浏览器，它可以自动同步分析相关源码，为我们的开发及源码分析提供了很大便利。

#### 1. 安装 source insight 软件

到网上搜索该软件，可以安装一个 30 天免费的版本，如图所示：



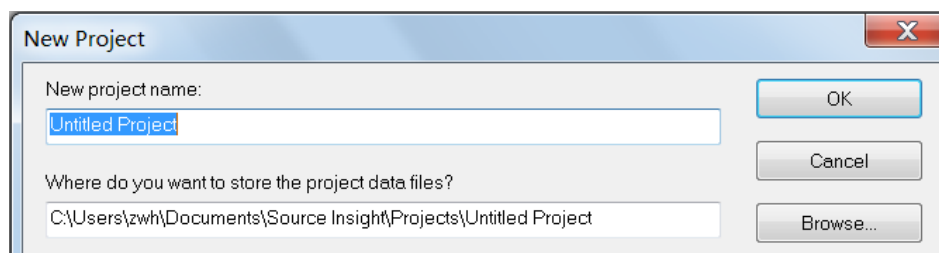
当然，也可以在 linux 下通过 `wine` 安装 source insight，安装过程可以参考如下资源：

<http://blog.csdn.net/gaojinshan/article/details/9272123>

注意：`wine` 安装在当前用户目录下，且是隐藏的，用 `ls -a` 可见，也可通过图形界面直接看。安装好的 `sourceinsight` 程序放在 `wine/drive_c/Program Files(x86)/` 下。可直接通过点击启动。

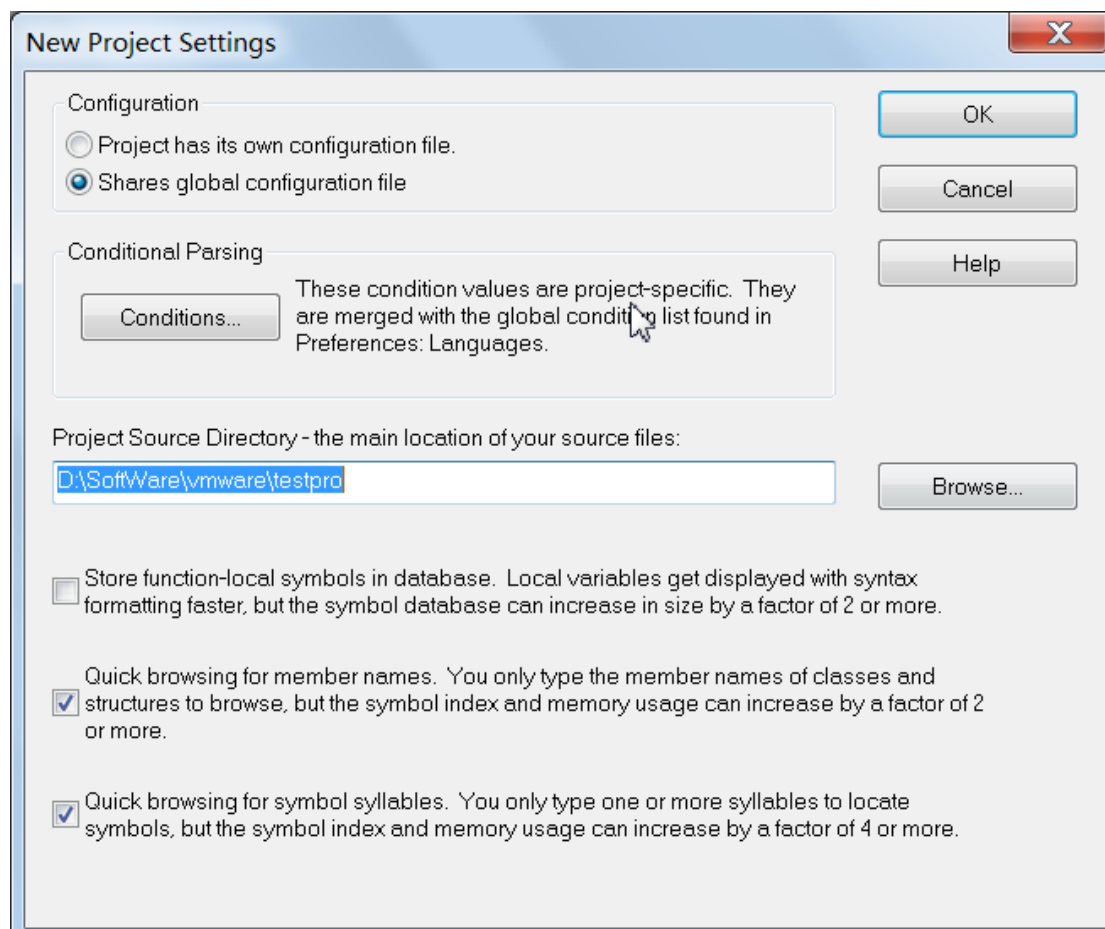
#### 2. 新建工程

- 1) 运行 source insight
- 2) 选择 `project->New project` 新建一个工程，如图所示：

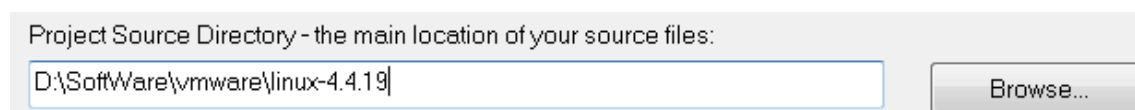


输入新建工程的名字及工程文件的保存路径，单击“OK”关闭该窗口，将同时打开新建工程的设置对话框，如图所示：

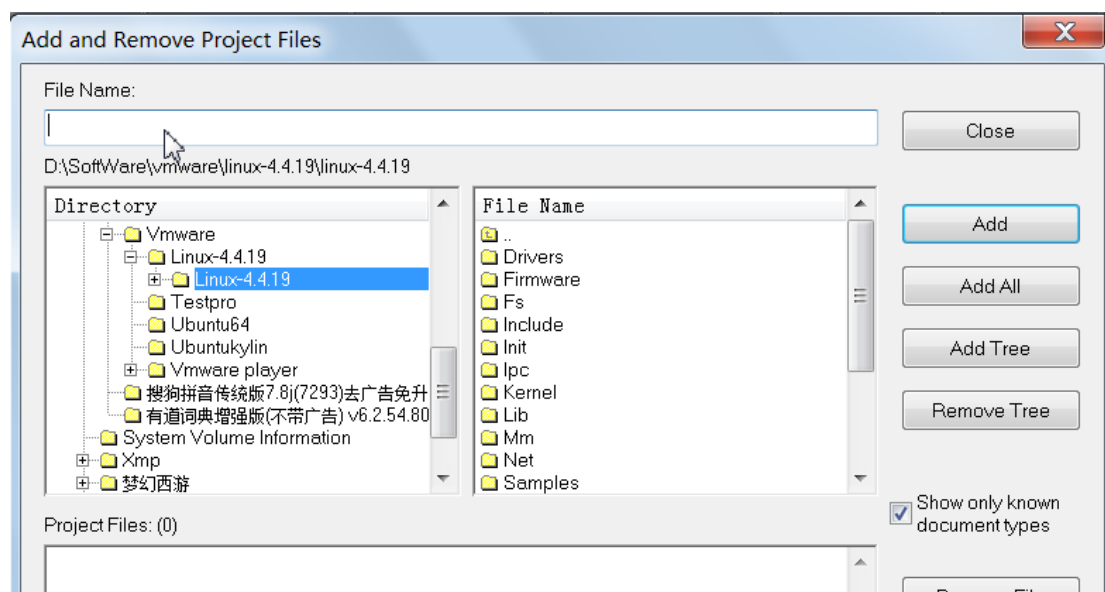




3) 在上面的设置对话框中，首先设置配置文件，可采用默认设置，也可以选择“Project has its own configuration file”；然后选择要添加代码的目录，如本示例设置为 linux 源码所在目录：



最后单击“OK”关闭该对话框，将打开工程文件加载对话框，如图所示：



可直接选择刚才的 linux 源码目录，单击“Add Tree”按钮，将递归加入指定目录中所有文件到工程中。关闭该对话框。

#### 4) “同步”文件或者“重编译”工程

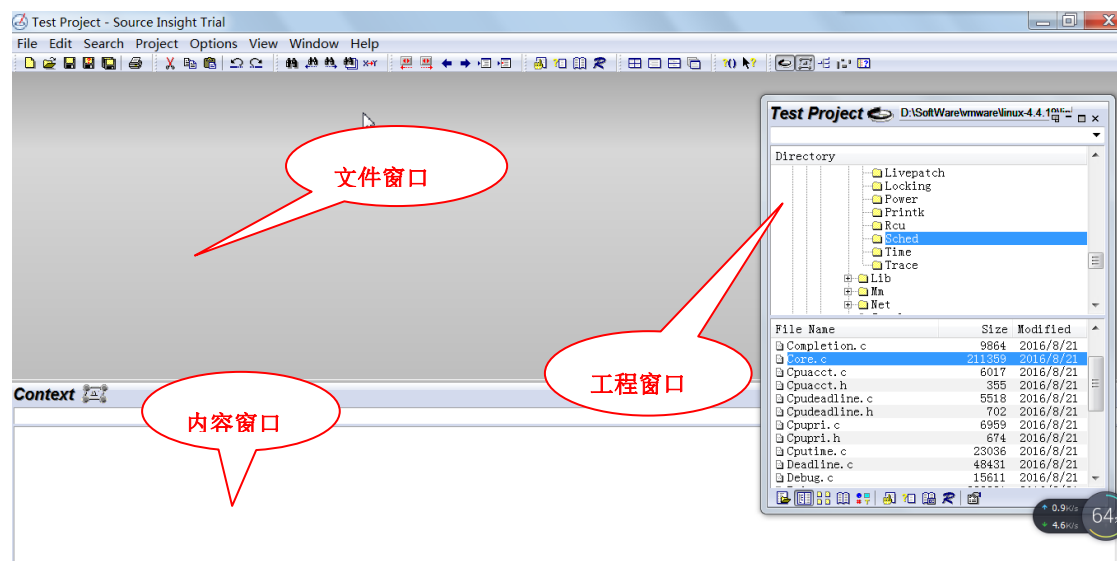
这是很重要的一步,将对代码间调用关系等进行内部初始化。推荐大家进行“重新编译”工程，这样可以建立一个与路径无关的工程，也就是这个工程拿到哪都可以使用，而同步不可以。操作方法是：

同步文件：选择 project->synchronize file...，在显示的对话框中你可以选择：Remove missing files from project 和 Suppress warning messages, 或者再加上 Force all files to be re-parsed，然后单击“OK”，之后工程中的源码就可以进行关联了。

重编译工程：选择 project->rebuild project...，在显示的对话框中，只选择第三项：Re-Create the whole project from scratch, 然后单击“OK”就可以了。

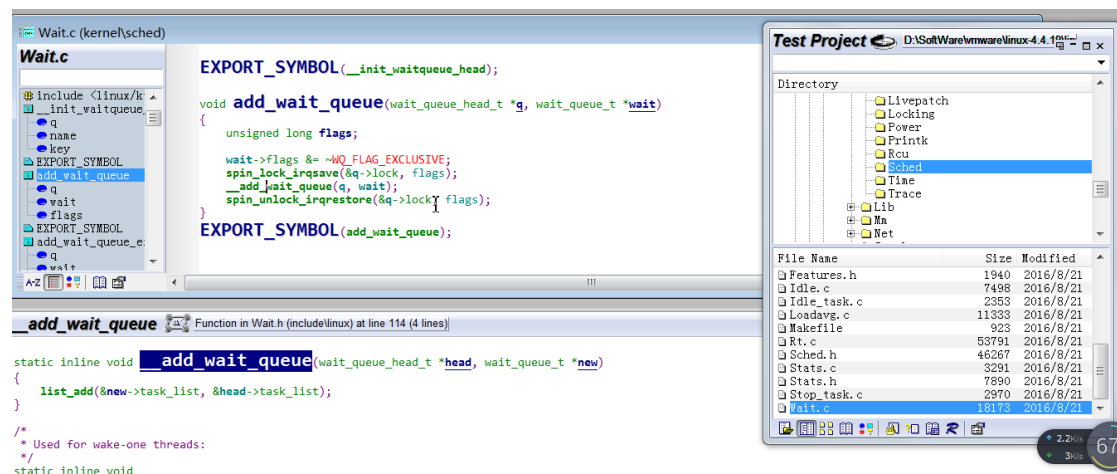
### 3. 使用 sought insight 查看源码

启动 soughtinsight, 选择 project->Open Project...打开工程, 如前面建立的 test 工程, 默认是打开三个窗口, 如图所示:

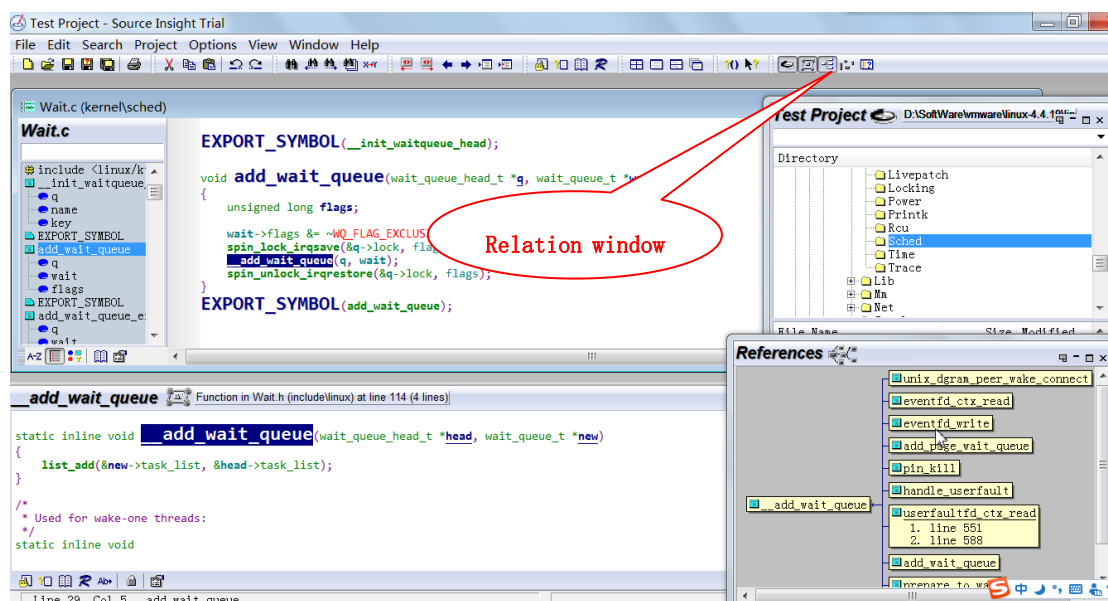


(1) 查看函数及变量:

在“工程窗口”中选择并双击一个要查看的文件, 将在“文件窗口”显示该文件的所有内容: 包含的头文件、定义的变量、函数 (包括函数中的变量定义、所调用的其他函数等), 在其中选择一个函数 (或变量) 并单击, 将在“内容窗口”中显示该函数 (或变量) 的详细定义, 如图所示:



此时如果再单击工具栏上的“relation window”, 将显示所有调用该函数的位置, 如图中右下角窗口所示:

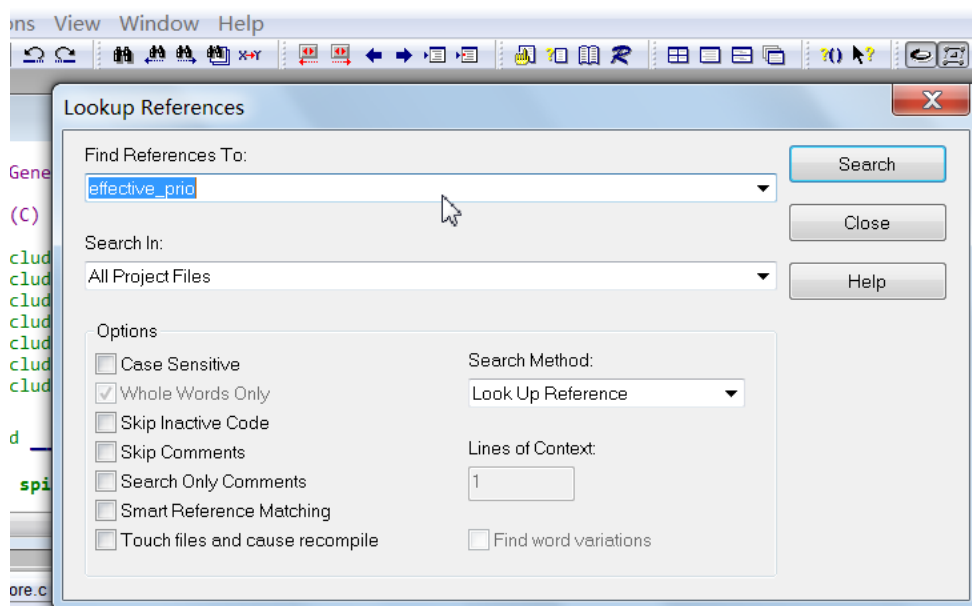


## (2) 搜索字符串

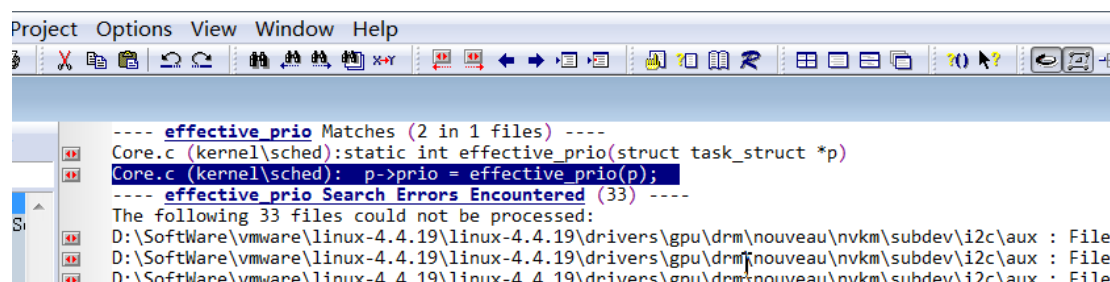
单击工具栏上的“R”按钮，将显示字符串查找窗口“Lookup References”，其中查找选项“Options”按如图设置即可，如图所示：

## (3) 使用鼠标右键功能

在 source insight 中，对任何文件、函数或者变量点击鼠标右键，会显示一个快捷菜单，能更加方便的实现跳转功能，请大家参考其他相关资料学习。



如查找“effective\_prio”，单击“Search”后显示结果如图所示：



点击左边或工具栏中的红色小按钮就可以展开内容，其中工具栏中的按钮下面还有一个红色的“向左箭头”和“向右箭头”，标明一个向前，一个向后依次打开，非常方便。

## 6.2 linux 中的汇编语言

虽然 Linux 内核的绝大部分代码是用 C 语言编写的，但仍然不可避免地在某些关键地方使用了汇编代码，其中主要是在 Linux 的启动部分。Linux 主要使用 AT&T 汇编语言，它与 intel 汇编大同小异，为方便大家阅读源码，下面简单介绍 AT&T 汇编的一些基本知识。

### 6.2.1 AT&T 与 intel 汇编的语法格式比较

#### 1. 寄存器及立即数的前缀

在 AT&T 汇编格式中，寄存器名前要冠以“%”，立即数前要冠以“\$”；而在 Intel 汇编格式中，寄存器名及立即数都不需要加前缀。例如：

AT&T 格式	Intel 格式
pushl %eax	push eax
pushl \$1	push 1

#### 2. 操作数的方向

AT&T 和 Intel 格式中的源操作数和目标操作数的位置正好相反。在 Intel 汇编格式中，目标操作数在源操作数的左边；而在 AT&T 汇编格式中，目标操作数在源操作数的右边。例如：

AT&T 格式	Intel 格式
addl \$1, %eax	add eax, 1

#### 3. 操作数的字长

在 AT&T 汇编格式中，操作数的字长由操作符的最后一个字母决定，后缀‘b’、‘w’、‘l’分别表示操作数为字节（byte，8 比特）、字（word，16 比特）和长字（long，32 比特）；而在 Intel 汇编格式中，操作数的字长是用“byte ptr”和“word ptr”等前缀来表示的。例如：

AT&T 格式	Intel 格式
movb val, %al	mov al, byte ptr val

#### 4. Jump 和 call 指令的前缀

在 AT&T 汇编格式中，绝对转移和调用指令（jump/call）的操作数前要加上‘\*’作为前缀，而在 Intel 格式中则不需要。

#### 5. 内存单元操作数

内存单元操作数的寻址方式是间接寻址。在 AT&T 汇编格式中，基址寄存器用（）括起来，而 Intel 中用[]括起来。例如：

AT&T 格式	Intel 格式
movl -4(%ebp), %eax	mov eax, [ebp - 4]
movl 5(%ebx), %eax	mov eax, [ebx+5]

### 6.2.2 Gcc 嵌入式汇编

Linux 下用汇编语言编写的代码具有两种不同的形式。第一种是整个程序全部用汇编语言编写；第二种是内嵌的汇编代码，指嵌入到 C 语言程序中的汇编代码片段。下面介绍 gcc 嵌入式汇编的相关知识。

#### 1. 嵌入式汇编的一般形式

```
__asm__ __volatile__ ("asm statements"    //指令部
: outputs (optional)  //输出部
: inputs (optional)   //输入部
: modified (optional) //修改部
);
```

下面对语句格式的各部分分别进行说明：

(1) \_\_asm\_\_

表示汇编代码的开始，也可以写成“asm”，两者完全相同。

(2) \_\_volatile\_\_

这是一个可选项，它告诉编译器不要优化该汇编语句，主要用于硬件级程序设计。

(3) 指令部：“asm statements”

是汇编指令部分，可以是一条或多条指令。如果是多条指令，则指令间需要使用“\n\t”进行分隔。

如读 CR0 寄存器的 read\_cr0() 的定义为：

```
static inline unsigned long read_cr0(void)
{
    unsigned long cr0; //汇编内部定义的局部变量
    asm volatile("movq %%cr0,%0" : "=r" (cr0)); // 嵌入式汇编语句
```

```
    return cr0;
}
```

使用嵌入式汇编时，汇编语句中的操作数如何与 C 代码中的变量相结合是个很大的问题。GCC 的解决方法是：需要使用指定寄存器的值时，寄存器名前面应该加上两个‘%’，如示例中的“%%cr0”表示使用 cr0 寄存器；需要使用 C 代码中的变量时，采用加上前缀“%”的数字(如%0, %1)来表示，其中数字是从输出部的第一个约束开始从 0 依次编号。如示例汇编语句中的操作数“%0”对应输出部的“cr0”变量。

#### (4) 输出部: : outputs (optional)

输出部分用于规定输出变量(目标操作数)如何与汇编语句中的操作数相结合的约束条件，可以有多个约束，以逗号分开。每个约束以“=”开头(表示只用输出)，接着用一个字母来表示操作数的类型，最后用“( )”说明变量名。

如上例中的：

: "=r" (cr0)

“=r”表示相应的目标操作数(指令部分的“%0”)可以使用任何一个通用寄存器；并且变量 cr0 存放在这个寄存器中。

实际上，除“=”外，还有其他保留字，相关含义是：

保留字	含义
=	只写/输出变量
+	可读可写变量
&	该输出操作数不能使用输入操作数相同的寄存器

下表中列举了 x86 中最常用的约束字母及含义：

约束字母	含义
m, v, o	表示内存单元
r	任意通用寄存器
q	寄存器 EAX/EBX/ECX/EDX 之一
a, b, c, d	表示寄存器 EAX/EBX/ECX/EDX
S, D	寄存器 ESI 或 EDI
A	与 a+b 相同，使用 EAX 与 EBX 联合，形成一个 64 位寄存器
I	常数 0-31

#### (5) 输入部: : inputs (optional)

输入约束格式与输出约束相似，但是没有“=”号。如果一个输入约束要求使用寄存器，则 GCC 在预处理时就会为之分配一个寄存器，并插入必要的指令将操作数装入该寄存器。如果输入部某个操作数所要求使用的寄存器与前面输出部某个操作数所要求的是同一个寄存器，就把输出部对应操作数的编号(如“0”、“1”等)放在输入部中相应位置，但在输入部中的参数编号需要另外从新编号。程序样例见后面“linux 源码中嵌入式汇编举例”中的 switch\_to() 函数。

#### (6) 修改部: : registers\_modified (optional)

汇编语句在执行过程中可能会修改某些寄存器或者内存单元的值，在这里进行列出。一般是“memory”，表示内存发现变化：:“memory”

此外还有 cc, 表示改变条件代码寄存器(condition code register), 或者某个寄存器名字等。

需要说明的是, 上述各部分中, 指令部是必须的, 输入部、输出部及修改部是可选的, 当输入部存在, 而输出部不存在时, 冒号“:”要保留; 当修改部存在时, 三个冒号都要保留。如宏定义\_\_cli():

```
#define __cli() __asm__ __volatile__("cli": : : "memory")
```

下面是一个简单的嵌入式汇编程序, 其功能是将变量 a 的值赋予变量 b, 请大家自行分析:

```
/* Embedded.c */
int main()
{
    int a = 10, b = 0;
    __asm__ __volatile__("movl %1, %%eax;\n\n\r"
                        "movl %%eax, %0;"
                        : "=r" (b) /* 输出部 */
                        : "r" (a) /* 输入部 */
                        : "%eax"); /* 修改部 */
    printf("Result: %d, %d\n", a, b);
}
```

## 2. Linux 源码中嵌入式汇编举例

(1) 简单应用:

```
#define __save_flags(x) \
    asm volatile ("stc ccr,%w0": "=r" (x)) //将 flags 值压栈
#define __restore_flags(x) \
    asm volatile ("ldc %w0, ccr": : "r" (x)) //恢复 flags 值到 cpu 中
```

(2) 复杂应用:

```
static inline int strcmp(const char *cs, const char *ct)
{
    char res;
    asm ("
        1:      move.b  (%0)+, %2\n      /* get *cs */
        \"      cmp.b   (%1)+, %2\n      /* compare a byte */
        \"      jne     2f\n      /* not equal, break out */
        \"      tst.b   %2\n      /* at end of cs? */
        \"      jne     1b\n      /* no, keep going */
        \"      jra     3f\n      /* strings are equal */
        2:      sub.b   -(%1), %2\n      /* *cs - *ct */
        3:
        : "+a" (cs), "+a" (ct), "=d" (res));
    return res;
}
```



第二个例子是函数 `switch_to()` (`include/asm-x86/system_32.h` 文件中)，它是 linux 上下文切换的核心部分：

```

1 extern struct task_struct * FASTCALL(__switch_to(struct task_struct *prev,
struct task_struct *next));
2 #define switch_to(prev,next,last) do {
3     unsigned long esi,edi;
4     asm volatile("pushfl\n\t"           /* Save flags */      \
5                 "pushl %%ebp\n\t"       \
6                 "movl %%esp,%0\n\t"     /* save ESP */      \
7                 "movl %5,%%esp\n\t"     /* restore ESP */   \
8                 "movl $1f,%1\n\t"       /* save EIP */      \
9                 "pushl %6\n\t"          /* restore EIP */   \
10                "jmp __switch_to\n\t"   \
11                "1:\n\t"                \
12                "popl %%ebp\n\t"        \
13                "popfl"                 \
14                : "=m" (prev->thread.esp), "=m" (prev->thread.eip), \
15                "=a" (last), "=S" (esi), "=D" (edi)                \
16                : "m" (next->thread.esp), "m" (next->thread.eip), \
17                "2" (prev), "d" (next));
18    } while (0)

```

下面对上述代码做简单介绍：

第 1 行：`FASTCALL` 告诉编译程序使用 register 传递参数，而“`asm`”标记则要求采用 stack 传递参数；

第 11 行：参数 1 被用作返回地址；

第 14-15 行：输出参数部分，各参数 `prev->thread.esp`、`prev->thread.eip`、`last`、`esi`、`edi` 对应的编号分别是：`%0`、`%1`、`%2`、`%3`、`%4`；

第 16-17 行：输入参数部分，各参数 `next->thread.esp`、`next->thread.eip`、`prev`、`next` 对应的编号分别是：`%5`、`%6`、`%7`、`%8`，其中 34 行上的参数 `prev` 与输出部中的 2 号参数使用同一寄存器 `eax`，所以在约束位置填写“2”。

## 6.3 特殊的 C 语言用法

### 1. `asm` 及 `FASTCALL`

`asm` 告诉编译程序使用堆栈传递参数，而 `FASTCALL` 通知编译程序使用通用寄存器传递参数。

如获取系统时间函数：

```

asm linkage long sys_gettimeofday(struct timeval __user *tv,
struct timezone __user *tz)

```

实现上下文切换函数：

```

struct task_struct FASTCALL * __switch_to(struct task_struct *prev_p,
struct task_struct *next_p)

```

## 2. UL

UL 常被用在数值常数后，标明该常数是“unsigned long”类型，以保证特定体系结构内的数据不会溢出其数据类型所规定的范围。如：

```
#define SLAB_RECLAIM_ACCOUNT    0x00020000UL /* Objects are reclaimable */
```

## 3. static inline

被关键字 static inline 修饰的函数建议 gcc 在编译时将其代码插入到所有调用它的程序中，从而节省了函数调用的开销。但使用 inline 会增加二进制映像的大小，可能会降低访问 CPU 高速缓存的速度。

## 4. const 和 volatile

const 不一定只代表常数，有时也表示“只读”的意思。如“const int \*x”中，x 是一个指向 const 整数的指针，可以修改该指针，但不能修改这个整数；而在“int const \*x”中，x 是一个指向整数的 const 指针，可以修改该整数，但不能修改指针 x。

关键字 volatile 通知编译程序每次使用被它修饰的变量时都要重新加载其值，而不是存储并访问一个副本。

## 5. 宏 \_\_init

宏 \_\_init 告诉编译程序相关的函数和变量仅用于初始化。编译程序将标有 \_\_init 的所有代码存储到特殊的内存段中，初始化结束后就释放这段内存。如编写模块代码的初始化函数时，可以这样定义：

```
static int __init mymodule_init(void)
```

与之类似，如果某些数据也只在初始化时才用到，则可将其标记为 \_\_initdata。如 ESP 设备驱动程序中：

```
drivers/char/esp.c
```

```
109 static char serial_name[] __initdata = "ESP serial driver";
```

```
110 static char serial_version[] __initdata = "2.2";
```

同样，宏 \_\_exit 和 \_\_exitdata 仅用于退出和关闭例程，一般在注销设备驱动程序或模块时才使用。

## 6. 宏 likely () 和 unlikely ()

现代 CPU 具有精确的启发式分支预测法，它尝试预测下一条到来的命令，以便达到最高的速度。宏 likely 和 unlikely 允许开发者通过编译程序告诉 CPU：某段代码很可能被执行，因而应该预测到；某段代码很可能不被执行，因此不必预测。两个宏的定义如下：

```
include/linux/compiler.h
```

```
# define likely(x)    __builtin_expect(!!(x), 1)
```

```
# define unlikely(x)  __builtin_expect(!!(x), 0)
```

## 7. 宏 IS\_ERR() 和 PTR\_ERR()

这两个宏定义在 `include/linux/err.h` 中。宏 `IS_ERR` 用于判断内核函数的返回值是否是一个有效指针,即可用来判断内核代码执行是否有错误;而宏 `PTR_ERR` 则返回该错误代码。

### 参考文献:

1. 陈莉君, 康华. linux 操作系统原理与应用. 北京: 清华大学出版社, 2012.
2. [美]Claudia Salzberg Rodriguez, Gordon Fischer, Steven Smolski. linux 内核编程. 北京: 机械工业出版社, 2006.

# 实验一 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,读者应理解 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) 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,并返回进程最新的 `nice` 值及优先级 `prio`。建议调用原型为:

```
int mysetnice(pid_t pid, int flag, int nicevalue, void __user * prio, void __user * nice);
```

参数含义:

`pid`: 进程 ID。

`flag`: 若值为 0, 表示读取 `nice` 值; 若值为 1, 表示修改 `nice` 值。

`Prio`、`nice`: 进程当前优先级及 `nice` 值。

返回值: 系统调用成功时返回 0, 失败时返回错误码 `EFAULT`。

(2) 写一个简单的应用程序测试 (1) 中添加的系统调用。

(3) 若程序中调用了 linux 的内核函数, 要求深入阅读相关函数源码。

### 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内核函数，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”来调用系统调用，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. 系统调用号与系统调用表

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；同时每个系统调用需要一个服务例程完成其具体功能。Linux 内核中设置了一张系统调用表，用于关联系统调用号及其相对应的服务例程入口地址，定义在./arch/x86/entry/syscalls/syscall\_64.tbl 文件中（32 位系统是 syscall\_32.tbl），每个系统调用占一表项，比如大家比较熟悉的几个系统调用的调用号是：

表 7-5 系统调用号举例

系统调用号	32 位/64 位/common	系统调用名称	服务例程入口
0	common	read	sys_read
1	common	write	sys_write
2	common	open	sys_open
3	common	close	sys_close
57	common	fork	stub_fork

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，先将系统调用号存入 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。

### 2. 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，其代码实现通常存放在./kernel/sys.c 文件中。服务例程的原型声明则是在/include/linux/syscalls.h 中，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。在新版本的内核中，引入了宏“SYSCALL\_DEFINEN(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

在./kernel/sys.c 文件中的实现格式为:

```
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, 服务例程的实现格式为:

```
SYSCALL_DEFINE0(zwhsyscall)
```

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>

### 3. 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, 通常是 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。

### 4. 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()`, 在较低内核版本中, 定义在./arch/x86/lib/usercopy\_32.c 文件中; 对于内核 4.12, 定义在./include/linux/uaccess.h 文件中:

(1) `copy_to_user()`:

```
static __always_inline unsigned long __must_check
```

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

功能: 将数据块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地址;

`from`: 内核空间的源地址

`n`: 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()`

```
static __always_inline unsigned long __must_check
```

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

功能: 将数据块从用户空间复制到内核空间

参数含义:

to: 内核空间的目的地址;

from: 用户空间的源地址

n: 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 Linux 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, 该调用没有输入参数, 名字叫“zwhsyscall”。采用的内核版本是最新的 4.12, x86 平台、64 位。注意必须以 root 身份才能完成下述操作。

#### 1. 分配系统调用号, 修改系统调用表

查看系统调用表 (./arch/x86/entry/syscalls/syscall\_64.tbl), 如图 7-16 所示:

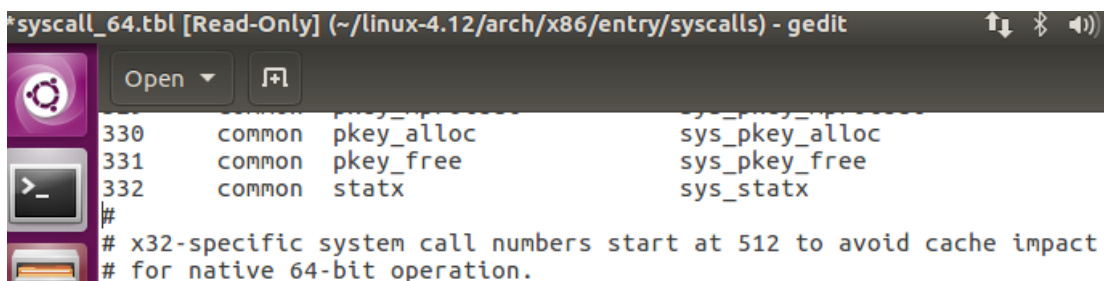


图 7-16 系统调用表部分内容

每个系统调用在表中占一表项, 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>

选择一个未使用的系统调用号进行分配, 比如当前系统使用到 332 号, 则新添加的系统调用可使用 333 号。确定调用号后, 应在系统调用表中关联新调用的调用号与服务例程入口, 即在 syscall\_64.tbl 文件中为新调用添加一条记录, 修改结果如图 7-17 所示:

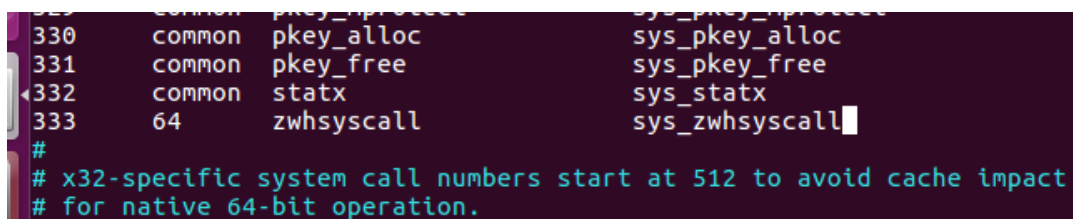


图 7-17 修改系统调用表

#### 2. 申明系统调用服务例程原型

Linux 系统调用服务例程的原型声明在文件 linux-4.12/include/linux/syscalls.h 中, 可在文件末尾添加如图 7-18 所示内容:

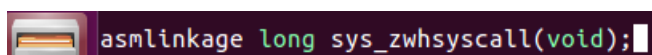


图 7-18 声明系统调用服务例程原型

### 3. 实现系统调用服务例程

下面为新调用 `zwhsyscall` 编写服务例程 `sys_zwhsyscall`，通常添加在 `sys.c` 文件中，其完整路径为：`linux-4.12/kernel/sys.c`，如图 7-19 所示：

```
SYSCALL_DEFINE0(zwhsyscall)
{
    printk("Hello,this is zwh's syscall test!\n");
    return 0;
}
```

图 7-19 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 7.2.4 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  #define _GNU_SOURCE
2  #include <linux/unistd.h>
3  #include <sys/syscall.h>
4  #define __NR_mysyscall 333 /*系统调用号根据实验具体数字而定*/
5  int main() {
6      syscall(__NR_mysyscall); /*或 syscall(333) */
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 7-20 所示：

```
[ 25.002603] Bluetooth: RFCOMM socket layer initialized
[ 25.002609] Bluetooth: RFCOMM ver 1.11
[ 368.541823] Hello,this is zwh's syscall test!
zwh@ubuntu:~$
```

图 7-20 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-4.12.tar.xz。

虚拟机：VMware-player-12.1.1-3770994.exe

虚拟机的建议配置参数：磁盘空间 30GB~40GB 以上，内存 2GB 以上。由于在内核编译过程中会生成较多的临时文件，如果磁盘空间预留太小，会出现磁盘空间不足的错误而导致内核编译失败；内存太小会影响编译速度，一般内核编译时间是 1.5h~3h。

## 2. 下载内核源码

Linux 的内核源代码是完全公开的，有很多网站都提供源码下载，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 7-21 所示：

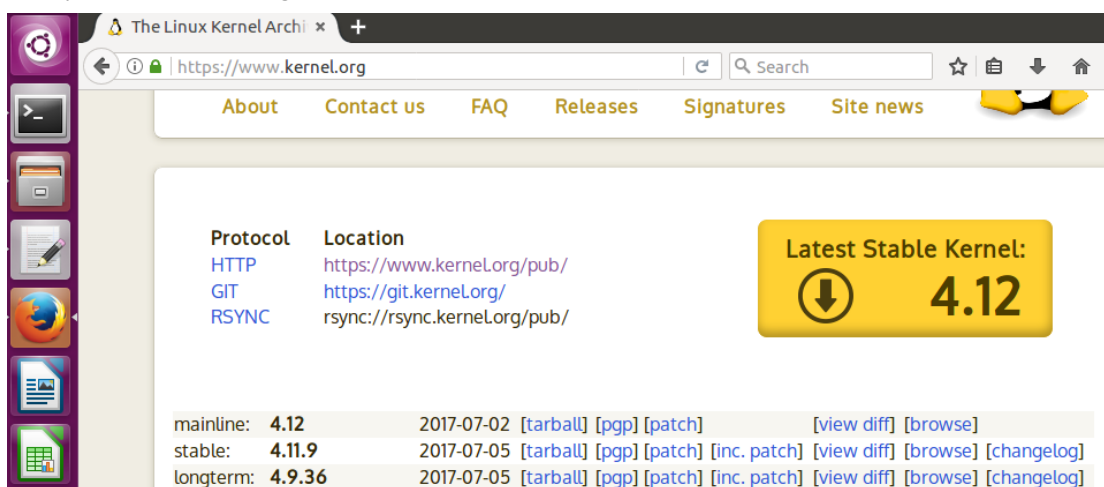


图 7-21 linux 官方网站

## 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/home 或其他比较空闲的目录中，然后进入压缩文件所在子目录，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`

注意：由于编译过程中会生成很多临时文件，所以要确保压缩文件所在子目录有足够的空闲空间，最好能有 15-20GB。笔者在建立虚拟机时预留了 40GB 磁盘空间。

## 4. 清除残留的.config 和.o 文件

当编译出错需要重新编译或不是第一次编译，都需要清除残留的.config 和.o 文件，方法是进入 linux-4.12 子目录，执行以下命令：

```
# 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
```

安装完成后再次执行：`# make mrproper`



## 5. 配置内核

运行命令：`# make menuconfig`

运行该命令过程中，可能会出现如图 7-22 所示错误信息：

```
root@ubuntu:/usr/src/linux-source-4.4.0# make menuconfig
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kconfig/mconf.o
In file included from scripts/kconfig/mconf.c:23:0:
scripts/kconfig/lxdialog/dialog.h:38:20: fatal error: curses.h: No such file or
directory
compilation terminated.
scripts/Makefile.host:108: recipe for target 'scripts/kconfig/mconf.o' failed
make[1]: *** [scripts/kconfig/mconf.o] Error 1
Makefile:541: recipe for target 'menuconfig' failed
make: *** [menuconfig] Error 2
```

图 7-22 缺少套件 ncurses devel 的错误信息

这是因为 Ubuntu 系统中可能缺少一个套件 ncurses devel，安装方法是执行命令：

`# apt-get install libncurses5-dev`

之后再执行“`make menuconfig`”命令将打开如图 7-23 所示配置对话框，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：选择<save>保存配置信息，然后选择<exit>退出对话框。

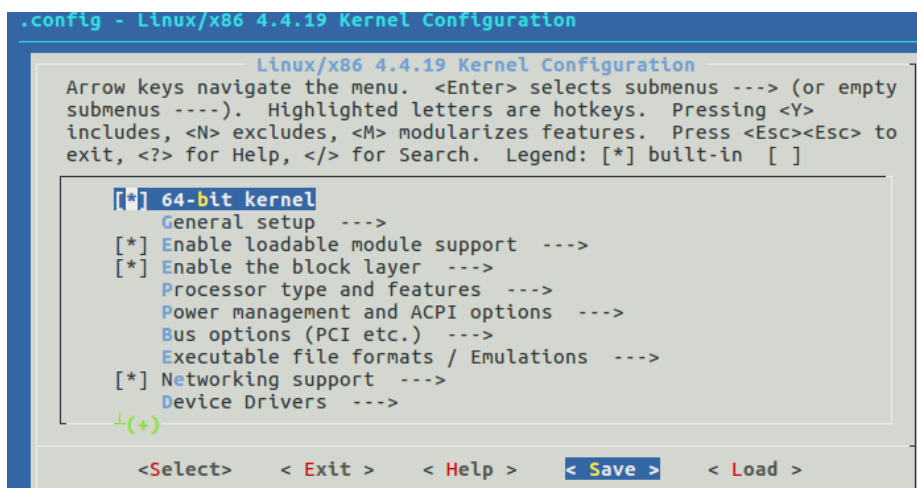


图 7-23 配置内核界面

## 6. 编译内核，生成启动映像文件

内核配置完成后，编译内核，并生成启动映像文件 bzImage（位于 `./arch/x86_64/boot/bzImage`）：

执行命令：`# make`

如果是多核 cpu 可使用 `make -j` 来加快编译速度。可能会出现图 7-24 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 direc
tory
```

图 7-24 编译错误

这是因为没有安装 openssl，openssl 的安装方法是：

执行命令：`# apt-get install libssl-dev`

openssl 安装完成后，再执行 make 命令即可，需要较长时间，笔者 4 核处理器，用了约 20 分钟。

## 7. 编译模块

执行命令：# make modules

第一次编译模块需要很长时间，笔者大概用了两个半小时。

## 8. 安装内核

(1) 安装模块：# make modules\_install

(2) 安装内核：#make install

## 9. 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。

## 10. 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 7-25 所示：

```
zwh@ubuntu:~$ uname -a
Linux ubuntu 4.12.0 #1 SMP Sat Jul 8 20:23:38 PDT 2017 x86_64 x86_64 x86_64 GNU/
Linux
zwh@ubuntu:~$
```

图 7-25 查看内核版本

# 实验二 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) 请根据自身情况，进一步阅读分析程序中用到的相关内核函数的源码实现。

### 3. 学时安排（共 4 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，它不仅可以弥补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。

最后总结一下内核模块程序源码的组成：

头文件：	#include<linux/init.h #include<linux/module.h	必选
许可声明	MODULE_LICENSE(“Dual BSD/GPL”)	必选
加载函数	static int __init hello_init(void)	必选
卸载函数	static void __exit hello_exit(void)	必选
模块参数	module_param(name,type,perm)	可选
模块导出符号	EXPORT_SYMBOL(符号名)	可选
模块作者等信息	MODULE_AUTHOR(“作者名”)	可选

## 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（注意第一个字母“M”需要大写），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：

```
1  obj-m :=hello.o           //生成的模块名称是：hello.ko
2  KDIR :=/lib/modules/$(shell uname -r)/build
3  PWD :=$(shell pwd)       // PWD 是当前目录
4  default:
5      make -C $(KDIR) M=$(PWD) modules // -C 指定内核源码目录，M 指定模块源码目录
6  clean:
7      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。需要注意的是，在第 5 行和第 7 行的“make”之前应该是“tab”键，而不是空格。

## 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：

```
hello1print.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);
```

```
hello2print.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 :=hello1print.o hello2print.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:

```
1  obj-m :=hello2.o           //生成的模块名称是: hello2.ko
2  hello2-objs :=hello1print.o hello2print.o
3  KDIR :=/lib/modules/$(shell uname -r)/build
4  PWD :=$(shell pwd)        // PWD 是当前目录
5  default:
6      make -C $(KDIR) M=$(PWD) modules // -C 指定内核源码目录, M 指定模块源码目录
7  clean:
8      make -C $(KDIR) M=$(PWD) clean
```

注意: 1) 第 1 行和第 2 行中的“hello2”必须名字相同, 表示模块 hello2 的目标文件来源于 hello1print.o 和 hello2print.o。2) 在第 6 行和第 8 行的“make”之前应该是“tab”键, 而不是空格。

### 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行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,char,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 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

#### (1) list\_for\_each()宏和 list\_entry()宏：

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：

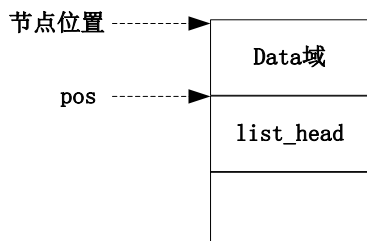


图2-1 list\_for\_each()宏

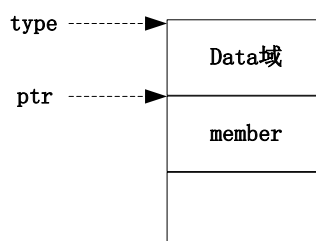


图2-2 list\_entry()宏

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) list\_for\_each\_entry()宏

```
/**
 * list_for_each_entry - iterate over list of given type
 * @pos:           the type * to use as a loop cursor.
 * @head:          the head for your list.
 * @member:        the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member) \
    for (pos = list_entry((head)->next, typeof(*pos), member); \
         prefetch(pos->member.next), &pos->member != (head); \
         pos = list_entry(pos->member.next, typeof(*pos), member))
```

该宏实际上是一个 for 循环, 利用传入的 pos 作为循环变量, 从表头 head 开始, 逐项向后 (next 方向) 移动 pos, 直至又回 head。prefetch() 可以不考虑, 用于预取以提高遍历速度。

## 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()` 访问该链表中的每个进程：

```

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

```

宏 `for_each_process` 定义在 `/include/linux/sched.h` 文件中。

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：

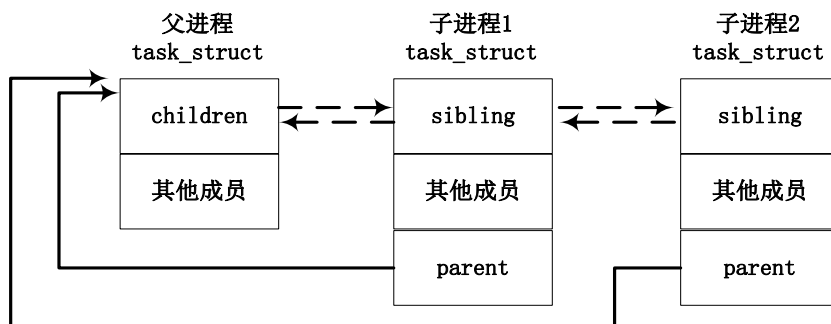


图2-3 进程家族关系

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()` 以及 `list_for_each_entry()` 来实现。对于指定的 `pid`，可以通过函数 `pid_task()` 和 `find_vpid()`（或者 `find_get_pid()`）配合使用找到其相应的 `task_struct` 结构，位于 `linux/kernel/pid.c` 文件中：

```
struct task_struct *result = NULL;
if (pid) {
    struct hlist_node *first;
    first = rcu_dereference_check(pid->tasks[type].first,
    rcu_read_lock_held() ||
    lockdep_tasklist_lock_is_held());
    if (first)
        result = hlist_entry(first, struct task_struct, pids[(type)].node);
    }
return result;
}
struct pid *find_vpid(int nr)
{
    return find_pid_ns(nr, current->nsproxy->pid_ns);
}
struct pid *find_get_pid(pid_t nr)
{
    struct pid *pid;
    pid = get_pid(find_vpid(nr));
    rcu_read_unlock();
    return pid;
}
```

## 实验三 linux 设备驱动程序开发

### 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。

(4) 请根据自身情况, 进一步阅读分析程序中用到的相关内核函数的源码实现。

## 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。

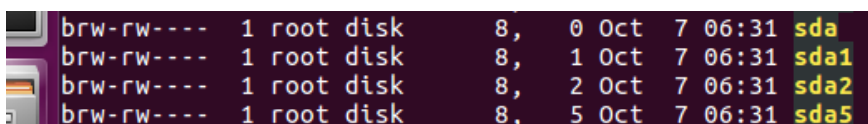
### 3.2.2 设备号的概念

#### 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：



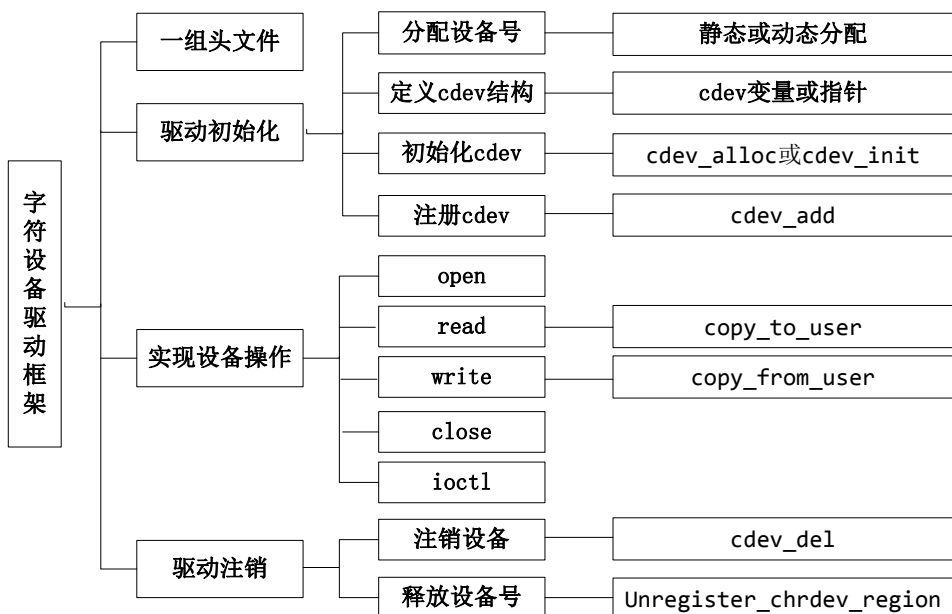


图 3-1 字符设备驱动程序框架图

### 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.h> //模块编程相关函数
#include <linux/init.h> //模块编程相关函数
#include <linux/kernel.h>
#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 *node, 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。

# 实验四 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，加深对不同进程的通信方式的理解。
- （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。
- （5）请根据自身情况，进一步阅读分析相关系统调用的内核源码实现。

### 2. 设计内容

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

#### （2）实现一个模拟的 shell:

编写三个不同的程序 **cmd1.c**, **cmd2.c**, **cmd3.c**，每个程序的功能自定，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) 实现一个管道通信程序：

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，测试管道的默认大小，并且要求利用 Posix 信号量机制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) 利用 linux 的消息队列通信机制实现两个线程间的通信：

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) 利用 linux 的共享内存通信机制实现两个进程间的通信：

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，收到应答消息后，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及同步使用。

## 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 调度给予进程时，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 基本使用:

### 4.2.1 常用命令

pwd, passwd, who, ps, pstree, kill, top, ls, cd, mkdir, rmdir, cp, rm, mv, cat, more, grep 等。  
自行查阅相关资料学习。

### 4.2.2 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。

### 4.2.3 vi 和 vim 编辑器

#### 1. vi 和 vim 简介

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 编辑器是一个全屏幕编辑器，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）。

#### 2. 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，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该模式下完成的。

#### 3. 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。

## 4. 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。[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。

## **(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

## **5. 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。

#### 4.2.4 Linux 环境下 C 编程

##### 1. GCC 编译器

###### (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

###### (4) 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）

### (5) 简单范例 1

1) 按“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下图所示。



```

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

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。

### (7) 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
```

## 2. 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

**6) 允许使用某个断点**

(gdb) enable breakpoint 1

该命令将允许断点 1

**7) 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) :**

显示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。

## 4.3 Linux 管道通信机制

管道是所有 UNIX 及 linux 都提供的一种进程间通信机制, 它是进程之间的一个单向数据流, 一个进程可向管道写入数据, 另一个进程则可以从管道中读取数据, 从而达到进程通信的目的。

### 4.3.1 无名管道

#### 1. 无名管道概念

无名管道通过 **pipe()** 系统调用创建, 它具有如下特点:

- (1) 它只能用于具有亲缘关系的进程 (如父子进程或者兄弟进程) 之间的通信。
- (2) 管道是半双工的, 具有固定的读端和写端。虽然 **pipe()** 系统调用返回了两个文件描述



符，但每个进程在使用一个文件描述符之前仍需先将另一个文件描述符关闭。如果需要双向的数据流，则必须通过两次 `pipe()` 建立起两个管道。

- (3) 管道可以看成是一种特殊的文件，对管道的读写与文件的读写一样使用普通的 `read`、`write` 等函数，但它不是普通的文件，也不属于任何文件系统，而只存在于内存中。

## 2. pipe 系统调用

- (1) 函数原型

```
#include <unistd.h>

int pipe(int filedes[2]);
```

- (2) 参数

`filedes[2]` 参数：是一个输出参数，它返回两个文件描述符，其中 `filedes[0]` 指向管道的读端，`filedes[1]` 指向管道的写端。

- (3) 功能

`pipe` 在内存缓冲区中创建一个管道，并将读写该管道的一对文件描述符保存在 `filedes` 所指的数组中，其中 `filedes[0]` 用于读管道，`filedes[1]` 用于写管道。

- (4) 返回值

成功返回 0；失败返回 -1，并在 `error` 中存入错误码。

- (5) 错误代码

EMFILE：进程使用的文件描述符过多

ENFILE：系统文件表已满

EFAULT：非法参数 `filedes`

## 3. 无名管道的阻塞型读写

管道缓冲区有 4096B 的长度限制，因此，采用阻塞型读写方式时，当管道已经写满时，写进程必须等待，直到读进程取走信息为止。同样，读空的管道时，也可能引起进程阻塞。

当管道大小（管道缓冲区中待读的字节数）为  $p$ ，而用户进程请求读  $n$  个字节时：

若不存在写进程：

- ①  $p=0$ ，则返回 0；
- ②  $0 < p < n$ ，则读得  $p$  个字节，返回  $p$ ，管道缓冲区中还剩 0 个字节；
- ③  $p \geq n$ ，则读得  $n$  个字节，返回  $n$ ，管道缓冲区中还剩  $p-n$  个字节；

若存在写进程，且写进程没因写管道而阻塞时：

- ①  $p=0$ ，读进程阻塞等待数据被写入管道；
- ②  $0 < p < n$ ，则读得  $p$  个字节，返回  $p$ ，管道缓冲区中还剩 0 个字节；
- ③  $p \geq n$ ，则读得  $n$  个字节，返回  $n$ ，管道缓冲区中还剩  $p-n$  个字节；

若存在写进程，且写进程因写管道而阻塞时：

- ①  $0 < p < n$ ，则读管道，当管道缓冲区变空时，阻塞，等待数据被写入。最后，返回实际读得的字节数；
- ②  $p \geq n$ ，则读得  $n$  个字节，返回  $n$ ，管道缓冲区中还剩  $p-n$  个字节。

当管道缓冲区中有  $u$  个字节未用，而用户进程请求写入  $n$  个字节时：

若不存在读进程，则向写管道的进程将发 SIGPIPE 信号，并返回 -EPIPE。

若存在至少一个读进程：

- ①  $u < n \leq 4096$  则写进程等待，直到有  $n-u$  个字节被释放为止，写入  $n$  个字节，返回  $n$ ；
  - ②  $n > 4096$  则写入  $n$  个字节（必要时等待）并返回  $n$ ；
- $u \geq n$  写入  $n$  个字节，返回  $n$ 。

## 4. 有名管道

管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe 或 FIFO）提出后，该限制得到了克服。FIFO 不同于管道之处在于它提供一个路径名与之关联，以 FIFO 的文件形式存在于文件系统中。这样，即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信（能够访问该路径的进程以及 FIFO 的创建进程之间），因此，通过 FIFO 不相关的进程也能交换数据。值得注意的是，FIFO 严格遵循先进先出（first in first out），对管道及 FIFO 的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如 lseek() 等文件定位操作。

有名管道通过 mkfifo 创建：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * pathname, mode_t mode)
```

该函数的第一个参数是一个普通的路径名，也就是创建后 FIFO 的名字。第二个参数与打开普通文件的 open() 函数中的 mode 参数相同。有名管道创建成功，mkfifo() 返回 0；否则返回-1。如果 mkfifo 的第一个参数是一个已经存在的路径名时，错误代码中会返回 EEXIST 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开 FIFO 的函数就可以了。

与普通文件类似，有名管道在使用之前必须先进行 open 操作，具体类似于文件的打开方式：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
```

对有名管道的 open 操作必须遵循下列规则：（1）如果当前打开操作是为读而打开 FIFO 时，若已经有相应进程为写而打开该 FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该 FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。（2）如果当前打开操作是为写而打开 FIFO 时，如果已经有相应进程为读而打开该 FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该 FIFO（当前打开操作设置了阻塞标志）；或者，返回 ENXIO 错误（当前打开操作没有设置阻塞标志）。

一旦打开操作成功，便可通过返回的文件描述符，利用 read、write 系统调用对管道进行读写操作，读写完成应使用 close 系统调用关闭有名管道。

## 4.4 Linux 消息队列通信机制

Linux 系统中，若干个进程可以共享一个消息队列，系统允许其中的一个或多个进程向消息队列写入消息，同时也允许一个或多个进程从消息队列中读取消息，从而完成进程之间的信息交换，这种通信机制被称作消息队列通信机制。消息队列通信机制是客户/服务器模型中常用的进程通信方式：客户向服务器发送请求信息，服务器读取消息并执行相应的请求。

消息可以是命令，也可以是数据。

## 1. 数据结构

### (1) 消息缓冲区 struct msgbuf

消息缓冲区是用来存放消息内容的结构体，而且这个结构体的第一个成员必须是一个大于 0 的长整数，表示对应消息的类型；不过，系统对结构体中其余成员的类型不做任何限制。

include/linux/msg.h 中给出的消息缓冲格式如下：

```
struct msgbuf{
    /* 消息定义的参照格式 */
    long mtype; /* 消息类型（大于 0 的长整数） */
    char mtext[1]; /*消息正文*/
};
```

应用程序员可以重新定义消息缓冲区结构体，其中，成员(mtext)不仅能定义为长度为 1 的字符数组，也可以定义成长度大于 1 的字符数组，或定义成其他的数据类型，Linux 也允许消息正文的长度为 0，即结构体中没有 mtext 域。

虽然，Linux 没限定 mtext 的类型，但却限制了消息的长度，一个消息的最大长度由宏 MSGMAX 决定，根据版本的不同，其取值可能为 8192 或其他值。

### (2) 消息结构 struct msg

消息队列中的每个消息节点中不仅包含了消息内容，还包含了一些其他信息，消息节点由消息结构来描述。include/linux/msg.h 中给出的消息结构格式如下：

```
struct msg {
    struct msg *msg_next; /*消息队列链接指针，指向队列中的下一条消息 */
    long msg_type; /*消息类型，同 struct msgbuf 中的 mtype*/
    char *msg_spot; /* 消息正文的地址, 指向 msgbuf 的消息正文 */
    time_t msg_stime; /* 消息发送的时间 */
    short msg_ts; /* 消息正文的大小 */
};
```

### (3) IPC 对象访问权限 struct ipc\_perm

```
struct ipc_perm{
    key_t key; /* IPC 对象键值 */
    ushort uid; /* owner euid and egid */
    ushort gid;
    ushort cuid; /* creator euid and egid */
    ushort cgid;
    ushort mode; /* 访问权限 */
    ushort seq; /* slot usage sequence number, 即 IPC 对象使用频率信息 */
};
```

其中：

**Key** 是 IPC 对象（例如消息队列，共享存储器等）的键值，每个 IPC 对象都关联着一个唯一的长整型的键值，不同的进程通过相同的键值可访问到同一个 IPC 对象。用户进程在创

建 IPC 对象时可以指定 key 为某个大于 0 的整数，此时，需要用户自己保证该 key 值不与系统中存在的其他 IPC 键值相冲突。更常用的方式是通过函数调用 `ftok(pathname, proj_jd)` 请求系统为用户进程生成一个键值，其中的 `pathname` 是一个实际存在的文件的路径名，而且用户进程具有对该文件的访问权限，`proj_jd` 是一个整数，但 `ftok` 只会用到其低 8 位的值（该值不能为 0），只要路径名访问到的是同一个文件，而且 `proj_jd` 的低 8 位的值相同，则 `ftok()` 调用便将产生相同的键值；如果使用不同的文件路径名和 `proj_jd`，虽然系统不能保证、但通常生成的键值是不同的。

**Mode** 中给出了该 IPC 对象的访问权限，由 9 个二进制位表示所有者、同组用户、其他组用户的访问权限。它可以是下列权限的组合：

访问权限	八进制整数
拥有者可读	0400
拥有者可写	0200
同组用户可读	0040
同组用户可写	0020
其他用户可读	0004
其他用户可写	0002

由于系统规定任何 IPC 结构都不存在可执行权限，因此一个 IPC 对象的权限最大值为 0666（八进制）。

#### （4）消息队列结构体 `struct msqid_ds`

系统中每个消息队列由一个 `struct msqid_ds` 类型的变量来描述，`struct msqid_ds` 的格式如下：

```
struct msqid_ds {
    struct ipc_perm msg_perm; /* 消息队列访问权限*/
    struct msg *msg_first; /* 队列上第一条消息，即链表头*/
    struct msg *msg_last; /* 队列中的最后一条消息，即链表尾 */
    time_t msg_stime; /* 发送给队列的最后一条消息的时间 */
    time_t msg_rtime; /* 从消息队列接收到最后一条消息的时间 */
    time_t msg_ctime; /* 最后修改队列的时间*/
    ...
    ushort msg_cbytes; /*队列上所有消息总的字节数 */
    ushort msg_qnum; /*当前队列上消息的个数 */
    ushort msg_qbytes; /* 队列允许的最大的字节数 */
    ushort msg_lspid; /* 发送最后一条消息的进程的 pid */
    ushort msg_lrpid; /* 接收最后一条消息的进程的 pid */
};
```

Linux 还通过宏 `MSGMNB` 限定了一个消息队列的最大长度（队列中所有消息总的字节数）。

## 2. 消息队列相关的系统调用

Linux 提供了一组消息队列相关的系统调用来方便用户进行消息通信。

## (1) msgget 系统调用

### ①函数原型:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

### ②参数:

key      key 为 0(IPC\_PRIVATE), 则创建一个新的消息队列; 否则, key 为一个大于 0 的长整数, 它对应于消息队列的键值, 通常是通过 ftok() 函数生成的。

msgflg 对消息队列的访问权限和控制命令的组合。其中访问权限见“IPC 对象访问权限 struct ipc\_perm”部分的说明。而控制命令 IPC\_CREAT 表示, 如果 key 对应的消息队列不存在, 则创建它; 而 IPC\_EXCL 必须与 IPC\_CREAT 一起使用, 它表示: 如果 key 对应的消息队列不存在, 则创建一个新的队列, 否则返回-1。

③功能: 如果 IPC\_CREAT 单独使用, semget() 为一个新创建的消息队列返回标识数, 或者返回具有相同键值的已存在消息队列标识数。如果 IPC\_EXCL 与 IPC\_CREAT 一起使用, 要么创建一个新的队列并返回它的标识数, 如果队列已存在, 则返回-1。

④返回值: 成功, 返回消息队列的标识数; 出错, 返回-1, 同时将错误代码存放在 error 中。对于新创建的消息队列, 其 msgid\_ds 结构成员变量的初值设置如下:

msg\_qnum、msg\_lspid、msg\_lrpid 设置为 0;

msg\_stime、msg\_rtime 设置为 0;

msg\_ctime 设置为当前时间;

msg\_qbytes 设成系统的限制值, 即宏 MSGMNB;

msgflg 的读写权限写入 msg\_perm.mode 中;

msg\_perm 结构的 uid 和 cuid 成员被设置成当前进程的有效用 ID, id 和 cuid 成员被设置成当前进程的有效组 ID。

⑤错误代码: EACCES: 指定的消息队列已存在, 但调用进程没有权限访问它

EEXIST: key 指定的消息队列已存在, 而 msgflg 中同时指定 IPC\_CREAT 和 IPC\_EXCL 标志

ENOENT: key 指定的消息队列不存在同时 msgflg 中没有指定 IPC\_CREAT 标志

ENOMEM: 需要建立消息队列, 但内存不足

ENOSPC: 需要建立消息队列, 但已达到系统的限制

## (2) msgsnd 系统调用

### ①函数原型:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

### ②参数:

msqid 消息队列的标识数  
 msgp 存放欲发送消息内容的消息缓冲区指针  
 msgsz 消息正文(而非整个消息结构)的长度  
 msgflg: 0 ——消息队列满时, msgsnd 将会阻塞  
 IPC\_NOWAIT ——消息队列满时, msgsnd 立即返回-1  
 MSG\_NOERROR——消息正文长度超过 msgsz 字节时, 不报错, 而是直接截去其中多余的部分, 并只将前面的 msgsz 字节发送出去

③功能: 在标识数为 msqid 的消息队列中添加一个消息, 即向标识数为 msqid 的消息队列发送一个消息。

④返回值: 消息发送成功, 返回 0; 否则返回-1, 同时 error 中存有错误代码

⑤错误代码: EAGAIN ——参数 msgflg 设为 IPC\_NOWAIT, 而消息队列已满

EACCESS——无权限写入消息队列

EFAULT ——参数 msgp 指向的地址无法访问

EIDRM ——标识符为 msqid 的消息队列已被删除

EINTR ——队列已满而处于阻塞的情况下, 被信号唤醒

EINVAL ——无效的参数 msqid、或消息类型 type 小于等于 0、或 msgsz 为负数或超过系统限制值 MSGMAX

ENOMEM ——系统无足够内存空间存放 msgbuf 消息的副本

### (3) msgrcv 系统调用

①函数原型:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);
```

②参数:

msqid 消息队列的标识数  
 msgp 存放欲接收消息内容的消息缓冲区指针  
 msgsz 消息正文(而非整个消息结构)的长度  
 msg-typ 0 ——接收消息队列中的第一个消息  
 >0 ——接收第一个类型为 msgtyp 的消息  
 <0 ——接收第一个类型小于等于 msgtyp 的绝对值的消息  
 msgflg 0 ——没有可以接收的消息时, msgrcv 阻塞  
 IPC\_NOWAIT ——没有可以接收的消息时, 立即返回-1  
 MSG\_EXCEPT ——返回第一个类型不为 msgtyp 的消息  
 MSG\_NOERROR——消息正文长度超过 msgsz 字节时, 将直接截去其中多余的部分

分

③功能: 如果传递给参数 msgflg 的值为 IPC\_NOWAIT, 并且没有可取的消息, 那么

给调用进程返回 ENOMSG 错误码，否则，调用进程阻塞，直到一条满足要求的消息到达消息队列。如果进程正在等待消息，而相应的消息队列被删除，则返回 EIDRM。如果当进程正在等待消息时，捕获到了一个信号，则返回 EINTR

④返回值： 接收成功，返回实际接收到的消息正文的字节数；否则返回-1，同时 error 中存有错误代码

⑤错误代码：E2BIG ——消息长度超过 msgsz，且 MSG\_NOERROR 标志没被使用

EACCESS——无权限读取消息队列

EFAULT ——参数 msgp 指向的地址无法访问

EIDRM ——标识符为 msqid 的消息队列已被删除

EINTR ——等待消息的情况下，被信号唤醒

EINVAL ——无效的参数 msqid、或 msgsz 为负数

ENOMSG ——参数 msgflg 设为 IPC\_NOWAIT，但无满足要求的消息可接收

#### (4) msgctl 系统调用

①函数原型：

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

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

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

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

②参数：

msqid 消息队列的标识数

cmd IPC\_STAT ——将对应消息队列结构体的值复制到一份到 buf 所指的结构体中，调用者必须有读消息队列的权限

IPC\_SET —— 将 buf 所指结构体中的部分信息：

msg\_perm.uid, msg\_perm.gid, mst\_perm.mode, msg\_qbytes 写到消息队列结构体中，并且更新消息队列结构体 msg\_ctime 成员的值。调用者必须有相应的权限。

IPC\_RMD —— 删除消息队列，并唤醒该消息队列上等待读或等待写的进程。调用者必须有相应的权限。

③功能： 获取或设置消息队列的属性信息，或者删除消息队列

④返回值： 成功，返回 0；否则返回-1，同时 error 中存有错误代码

⑤错误代码：EACCESS—— cmd 为 IPC\_STAT，但调用进程无读消息队列的权限

EFAULT —— 参数 cmd 为 IPC\_STAT 或 IPC\_SET ，但 buf 指向的地址无法访问

EIDRM ——标识符为 msqid 的消息队列已被删除

EINTR ——等待消息的情况下，被信号唤醒

EINVAL ——cmd 或 msqid 为无效的参数

EPERM —— 参数 cmd 为 IPC\_RMD 或 IPC\_SET ，但调用进程无足够的权限

## 4.5 linux 共享内存通信

共享内存是 Linux 支持的三种进程间通信机制（IPC）中的一种。它实际上是一段特殊的内存区域。这一段内存区域可以被两个或者两个以上的进程映射到自身的地址空间中。一个进程写入共享内存中的信息，可以被其它使用这些共享内存的进程，通过一个简单点的内存读操作（memory read operation）读出，从而实现了进程间的通信。共享一个或多个进程通过同时出现在它们的虚拟地址空间的内存进程通信。这块虚拟内存的页面在每一个共享进程的页表中都有页表项目引用。但是不需要在所有进程的虚拟内存都有相同的地址。

### 1. 共享内存的系统调用

任何 Linux 进程创建时，都有很大的虚拟地址空间，这块虚拟地址空间只有一部分放着代码、数据、堆和堆栈，剩余的那些部分在初始化时是空闲的。一块共享内存一旦被连接（attach），即会被映射入空闲的虚拟地址空间。随后，进程即可像对待普通的内存区域那样读、写共享内存。共享内存一共有四个系统调用，它们分别是：

- `shmget()` 创建一块共享内存。
- `shmat()` 将一块已存在共享内存映射（map）到一个进程的地址空间。
- `shmdt()` 取消一个进程的地址空间中一块共享内存块的映射（unmap）。
- `shmctl()` 是管理共享内存的函数，用于执行对共享内存的各种控制命令。

#### (1) `shmget()`: 分配一块共享内存

- 函数声明：

```
#include<sys/ipc.h>
#include<sys/shm.h>

int shmget(key_t, int size, int shmflg);
```

- 输入参数：

key:标识共享内存的键值，在调用前应赋初值

Size:所需的共享内存的最小尺寸（以字节为单位）

shmflg:将分配的共享内存属性标志

- 返回值：

若成功，则返回共享内存的标识符；否则返回-1，错误原因存在于 `errno` 中。

`errno=EINVAL`: 参数 `size` 小于 `SHMMIN` 或者大于 `SHMMAX`。

`EEXIST`: 欲建立 `key` 所指的共享内存，但已经存在。

`EIDRM`: 参数 `key` 所指的共享内存已经删除。

`ENOSPC`: 已经超过系统允许建立的共享内存的最大值（`SHMALL`）。

`ENOENT`: 参数 `key` 所指的共享内存不存在，参数 `shmflg` 也未设 `IPC_CREAT` 位。

`EACCESS`: 没有权限。

`ENOMEM`: 核心内存不足。

- 说明：

参数 `key` 的取值可以是一块已经存在的共享内存的键值、0、或 `IPC_PRIVATE`。如果 `key` 的取值为 `IPC_PRIVATE`，则函数 `shmget()` 将创建一块新的共享内存；如果 `key` 的取值为 0，



而参数 `shmflg` 中设置了 `IPC_CREATE` 这标志，则同样将创建一块新的共享内存。在 IPC 的通信模式下，不管是使用消息队列或者是共享内存，甚至是信号量，每个 IPC 的对象都有唯一的名字，它被称做“键”（key）。

参数 `size` 是要建立的共享内存的长度。所有的内存分配都是以页为单位的。

参数 `shmflg` 主要和一些标志有关。其中有效的标志包括 `IPC_CREAT` 和 `IPC_EXCL`，它们的功能与 `open(2)` 的 `O_CREAT` 和 `O_EXCL` 相当。

在 Linux 内核中，每一个新创建的共享内存都由一个 `shmid_ds` 的数据结构表示。如果函数 `shmget()` 成功创建了一块共享内存，则返回一个可以用于引用该共享内存的 `shmid_ds` 数据结构的标识符。

每一个 IPC 对象，系统共用一个 `struct ipc_perm` 的数据结构来存放权限信息，以确定一个 ipc 操作是否可以访问该 IPC 对象。

## (2) `shmat()`: 连接（attach）一块共享内存

- 函数声明：

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

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

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- 输入参数：

`shmid`: 欲连接（attach）的共享内存的标识符。

`shmaddr`: 欲连接（attach）的地址。

`shmflg`: 一个标识符。

- 返回值：

若成功则返回已连接好的地址，否则返回 -1，错误原因存在于 `errno`。

`errno=EINVAL`: 参数 `shmid` 无效；参数 `shmaddr` 并非页对齐（page aligned），而参数 `shmflg` 中并未设置 `SHM_RND` 这个标志位。

`ENOMEM`: 分配标识符或页表时，系统内存不足。

`EACCESS`: 调用进程没有权限以指定的方式连接该共享内存。

- 说明：

函数 `shmat()` 将以参数 `shmid` 为标识符的共享内存连接（attach）到调用进程的数据段，连接的地址由 `shmaddr` 指定。

参数 `shmflg` 除了可以设置 `SHM_RND` 标志位外，还可以设置 `SHM_RDONLY` 标志位。

## (3) `shmdt()`: 断开（detach）一块共享内存的连接

- 函数声明：

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

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

```
int shmdt (const void *shmaddr);
```

- 输入参数：

`shmaddr`: 欲断开连接（detach）的共享内存的虚拟地址。

- 返回值:

若成功则返回已连接好的地址, 否则返回-1, 错误原因存在于 `errno` 中。

`errno=EINVAL`: 参数 `shmaddr` 无效或参数 `shmaddr` 地址并非共享内存地址。

说明:

函数 `shmdt` 从调用进程的数据段, 断开地址在 `shmaddr` 的共享内存。要分离的共享内存必须是当前连接到调用进程地址空间的共享内存, 也就是说 `shmaddr` 必须等于连接某个共享内存时调用 `shmat` 时的返回值。

#### (4) `shmctl()`: 对一块共享内存的控制操作

- 函数声明:

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

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

```
int shmctl (int shmid, int cmd, struct shmid_ds *buff);
```

- 输入参数:

`shmid`: 为欲处理的共享内存的标识符。

`cmd`: 为欲进行的操作。

`buf`: 缓存。

- 返回值:

若成功则返回共享内存的标识符, 否则返回-1, 错误原因存在于 `errno` 中。

`errno=EINVAL`: 参数 `shmid` 是个无效的标识符或 `cmd` 为无效命令。

`EFAULT`: 参数 `cmd` 为 `IPC_SET` 或 `IPC_STAT`, 但是参数 `buf` 却指向无效的地址。

`EIDRM`: 参数 `shmid` 所指的共享内存已经删除。

`EPERM`: 参数 `cmd` 为 `IPC_SET` 或 `IPC_RMID`, 但是调用进程并不是创建者, 所有者或超级用户, 并且调用进程没有授予这些组或域的权限。

`EACCESS`: 参数 `cmd` 为 `IPC_STAT`, 但是没有权限读写该共享内存。

- 说明

该系统调用允许用户得到一块共享内存的相关信息, 设置一块共享内存的所有者, 组以及读写权限, 或者销毁一块共享内存。以 `shmid` 为标识符的共享内存的相关信息将被放在一个 `shmid_ds` 的数据结构体中返回。

## 实验五 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？

(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？

## 5.2 预备知识

### 5.2.1 FAT 文件系统介绍

#### 1. 概述

FAT 文件系统是微软公司在其早期的操作系统 MS-DOS 及 Windows9x 中采用的文件系统，它被设计用来管理小容量的磁盘空间。FAT 文件系统是以他的文件组织方式——文件分配表（file allocation table, FAT）命名的，文件分配表的每个表项中存放某文件的下一个盘块号，而该文件的起始盘块号则保存在它的文件控制块 FCB 中。在文件分配表中，一般用 FFFF 来标识文件的结束；用 0000 来标识某个逻辑块未被分配，即是空闲块。为了提高文件系统的可靠性，在逻辑磁盘上通常设置两张文件分配表，它们互为备份。此外，文件分配表必须存放在逻辑磁盘上的固定位置，而根目录区通常位于 FAT2 之后，以便操作系统在启动时能够定位所需的文件，其磁盘布局如图 3-1 所示：

引导块	FAT1	FAT2	根目录区	数据区
-----	------	------	------	-----

图 5-1 FAT 文件系统磁盘布局

上述磁盘布局中，引导块中主要存放了用于描述分区的各种信息，包括逻辑块的大小、文件分配表的大小及位置、根目录的大小及位置等。除此之外，用于加载操作系统内核的引导程序也存储在引导块中。

FAT 文件系统家族又分为 FAT12、FAT16、FAT32 三种类型，这里的数字表示文件分配表中每个表项（即簇号）所占的位数，即 FAT12 中每个表项占 1.5 个字节（12 位），FAT16 中每个表项占 2 个字节（16 位），FAT32 中每个表项占 4 个字节（32 位）。由于 FAT 文件系统是以簇为单位为文件分配磁盘空间的（一个簇是一组连续的扇区，通常包含  $2^n$  个扇区），因此，FAT32 比 FAT12 和 FAT16 支持更多的簇数、更小的簇大小和更大的磁盘容量，从而大大提高磁盘空间的利用率。通常，FAT12 适用于小容量磁盘，如软盘；FAT16 是 MS-DOS 的文件系统；FAT32 是 Windows9x 中的主要文件系统，开始支持大容量磁盘。

## 2. 文件控制块 FCB

为了正确、方便地操作文件，必须设置相应的数据结构用于存放文件的描述和控制信息，常用的数据结构有文件控制块（简称 FCB）和索引节点（简称 i 节点）。在 FAT 文件系统中使用文件控制块。文件与文件控制块一一对应，而文件控制块的有序集合就称为文件目录，即一个文件控制块就是一个文件目录项。

虽然不同文件系统的文件控制块的内容和格式不完全相同，但通常都包括以下三类信息：基本信息、存取控制信息和使用信息。

（1）基本信息。包括文件名、用户名、文件类型、文件的物理地址、文件长度、文件的逻辑结构和物理结构等。

（2）存取控制信息。一般分别给出文件主、伙伴用户、一般用户的存取权限。

（3）使用信息。包括文件的建立日期及时间、上次存取文件的日期及时间、当前的使用信息等。

以 MS-DOS（使用 FAT16 文件系统）为例，它的每个文件控制块包括 32 个字节，其字节分配情况如图 3-2 所示：

字节	8B	3B	1B	10B	2B	2B	2B	4B
	文件名	扩展名	属性	保留	时间	日期	首块号	大小

图 5-2 MS-DOS 的文件控制块

其中属性字段占一个字节，它的每一位用来表示该文件是否具有某种属性，如果某一位

的值为 1，则表示该文件具有该属性。各位所表示的属性如表 3-1 所示：

表 5-1 文件属性对照表

位	7	6	5	4	3	2	1	0
属性	保留	保留	存档	子目录	卷标	系统文件	隐藏	只读

### 3. 根目录区

FAT12、FAT16 的根目录区是固定区域、固定大小的，位于第二个 FAT 之后，如图 3-1 所示，且占据若干连续扇区，其中 FAT12 占 14 个扇区，一共 224 个根目录项；而 FAT16 占 32 个扇区，最多保存 512 个目录项，作为系统区的一部分。FAT32 的根目录是作为文件处理的，采用与子目录文件相同的管理方式，其位置不是固定的，不过一般情况也是位于第二个 FAT 之后的，其大小可视需要增加，因此根目录下的文件数目不再受最多 512 个的限制。

### 5.2.2 几个 C 语言库函数介绍

由于我们的文件系统是建立在内存的虚拟磁盘上的，在退出文件系统的时候必须以一个文件的形式保存到磁盘上；而在启动文件系统的时候必须从磁盘上将该文件读入到内存的虚拟磁盘中。下面介绍几个可能会用到的 C 库函数，在使用这些库函数之前必须包含头文件“stdio.h”。

#### 1. 打开文件函数 fopen()

(1) 格式：FILE \*fopen(const char \*filename, const char \*mode)

(2) 功能：按照指定打开方式打开指定文件。

(3) 输入参数说明：

filename：待打开的文件名，如果不存在就创建该文件。

mode：文件打开方式，常用的有：

- “r”：为读而打开文本文件（不存在则出错）。
- “w”：为写而打开文本文件（若不存在则创建该文件；反之，则从文件起始位置写，原内容将被覆盖）。
- “a”：为在文件末尾添加数据而打开文本文件。（若不存在则创建该文件；反之，在原文件末尾追加）。
- “r+”：为读和写而打开文本文件。（读时，从头开始；在写数据时，新数据只覆盖所占的空间，其后不变）。
- “w+”：首先建立一个新文件，进行写操作，随后可以从头开始读。（若文件存在，原内容将全部消失）。
- “a+”：功能与“a”相同；只是在文件末尾添加新的数据后，可以从头开始读。

另外，上述模式字符串中都可以加一个“b”字符，如 rb、wb、ab、rb+、wb+、ab+等组合，字符“b”表示 fopen() 函数打开的文件为二进制文件，而非纯文字文件。

(4) 输出：一个指向 FILE 类型的指针。

#### 2. 关闭文件函数 fclose()

(1) 格式：int fclose(FILE \* stream);

(2) 功能: 用来关闭先前fopen()打开的一个文件。此动作会让缓冲区内的数据写入文件中, 并释放系统所提供的文件资源。

(3) 输入参数说明:

stream: 指向要关闭文件的指针, 它是先前执行fopen()函数的返回值。

(4) 输出: 若关闭文件成功则返回0; 有错误发生时则返回EOF并把错误代码存到errno。

### 3. 读文件函数 fread()

(1) 格式: `size_t fread(void *buffer, size_t size, size_t count, FILE *stream);`

(2) 功能: 读二进制文件到内存。

(3) 输入参数说明:

buffer: 用于存放输入数据的缓冲区的首地址;

stream: 使用fopen()打开的文件的指针, 用于指示要读取的文件;

size: 每个数据块的字节数;

count: 要读入的数据块的个数;

size\*count: 表示要求读取的字节数。

(4) 输出: 实际读取的数据块的个数。

### 4. 写文件函数 fwrite()

(1) 格式: `size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);`

(2) 功能: 将数据写到二进制文件中。

(3) 输入参数说明:

buffer: 用于存放输出数据的缓冲区的首地址;

stream: 使用fopen()打开的文件的指针, 用于指示要写出的文件;

size: 每个数据块的字节数;

count: 要写出的数据块的个数;

size\*count: 表示要求写出的字符数。

(4) 输出: 实际写出的数据块的个数。

### 5. 判断文件结束函数 feof ()

(1) 格式: `int feof(FILE * stream)`

(2) 功能: 用来判断是否已读取到文件末尾。

(3) 输入参数说明:

stream: 使用 fopen()打开的文件的指针, 用于指示要判断的文件。

(4) 输出: 如果已读到文件尾则返回非零值, 其他情况返回 0。

### 6. 定位文件函数 fseek()

(1) 格式: `int fseek(FILE *stream, long offset, int origin );`

(2) 功能: 移动文件读写指针在文件中的位置。

(3) 输入参数说明:

stream: 使用fopen()打开的文件的指针, 用于指示要定位读写指针的文件;

offset: 位移量, 以字节为单位;

origin: 初始位置, 有三个常量:

SEEK\_CUR: 读写指针当前位置;

SEEK\_SET: 文件开头;

SEEK\_END: 文件末尾。

当 origin 值为 SEEK\_CUR 或 SEEK\_END 时, 参数 offset 可以为负值。

### 5.3 实例系统的设计与实现

本实例系统是仿照 FAT16 文件系统来设计实现的, 但根目录没有采用 FAT16 的固定位置、固定大小的根目录区, 而是以根目录文件的形式来实现的, 这也是目前主流文件系统对根目录的处理方式。

#### 5.3.1 数据结构设计

##### 1. 需要包含的头文件

(1) #include <stdio.h>

(2) #include <malloc.h>

(3) #include <string.h>

(4) #include <time.h>

##### 2. 定义的常量

(1) #define BLOCKSIZE 1024 磁盘块大小

(2) #define SIZE 1024000 虚拟磁盘空间大小

(3) #define END 65535 FAT 中的文件结束标志

(4) #define FREE 0 FAT 中盘块空闲标志

(5) #define ROOTBLOCKNUM 2 根目录区所占盘块总数

(6) #define MAXOPENFILE 10 最多同时打开文件个数

##### 3. 数据结构

(1) 文件控制块 FCB

用于记录文件的描述和控制信息, 每个文件设置一个 FCB, 它也是文件的目录项的内容。

typedef struct FCB //仿照 FAT16 设置的

{

char filename[8]; //文件名

char exname[3]; //文件扩展名

unsigned char attribute; //文件属性字段: 为简单起见, 我们只为文件设置

```

//了两种属性：值为 0 时表示目录文件，值为 1 时表示数据文件
unsigned short time;//文件创建时间
unsigned short data;//文件创建日期
unsigned short first;//文件起始盘块号
unsigned long length;//文件长度（字节数）
char free; //表示目录项是否为空，若值为 0，表示空，值为 1，表示已分配
.....
}fcb;

```

## （2）文件分配表 FAT

在本实例中，文件分配表有两个作用：一是记录磁盘上每个文件所占据的磁盘块的块号；二是记录磁盘上哪些块已经分配出去了，哪些块是空闲的，即起到了位示图的作用。若 FAT 中某个表项的值为 FREE，则表示该表项所对应的磁盘块是空闲的；若某个表项的值为 END，则表示所对应的磁盘块是某文件的最后一个磁盘块；若某个表项的值是其他值，则该值表示某文件的下一个磁盘块的块号。为了提高系统的可靠性，本实例中设置了两张 FAT 表，它们互为备份，每个 FAT 占据两个磁盘块。

```

typedef struct FAT
{
    unsigned short id;
}fat;

```

## （3）用户打开文件表 USEROPEN

当打开一个文件时，必须将文件的目录项中的所有内容全部复制到内存中，同时还要记录有关文件操作的动态信息，如读写指针的值等。在本实例中实现的是一个用于单用户单任务系统的文件系统，为简单起见，我们把用户文件描述符表和内存 FCB 表合在一起，称为用户打开文件表，表项数目为 10，即一个用户最多可同时打开 10 个文件。然后用一个数组来描述，则数组下标即某个打开文件的描述符。另外，我们在用户打开文件表中还设置了一个字段“char dir[80]”，用来记录每个打开文件所在的目录名，以方便用户打开不同目录下具有相同文件名的不同文件。

```

typedef struct USEROPEN
{
    char filename[8]; //文件名
    char exname[3]; //文件扩展名
    unsigned char attribute;//文件属性：值为 0 时表示目录文件，值为 1 时表示数据文件
    unsigned short time;//文件创建时间
    unsigned short data;//文件创建日期
    unsigned short first;//文件起始盘块号

```



```

    unsigned long length;//文件长度（对数据文件是字节数，对目录文件可以是目
录项个数）
    char free; //表示目录项是否为空，若值为 0，表示空，值为 1，表示已分配
    //前面内容是文件的 FCB 中的内容。
    // 下面设置的 dirno 和 diroff 记录了相应打开文件的目录项在父目录文件中的位
    置，//这样如果该文件的 fcb 被修改了，则要写回父目录文件时比较方便
    int dirno; //相应打开文件的目录项在父目录文件中的盘块号
    int diroff;// 相应打开文件的目录项在父目录文件的 dirno 盘块中的目录项序号
    char dir[MAXOPENFILE][80]; //相应打开文件所在的目录名，这样方便快速检查
    出指定文件是否已经打开
    int count; //读写指针在文件中的位置
    char fcbstate; //是否修改了文件的 FCB 的内容，如果修改了置为 1，否则为 0
    char topenfile; //表示该用户打开表项是否为空，若值为 0，表示为空，否则表
    示已被某打开文件占据
}useropen;

```

#### （4）引导块 BLOCK0

在引导块中主要存放逻辑磁盘的相关描述信息，比如磁盘块大小、磁盘块数量、文件分配表、根目录区、数据区在磁盘上的起始位置等。如果是引导盘，还要存放操作系统的引导信息。本实例是在内存的虚拟磁盘中创建一个文件系统，因此所包含的内容比较少，只有磁盘块大小、磁盘块数量、数据区开始位置、根目录文件开始位置等。

```

typedef struct BLOCK0 //引导块内容
{
    //存储一些描述信息，如磁盘块大小、磁盘块数量、最多打开文件数等、
    char information[200];
    unsigned short root; //根目录文件的起始盘块号
    unsigned char *startblock; //虚拟磁盘上数据区开始位置
}block0;

```

#### 4. 全局变量定义

- (1) unsigned char \*myvhard: 指向虚拟磁盘的起始地址
- (2) useropen openfilelist[MAXOPENFILE]: 用户打开文件表数组
- (3) useropen \*ptrcurdir: 指向用户打开文件表中的当前目录所在打开文件表项的位置;
- (4) char currentdir[80]: 记录当前目录的目录名（包括目录的路径）
- (5) unsigned char\* startp: 记录虚拟磁盘上数据区开始位置

#### 5. 虚拟磁盘空间布局

由于真正的磁盘操作需要涉及到设备的驱动程序,所以本实例是在内存中申请一块空间作为虚拟磁盘使用,我们的文件系统就建立在这个虚拟磁盘上。虚拟磁盘一共划分成 1000 个磁盘块,每个块 1024 个字节,其布局格式是模仿 FAT 文件系统设计的,其中引导块占一个盘块,两张 FAT 各占 2 个盘块,剩下的空间全部是数据区,在对虚拟磁盘进行格式化的时候,将把数据区第 1 块(即虚拟磁盘的第 6 块)分配给根目录文件,如图 3-3 所示:

块数	1块	2块	2块	995块
	引导块	FAT1	FAT2	数据区

图 5-3 虚拟磁盘空间布局

当然,也可以仿照 FAT16 文件系统,设置根目录区,其位置紧跟第 2 张 FAT 后面,大小也是固定的,这个思路相对要简单一点,请同学们自己去实现。

### 5.3.2 实例主要命令及函数设计

#### 1. 系统主函数 main()

- (1) 对应命令: 无
- (2) 命令调用格式: 无
- (3) 函数设计格式: void main()
- (4) 功能: 系统主函数
- (5) 输入: 无
- (6) 输出: 无
- (7) 函数需完成的工作:
  - ① 对前面定义的全局变量进行初始化;
  - ② 调用 startsys() 进入文件系统;
  - ③ 列出文件系统提供的各项功能及命令调用格式;
  - ④ 显示命令行提示符,等待用户输入命令;
  - ⑤ 将用户输入的命令保存到一个 buf 中;
  - ⑥ 对 buf 中的内容进行命令解析,并调用相应的函数执行用户键入的命令;
  - ⑦ 如果命令不是“my\_exitsys”,则命令执行完毕后转④。

#### 2. 进入文件系统函数 startsys()

- (1) 对应命令: 无
- (2) 命令调用格式: 无
- (3) 函数设计格式: void startsys()
- (4) 功能: 由 main() 函数调用,进入并初始化我们所建立的文件系统,以供用户使用。
- (5) 输入: 无
- (6) 输出: 无。
- (7) 函数需完成的工作:
  - ① 申请虚拟磁盘空间;

② 使用 c 语言的库函数 `fopen()` 打开 `myfsys` 文件：若文件存在，则转③；若文件不存在，则创建之，转⑤

③ 使用 c 语言的库函数 `fread()` 读入 `myfsys` 文件内容到用户空间中的一个缓冲区中，并判断其开始的 8 个字节内容是否为“10101010”（文件系统魔数），如果是，则转④；否则转⑤；

④ 将上述缓冲区中的内容复制到内存中的虚拟磁盘空间中；转⑦

⑤ 在屏幕上显示“`myfsys` 文件系统不存在，现在开始创建文件系统”信息，并调用 `my_format()` 对①中申请到的虚拟磁盘空间进行格式化操作。转⑥；

⑥ 将虚拟磁盘中的内容保存到 `myfsys` 文件中；转⑦

⑦ 使用 c 语言的库函数 `fclose()` 关闭 `myfsys` 文件；

⑧ 初始化用户打开文件表，将表项 0 分配给根目录文件使用，并填写根目录文件的相关信息，由于根目录没有上级目录，所以表项中的 `dirno` 和 `diroff` 分别置为 5（根目录所在起始块号）和 0；并将 `ptrcurdir` 指针指向该用户打开文件表项。

⑨ 将当前目录设置为根目录。

### 3. 磁盘格式化函数 `my_format()`

(1) 对应命令：`my_format`

(2) 命令调用格式：`my_format`

(3) 函数设计格式：`void my_format()`

(4) 功能：对虚拟磁盘进行格式化，布局虚拟磁盘，建立根目录文件（或根目录区）。

(5) 输入：无

(6) 输出：无。

(7) 函数需完成的工作：

① 将虚拟磁盘第一个块作为引导块，开始的 8 个字节是文件系统的魔数，记为“10101010”；在之后写入文件系统的描述信息，如 FAT 表大小及位置、根目录大小及位置、盘块大小、盘块数量、数据区开始位置等信息；

② 在引导块后建立两张完全一样的 FAT 表，用于记录文件所占据的磁盘块及管理虚拟磁盘块的分配，每个 FAT 占据两个磁盘块；对于每个 FAT 中，前面 5 个块设置为已分配，后面 995 个块设置为空闲；

③ 在第二张 FAT 后创建根目录文件 `root`，将数据区的第 1 块（即虚拟磁盘的第 6 块）分配给根目录文件，在该磁盘上创建两个特殊的目录项：“.”和“..”，其内容除了文件名不同之外，其他字段完全相同。

### 4. 更改当前目录函数 `my_cd()`

(1) 对应命令：`my_cd`

(2) 命令调用格式：`my_cd dirname`

(3) 函数设计格式：`void my_cd(char *dirname)`

(4) 功能：改变当前目录到指定的名为 `dirname` 的目录。

(5) 输入：

dirname: 新的当前目录的目录名;

(6) 输出: 无

(7) 函数需完成的工作:

① 调用 `my_open()` 打开指定目录名的父目录文件, 并调用 `do_read()` 读入该父目录文件内容到内存中;

② 在父目录文件中检查新的当前目录名是否存在, 如果存在则转③, 否则返回, 并显示出错信息;

③ 调用 `my_close()` 关闭①中打开的父目录文件;

④ 调用 `my_close()` 关闭原当前目录文件;

⑤ 如果新的当前目录文件没有打开, 则打开该目录文件; 并将 `ptrcurdir` 指向该打开文件表项;

⑥ 设置当前目录为该目录。

## 5. 创建子目录函数 `my_mkdir()`

(1) 对应命令: `my_mkdir`

(2) 命令调用格式: `my_mkdir dirname`

(3) 函数设计格式: `void my_mkdir(char *dirname)`

(4) 功能: 在当前目录下创建名为 `dirname` 的子目录。

(5) 输入:

dirname: 新建目录的目录名。

(6) 输出: 无。

(7) 函数需完成的工作:

① 调用 `do_read()` 读入当前目录文件内容到内存, 检查当前目录下新建目录文件是否重名, 若重名则返回, 并显示错误信息;

② 为新建子目录文件分配一个空闲打开文件表项, 如果没有空闲表项则返回-1, 并显示错误信息;

③ 检查 FAT 是否有空闲的盘块, 如有则为新建目录文件分配一个盘块, 否则释放①中分配的打开文件表项, 返回, 并显示错误信息;

④ 在当前目录中为新建目录文件寻找一个空闲的目录项或为其追加一个新的目录项; 需修改当前目录文件的长度信息, 并将当前目录文件的用户打开文件表项中的 `fcstate` 置为 1;

⑤ 准备好新建目录文件的 FCB 的内容, 文件的属性为目录文件, 以覆盖写方式调用 `do_write()` 将其填写到对应的空目录项中;

⑥ 在新建目录文件所分配到的磁盘块中建立两个特殊的目录项 “.” 和 “..” 目录项, 方法是: 首先在用户空间中准备好内容, 然后以截断写或者覆盖写方式调用 `do_write()` 将其写到③中分配到的磁盘块中;

⑦ 返回。

## 6. 删除子目录函数 `rmdir()`

- (1) 对应命令: `my_rmdir`
- (2) 命令调用格式: `my_rmdir dirname`
- (1) 函数设计格式: `void my_rmdir(char *dirname)`
- (2) 功能: 在当前目录下删除名为 `dirname` 的子目录。
- (3) 输入:
  - `dirname`: 欲删除目录的目录名。
- (4) 输出: 无。
- (5) 函数需完成的工作:

① 调用 `do_read()` 读入当前目录文件内容到内存, 检查当前目录下欲删除目录文件是否存在, 若不存在则返回, 并显示错误信息;

② 检查欲删除目录文件是否为空 (除了 “.” 和 “..” 外没有其他子目录和文件), 可根据其目录项中记录的文件长度来判断, 若不为空则返回, 并显示错误信息;

③ 检查该目录文件是否已经打开, 若已打开则调用 `my_close()` 关闭掉;

④ 回收该目录文件所占据的磁盘块, 修改 FAT;

⑤ 从当前目录文件中清空该目录文件的目录项, 且 `free` 字段置为 0: 以覆盖写方式调用 `do_write()` 来实现;

⑥ 修改当前目录文件的用户打开表项中的长度信息, 并将表项中的 `fcbsize` 置为 1;

⑦ 返回。

## 7. 显示目录函数 `my_ls()`

- (1) 对应命令: `my_ls`
- (2) 命令调用格式: `my_ls`
- (3) 函数设计格式: `void my_ls(void)`
- (4) 功能: 显示当前目录的内容 (子目录和文件信息)。
- (5) 输入: 无
- (6) 输出: 无
- (7) 函数需完成的工作:

① 调用 `do_read()` 读出当前目录文件内容到内存;

② 将读出的目录文件的信息按照一定的格式显示到屏幕上;

③ 返回。

## 8. 创建文件函数 `my_create()`

- (1) 对应命令: `my_create`
- (2) 命令调用格式: `my_create filename`
- (3) 函数设计格式: `int my_create (char *filename)`
- (4) 功能: 创建名为 `filename` 的新文件。
- (5) 输入:

`filename`: 新建文件的文件名, 可能包含路径。

(6) 输出: 若创建成功, 返回该文件的文件描述符 (文件打开表中的数组下标); 否

则返回-1。

(7) 函数需完成的工作：

① 为新文件分配一个空闲打开文件表项，如果没有空闲表项则返回-1，并显示错误信息；

② 若新文件的父目录文件还没有打开，则调用 `my_open()` 打开；若打开失败，则释放①中为新建文件分配的空闲文件打开表项，返回-1，并显示错误信息；

③ 调用 `do_read()` 读出该父目录文件内容到内存，检查该目录下新文件是否重名，若重名则释放①中分配的打开文件表项，并调用 `my_close()` 关闭②中打开的目录文件；然后返回-1，并显示错误信息；

④ 检查 FAT 是否有空闲的盘块，如有则为新文件分配一个盘块，否则释放①中分配的打开文件表项，并调用 `my_close()` 关闭②中打开的目录文件；返回-1，并显示错误信息；

⑤ 在父目录中为新文件寻找一个空闲的目录项或为其追加一个新的目录项；需修改该目录文件的长度信息，并将该目录文件的用户打开文件表项中的 `fcstate` 置为 1；

⑥ 准备好新文件的 FCB 的内容，文件的属性为数据文件，长度为 0，以覆盖写方式调用 `do_write()` 将其填写到⑤中分配到的空目录项中；

⑦ 为新文件填写①中分配到的空闲打开文件表项，`fcstate` 字段值为 0，读写指针值为 0；

⑧ 调用 `my_close()` 关闭②中打开的父目录文件；

⑨ 将新文件的打开文件表项序号作为其文件描述符返回。

## 9. 删除文件函数 `my_rm()`

(1) 对应命令：`my_rm`

(2) 命令调用格式：`my_rm filename`

(3) 函数设计格式：`void my_rm(char *filename)`

(4) 功能：删除名为 `filename` 的文件。

(5) 输入：

`filename`：欲删除文件的文件名，可能还包含路径。

(6) 输出：无。

(7) 函数需完成的工作：

① 若欲删除文件的父目录文件还没有打开，则调用 `my_open()` 打开；若打开失败，则返回，并显示错误信息；

② 调用 `do_read()` 读出该父目录文件内容到内存，检查该目录下欲删除文件是否存在，若不存在则返回，并显示错误信息；

③ 检查该文件是否已经打开，若已打开则关闭掉；

④ 回收该文件所占据的磁盘块，修改 FAT；

⑤ 从文件的父目录文件中清空该文件的目录项，且 `free` 字段置为 0；以覆盖写方式调用 `do_write()` 来实现；

⑥ 修改该父目录文件的用户打开文件表项中的长度信息，并将该表项中的 `fcstate` 置为 1；

⑦ 返回。

## 10. 打开文件函数 `my_open()`

(1) 对应命令: `my_open`

(2) 命令调用格式: `my_open filename`

(3) 函数设计格式: `int my_open(char *filename)`

(4) 功能: 打开当前目录下名为 `filename` 的文件。

(5) 输入:

`filename`: 欲打开文件的文件名

(6) 输出: 若打开成功, 返回该文件的描述符 (在用户打开文件表中表项序号); 否则返回-1。

(7) 函数需完成的工作:

① 检查该文件是否已经打开, 若已打开则返回-1, 并显示错误信息;

② 调用 `do_read()` 读出父目录文件的内容到内存, 检查该目录下欲打开文件是否存在, 若不存在则返回-1, 并显示错误信息;

③ 检查用户打开文件表中是否有空表项, 若有则为欲打开文件分配一个空表项, 若没有则返回-1, 并显示错误信息;

④ 为该文件填写空白用户打开文件表表项内容, 读写指针置为 0;

⑤ 将该文件所分配到的空白用户打开文件表表项序号 (数组下标) 作为文件描述符 `fd` 返回。

## 11. 关闭文件函数 `my_close()`

(1) 对应命令: `my_close`

(2) 命令调用格式: `my_close fd`

(3) 函数设计格式: `void my_close(int fd)`

(4) 功能: 关闭前面由 `my_open()` 打开的文件描述符为 `fd` 的文件。

(5) 输入:

`fd`: 文件描述符。

(6) 输出: 无。

(7) 函数需完成的工作:

① 检查 `fd` 的有效性 (`fd` 不能超出用户打开文件表所在数组的最大下标), 如果无效则返回-1;

② 检查用户打开文件表表项中的 `fcbstate` 字段的值, 如果为 1 则需要将该文件的 FCB 的内容保存到虚拟磁盘上该文件的目录项中, 方法是: 打开该文件的父目录文件, 以覆盖写方式调用 `do_write()` 将欲关闭文件的 FCB 写入父目录文件的相应盘块中;

③ 回收该文件占据的用户打开文件表表项 (进行清空操作), 并将 `topenfile` 字段置为 0;

④ 返回。

## 12. 写文件函数 `my_write()`

- (1) 对应命令: `my_write`
- (2) 命令调用格式: `my_write fd`
- (3) 函数设计格式: `int my_write(int fd)`

(4) 功能: 将用户通过键盘输入的内容写到 `fd` 所指定的文件中。磁盘文件的读写操作都必须以完整的数据块为单位进行, 在写操作时, 先将数据写在缓冲区中, 缓冲区的大小与磁盘块的大小相同, 然后再将缓冲区中的数据一次性写到磁盘块中; 读出时先将一个磁盘块中的内容读到缓冲区中, 然后再传送到用户区。本实例为了简便起见, 没有设置缓冲区管理, 只是在读写文件时由用户使用 `malloc()` 申请一块空间作为缓冲区, 读写操作结束后使用 `free()` 释放掉。

写操作常有三种方式: 截断写、覆盖写和追加写。截断写是放弃原来文件的内容, 重新写文件; 覆盖写是修改文件在当前读写指针所指的位置开始的部分内容; 追加写是在原文件的最后添加新的内容。在本实例中, 输入写文件命令后, 系统会出现提示让用户选择其中的一种写方式, 并将随后键盘输入的内容按照所选的方式写到文件中, 键盘输入内容通过 `CTR+Z` 键 (或其他设定的键) 结束。

- (5) 输入:

`fd`: `open()` 函数的返回值, 文件的描述符;

- (6) 输出: 实际写入的字节数。

- (7) 函数需完成的工作:

① 检查 `fd` 的有效性 (`fd` 不能超出用户打开文件表所在数组的最大下标), 如果无效则返回-1, 并显示出错信息;

② 提示并等待用户输入写方式: (1: 截断写; 2: 覆盖写; 3: 追加写)

③ 如果用户要求的写方式是截断写, 则释放文件除第一块外的其他磁盘空间内容 (查找并修改 FAT 表), 将内存用户打开文件表项中文件长度修改为 0, 将读写指针置为 0 并转④;

④; 如果用户要求的写方式是追加写, 则修改文件的当前读写指针位置到文件的末尾, 并转④; 如果写方式是覆盖写, 则直接转④;

④ 提示用户: 整个输入内容通过 `CTR+Z` 键 (或其他设定的键) 结束; 用户可分多次输入写入内容, 每次用回车结束;

⑤ 等待用户从键盘输入文件内容, 并将用户的本次输入内容保存到一个临时变量 `text[]` 中, 要求每次输入以回车结束, 全部结束用 `CTR+Z` 键 (或其他设定的键);

⑥ 调用 `do_write()` 函数将通过键盘键入的内容写到文件中。

⑦ 如果 `do_write()` 函数的返回值为非负值, 则将实际写入字节数增加 `do_write()` 函数返回值, 否则显示出错信息, 并转⑨;

⑧ 如果 `text[]` 中最后一个字符不是结束字符 `CTR+Z`, 则转⑦继续进行写操作; 否则转⑨;

⑨ 如果当前读写指针位置大于用户打开文件表项中的文件长度, 则修改打开文件表项中的文件长度信息, 并将 `fcbsize` 置 1;

⑩ 返回实际写入的字节数。

### 13. 实际写文件函数 `do_write()`



- (1) 对应命令：无
- (2) 命令调用格式：无
- (3) 函数设计格式：int my\_write(int fd, char \*text, int len, char wstyle)
- (4) 功能：被写文件函数 my\_write() 调用，用来将键盘输入的内容写到相应的文件中。

(5) 输入：

fd: open() 函数的返回值，文件的描述符；  
 text: 指向要写入的内容的指针；  
 len: 本次要求写入字节数  
 wstyle: 写方式

(6) 输出：实际写入的字节数。

(7) 函数需完成的工作：

① 用 malloc() 申请 1024B 的内存空间作为读写磁盘的缓冲区 buf，申请失败则返回-1，并显示出错信息；

② 将读写指针转化为逻辑块块号和块内偏移 off，并利用打开文件表表项中的首块号及 FAT 表的相关内容将逻辑块块号转换成对应的磁盘块块号 blkno；如果找不到对应的磁盘块，则需要检索 FAT 为该逻辑块分配一新的磁盘块，并将对应的磁盘块块号 blkno 登记到 FAT 中，若分配失败，则返回-1，并显示出错信息；

③ 如果是覆盖写，或者如果当前读写指针所对应的块内偏移 off 不等于 0，则将块号为 blkno 的虚拟磁盘块全部 1024B 的内容读到缓冲区 buf 中；否则使用 ASCII 码 0 清空 buf；

④ 将 text 中未写入的内容暂存到缓冲区 buff 的第 off 字节开始的位置，直到缓冲区满，或者接收到结束字符 CTR+Z 为止；将本次写入字节数记录到 tmpflen 中；

⑤ 将 buf 中 1024B 的内容写入到块号为 blkno 的虚拟磁盘块中；

⑥ 将当前读写指针修改为原来的值加上 tmpflen；并将本次实际写入的字节数增加 tmpflen；

⑦ 如果 tmpflen 小于 len，则转②继续写入；否则转⑧；

⑧ 返回本次实际写入的字节数。

#### 14. 读文件函数 my\_read()

(1) 对应命令：my\_read

(2) 命令调用格式：my\_read fd len

(3) 函数设计格式：int myread (int fd, int len)

(4) 功能：读出指定文件中从读写指针开始的长度为 len 的内容到用户空间中。

(5) 输入：

fd: open() 函数的返回值，文件的描述符；  
 len: 要从文件中读出的字节数。

(6) 输出：实际读出的字节数。

(7) 函数需完成的工作：

① 定义一个字符型数组 text[len]，用来接收用户从文件中读出的文件内容；

- ② 检查 fd 的有效性 (fd 不能超出用户打开文件表所在数组的最大下标)，如果无效则返回-1，并显示出错信息；
- ③ 调用 do\_read() 将指定文件中的 len 字节内容读出到 text[] 中；
- ④ 如果 do\_read() 的返回值为负，则显示出错信息；否则将 text[] 中的内容显示到屏幕上；
- ⑤ 返回。

## 15. 实际读文件函数 do\_read()

- (1) 对应命令：无
- (2) 命令调用格式：无
- (3) 函数设计格式：int do\_read (int fd, int len, char \*text)
- (4) 功能：被 my\_read() 调用，读出指定文件中从读写指针开始的长度为 len 的内容到用户空间的 text 中。
- (5) 输入：
  - fd: open() 函数的返回值，文件的描述符；
  - len: 要求从文件中读出的字节数。
  - text: 指向存放读出数据的用户区地址
- (6) 输出：实际读出的字节数。
- (7) 函数需完成的工作：
  - ① 使用 malloc() 申请 1024B 空间作为缓冲区 buf，申请失败则返回-1，并显示出错信息；
  - ② 将读写指针转化为逻辑块块号及块内偏移量 off，利用打开文件表表项中的首块号查找 FAT 表，找到该逻辑块所在的磁盘块块号；将该磁盘块块号转化为虚拟磁盘上的内存位置；
  - ③ 将该内存位置开始的 1024B（一个磁盘块）内容读入 buf 中；
  - ④ 比较 buf 中从偏移量 off 开始的剩余字节数是否大于等于应读写的字节数 len，如果是，则将从 off 开始的 buf 中的 len 长度的内容读入到 text[] 中；否则，将从 off 开始的 buf 中的剩余内容读入到 text[] 中；
  - ⑤ 将读写指针增加④中已读字节数，将应读写的字节数 len 减去④中已读字节数，若 len 大于 0，则转②；否则转⑥；
  - ⑥ 使用 free() 释放①中申请的 buf。
  - ⑦ 返回实际读出的字节数。

## 16. 退出文件系统函数 my\_exitsys()

- (1) 对应命令：my\_exitsys
- (2) 命令调用格式：my\_exitsys
- (1) 函数设计格式：void my\_exitsys()
- (2) 功能：退出文件系统。
- (3) 输入：无

(4) 输出：无。

(5) 函数需完成的工作：

- ① 使用 C 库函数 `fopen()` 打开磁盘上的 `myfsys` 文件；
- ② 将虚拟磁盘空间中的所有内容保存到磁盘上的 `myfsys` 文件中；
- ③ 使用 c 语言的库函数 `fclose()` 关闭 `myfsys` 文件；
- ④ 撤销用户打开文件表，释放其内存空间
- ④ 释放虚拟磁盘空间。