

---

华中科技大学计算机学院  
操作系统课程设计指导书

---

# 目 录

第 1 章	Linux 下 C 编程的相关知识.....	1
1.1.	Linux 下的 C 编程.....	1
1.	源程序的编译.....	1
2.	头文件和系统求助.....	2
1.2.	文件拷贝实例.....	2
1.3.	并发进程显示实例.....	6
1.	实现原理.....	7
2.	注意要点.....	7
3.	特别强调: .....	7
第 2 章	系统调用相关知识.....	9
2.1.	Linux 系统调用机制.....	9
1.	内核中系统调用的过程.....	9
2.	用户程序中系统调用的过程.....	16
2.2.	添加新的系统调用 (2.6.18 内核) .....	19
1.	步骤 1: 准备.....	19
2.	步骤 2: 下载源代码.....	19
3.	步骤 3: 修改相应内核文件.....	19
4.	步骤 4: 开始对新的内核进行编译.....	21
2.3.	对新加的系统调用进行测试.....	23
1.	旧版本的测试方法.....	23
2.	新版本的测试方法.....	23
2.4.	添加新的系统调用 (3.2.4 内核) .....	24
1.	步骤 1: 准备.....	24
2.	步骤 2: 修改相应内核文件.....	24
2.	步骤 4: 开始对新的内核进行编译.....	26
2.5.	对新加的系统调用进行测试 (3.2.4 内核) .....	27
1.	测试方法.....	27

---

第 3 章 设备驱动相关知识.....	28
3.1. 基础知识.....	28
3.2. 添加新模块的基本步骤.....	29
1. 写设备驱动源代码: .....	29
2. 编译.....	31
3. 挂载内核中模块.....	32
4. 创建新的虚拟设备文件.....	33
5. 测试新的设备驱动.....	33
6. 卸载操作.....	33
第 4 章 /proc 文件相关知识.....	34
4.1. 实现原理.....	34
4.2. 实现方法.....	34

---

## 第 1 章 Linux 下 C 编程的相关知识

### 1.1. Linux 下的 C 编程

#### 1. 源程序的编译

在 Linux 下面,如果要编译一个 C 语言源程序,我们要使用 GNU 的 gcc 编译器. 下面我们以一个实例来说明如何使用 gcc 编译器.

假设我们有下面一个非常简单的源程序(hello.c):

```
int main(int argc,char **argv)
{
    printf("Hello Linux\n");
}
```

要编译这个程序,我们只要在命令行下执行:

```
gcc -o hello hello.c
```

gcc 编译器就会为我们生成一个名为 hello 的可执行文件.在当前目录下执行 ./hello 就可以看到程序的输出结果了.命令行中 gcc 表示我们是用 gcc 来编译我们的源程序,-o 选项表示我们要求编译器给我们输出的可执行文件名为 hello 而 hello.c 是我们的源程序文件.

gcc 编译器有许多选项,一般来说我们只要知道其中的几个就够了. -o 选项我们已经知道了,表示我们要求输出的可执行文件名. -c 选项表示我们只要求编译器输出目标代码,而不必要输出可执行文件. -g 选项表示我们要求编译器在编译的时候提供我们以后对程序 进行调试的信息. 知道了这三个选项,我们就可以编译我们自己所写的简单的源程序了,如果你想要知道更多的选项,可以查看 gcc 的帮助文档 (可以用 man gcc 来查看),那里有着许多对其它选项的详细说明.

---

## 2. 头文件和系统求助

有时候我们只知道一个函数的大概形式,不记得确切的表达式,或者是不记得该函数在哪个头文件进行了说明。这个时候我们可以求助系统。比如说我们想知道 `fread` 这个函数的确切形式,我们只要执行 `man fread` 系统就会输出函数的详细解释和这个函数所在的头文件`<stdio.h>`说明。如果我们要 `write` 这个函数的说明,我们执行 `man write` 时,输出的结果却不是我们所需要的。因为我们要的是 `write` 这个函数的说明,可是出来的却是 `write` 这个命令的说明。为了得到 `write` 的函数说明我们要用 `man 2 write`。2 表示我们用的 `write` 这个函数是系统调用函数,还有一个我们常用的是 3 表示函数是 C 的库函数。记住不管什么时候, `man` 都是我们的最好助手。

### 1.2. 文件拷贝实例

本节主要针对一个实例: 编一个 C 程序, 其内容为实现文件拷贝的功能。

主要用到如下几个函数:

`open`: 打开文件

`close`: 关闭文件

`read`: 读操作

`write`: 写操作

当我们需要打开一个文件进行读写操作的时候,我们可以使用系统调用函数 `open`. 使用完成以后我们调用另外一个 `close` 函数进行关闭操作.

```
#include <fcntl.h>    //注意: 这个字母是小写的 L, 而不是数字 1。
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int open(const char *pathname,int flags);
int open(const char *pathname,int flags,mode_t mode);
int close(int fd);
```

`open` 函数有两个形式,其中 `pathname` 是我们要打开的文件名(包含路径名称,缺省是认为在 当前路径下面).`flags` 可以去下面的一个值或者是几个值的组合. 组合使用

---

时候用竖线 | 隔开。

O\_RDONLY:以只读的方式打开文件. //是字母 O,

O\_WRONLY:以只写的方式打开文件.

O\_RDWR:以读写的方式打开文件.

O\_APPEND:以追加的方式打开文件.

O\_CREAT:创建一个文件. //创建一个不存在的文件.

O\_EXEC:如果使用了 O\_CREAT 而且文件已经存在,就会发生一个错误.

O\_NOBLOCK:以非阻塞的方式打开一个文件.

O\_TRUNC:如果文件已经存在,则删除文件的内容.

前面三个标志只能使用任意的一个.

如果使用了 O\_CREATE 标志,那么我们要使用 open 的第二种形式.还要指定 mode 标志,用来表示文件的访问权限.

如果我们打开文件成功,open 会返回一个文件描述符,如 fd.我们以后对文件的所有操作就可以对这个文件描述符进行操作了.

当我们操作完成以后,我们要关闭文件了,只要调用 close 就可以了,其中 fd 是我们关闭的文件的描述符.

文件打开了以后,我们就要对文件进行读写了。我们可以调用函数 read 和 write 进行文件的读写.

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

```
ssize_t read(int fd, void *buffer,size_t count);
```

```
ssize_t write(int fd, const void *buffer,size_t count);
```

fd 是我们要进行读写操作的文件描述符,buffer 是我们要写入文件或读出文件内容的内存地址。count 是我们要读写的字节数。

对于普通的文件 read 从指定的文件(fd)中读取 count 字节到 buffer 缓冲区中(记住我们必须提供一个足够大的缓冲区),同时返回 count。如果 read 读到了文件的结尾或者被一个信号所中断,返回值会小于 count。如果是由信号中断引起返回,而且没有返回数据,read 会返回 -1,且设置 errno 为 EINTR。当程序读到了文件结尾的时候,read 会返回 0。

write 从 buffer 中写 count 字节到文件 fd 中,成功时返回实际所写的字节数。

---

下面我们学习一个实例,这个实例用来拷贝文件.

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

#define BUFFER_SIZE 1024    //缓冲区大小

int main(int argc,char **argv)
{
    int from_fd,to_fd;
    int bytes_read,bytes_write;
    char buffer[BUFFER_SIZE];    //设定一个缓冲区
    char *ptr;
    if(argc!=3)    //三个参数
    {
        fprintf(stderr,"Usage:%s fromfile tofile\n\a",argv[0]);
        return(-1);
    }
    /* 打开源文件 */
    if((from_fd=open(argv[1],O_RDONLY))== -1)
    {
        fprintf(stderr,"Open %s Error:%s\n",argv[1],strerror(errno));
        return(-1);
    }
    /* 创建目的文件 */
    if((to_fd=open(argv[2],O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR))== -1)
    {
        fprintf(stderr,"Open %s Error:%s\n",argv[2],strerror(errno));
```

---

```
return(-1);
}

/* 以下代码是一个经典的拷贝文件的代码 */
while(bytes_read=read(from_fd,buffer,BUFFER_SIZE))
{
/* 一个致命的错误发生了 */
if((bytes_read==-1)&&(errno!=EINTR)) break;
else if(bytes_read>0)
{
ptr=buffer;
while(bytes_write=write(to_fd,ptr,bytes_read))
{
/* 一个致命错误发生了 */
if((bytes_write==-1)&&(errno!=EINTR))break;
/* 写完了所有读的字节 */
else if(bytes_write==bytes_read) break;
/* 只写了一部分,继续写 */
else if(bytes_write>0)
{
ptr+=bytes_write;
bytes_read-=bytes_write;
}
}
/* 写的时候发生的致命错误 */
if(bytes_write==-1)break;
}
}

close(from_fd);
close(to_fd);
```



---

```
return(1);  
}
```

### 1.3. 并发进程显示实例

本节针对的实例：编一个 C 程序，其内容为分窗口同时显示三个并发进程的运行结果。要求使用 Linux 的图形库。

#### 进程介绍

通俗的讲，程序是一个包含可以执行代码的文件，是一个静态的文件。而进程是一个开始执行但是还没有结束的程序的实例。一个程序可能有许多进程，而每一个进程又可以有许多子进程。依次循环下去，而产生子孙进程。为了区分各个不同的进程，系统给每一个进程分配了一个 ID(就象我们的身份证)以便识别。为了充分的利用资源，系统还对进程区分了不同的状态，将进程分为新建、运行、阻塞、就绪和完成五个状态。

系统调用 `getpid` 可以得到进程的 ID，而 `getppid` 可以得到父进程(创建调用该函数进程的进程)的 ID：

```
#include <unistd>  
  
pid_t getpid(void);  
pid_t getppid(void);
```

如果要创建一个进程，可使用系统调用 `fork()`。

```
#include <unistd.h>  
  
pid_t fork();
```

当一个进程调用了 `fork` 以后，系统会创建一个子进程。这个子进程和父进程不同的地方只有他的进程 ID 和父进程 ID，其他的都是一样，就像父进程克隆(clone)自己一样。当然创建两个一模一样的进程是没有意义的。为了区分父进程和子进程，我们必须跟踪 `fork` 的返回值。当 `fork` 调用失败的时候(内存不足或者是用户的最大进程数已到) `fork` 返回 -1，否则 `fork` 的返回值有重要的作用。对于父进程 `fork` 返回子进程的 ID，而对于 `fork` 子进程返回 0。我们就是根据这个返回值来区分父子进程的。

---

## 1. 实现原理

先在父进程中利用 `fork()` 函数创建一个子进程，再在子进程中创建一个子进程，然后在各个进程的运行过程中分别生成一个窗口，同时在相应的进程窗口中显示该进程的 ID。生成窗口时可使用图形库 GTK 2.0，在每个窗口中加入几个构件（button, progress bars, label 等），并将每个构件所产生的动作与相应的信号处理函数相连接。

Linux 环境中，创建进程只需调用 `fork()` 函数即可。进程调用 `fork` 后，系统会创建一个子进程，此子进程与父进程唯一不同的地方在于其进程 ID 与父进程 ID：对于父进程，`fork` 返回子进程的 ID，对于子进程则返回 0，系统就是通过此返回值的不同来区分父子进程的。若 `fork` 调用失败，则返回 -1。

GTK (GIMP Toolkit) 是一个图形用户编程接口工具，本次实验将会用到的主要内容是控件、消息处理器和回调函数。利用控件可以实现一些图形的显示，比如显示窗口等等。消息处理器等待事件的发生（关闭窗口、点击按钮等），并捕获该信号，告诉 GTK 程序应该调用哪个回调函数进行相应的处理，并在终端中显示结果。详细内容请参见《GTK 2.0 教程》（可从网上下载）。

## 2. 注意要点

窗口、进度条和按钮的创建；

程序流程；

如何同时显示三个并发进程；

编译命令；

## 3. 特别强调：

**编译 GTK 程序与编译普通的 C 程序需要用到不同的命令。** 比如程序名为 `threeproc`，则编译命令为：`gcc `pkg-config --cflags --libs gtk+-2.0`threeproc.c -o threeproc`

其中 `pkg-config` 读取 GTK 附带的 `.pc` 文件来决定编译 GTK 程序需要的编译选项，`pkg-config --cflags gtk+-2.0` 列出 include 目录，`pkg-config --libs gtk+-2.0` 列出编译连接库。命令中的单引号是键盘上“1”键前面的那个，而不是回车键左边的那个，

---

否则会出错。--libs、--cflags 中都是两个横线,

注意 , 在不同的系统下, 编译命令的参数顺序可能略有不同, 如:

```
gcc 1_2.c -o 1_2`pkg-config --cflags --libs gtk+-2.0`
```

在某次实验的时候, 就**不能通过编译**, 提示 `1_2.c:1:21: 错误: gtk/gtk.h: 没有那个文件或目录`

但更换一下参数顺序就可以了:

```
gcc `pkg-config --cflags --libs gtk+-2.0`1_2.c -o 1_2
```

这点大家要注意, 当出现问题的要善于解决, 自己利用网络 and 资料, 进行处理, 多思考, 多尝试。

---

## 第 2 章 系统调用相关知识

**系统调用是应用程序和操作系统内核之间的功能接口**,通过系统调用进程可由用户模式转入内核模式,在内核模式下完成相应的服务之后再返回到用户模式。系统调用的主要目的是使得用户可以使用操作系统提供的有关设备管理、输入/输出系统、文件系统和进程控制、通信以及存储管理等方面的功能,而不必了解系统程序的内部结构和有关硬件细节,从而起到减轻用户负担和保护系统以及提高资源利用率的作用。

### 2.1. Linux 系统调用机制

#### 1. 内核中系统调用的过程

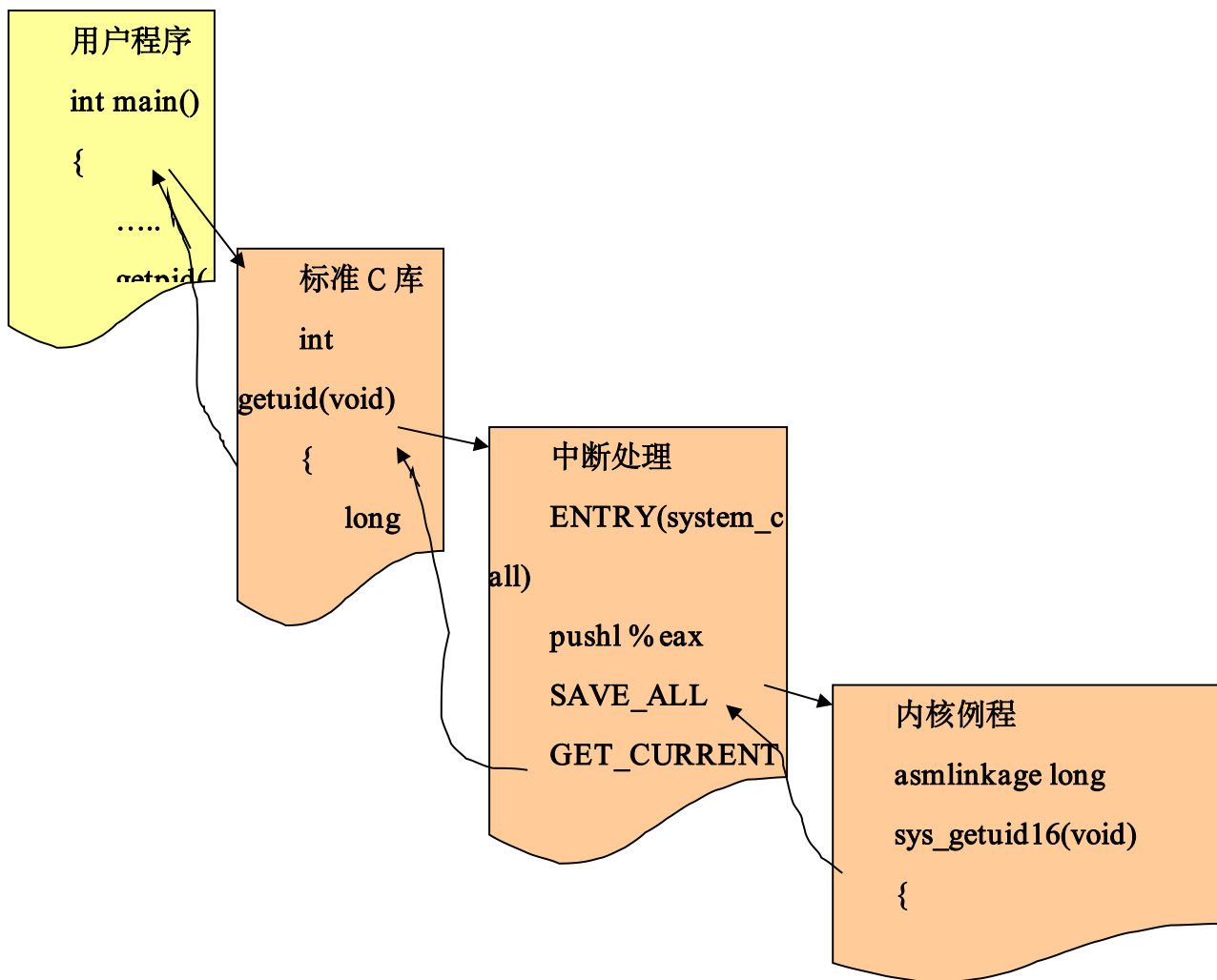
在 Linux 系统中, **系统调用是作为一种异常类型实现的**,它将执行相应的机器代码指令来产生异常信号。产生中断或异常的重要效果是系统自动将用户态切换为核心态来对它进行处理。这就是说,执行系统调用异常指令时,自动地将系统切换为核心态(模式切换, mode switch),并安排异常处理程序的执行。用户态的程序只有通过门(gate)陷入(trap)到系统内核中去(执行 int 指令),才能执行一些具有特权的内核函数。系统调用完成后,系统执行另一组特征指令(iret 指令)将系统返回到用户态,控制权返回给进程。

**Linux 用来实现系统调用异常的实际指令是:**

**int \$0x80**

这一指令使用中断/异常向量号 128 (即 16 进制的 80) 将控制权转移给内核(进行模式切换)。为达到在使用系统调用时不必用机器指令编程, 在标准的 C 语言库中为每一系统调用提供了一段短的子程序,完成机器代码的编程工作。事实上,机器代码段非常简短。它所要做的只是将送给系统调用的参数加载到 CPU 寄存器中,接着执行 int \$0x80 指令。然后运行系统调用, 系统调用的返回值将送入 CPU 的一个寄存器中,标准的库子程序取得这一返回值,并将它送回用户程序。

下面以 getuid()系统调用为例来看调用过程:



我们可以看到其中有一些宏定义,我们可以看看这些宏的定义  
(arch/i386/kernel/entry.S).

```

.....
#define SAVE_ALL \
    cld; \
    pushl %es; \
    pushl %ds; \
    pushl %eax; \
    pushl %ebp; \
    pushl %edi; \
    pushl %esi; \
    pushl %edx; \

```

---

```
    pushl %ecx; \
    pushl %ebx; \
    movl $(__USER_DS),%edx; \
    movl %edx,%ds; \
    movl %edx,%es;
```

我们可以看到 SAVE\_ALL 主要是保存寄存器信息,即现场保留.其中, `movl $(__USER_DS), %edx;` 从这一句开始是重新填充 DS, ES 段。

```
#define RESTORE_INT_REGS \
```

```
    popl %ebx; \
    popl %ecx; \
    popl %edx; \
    popl %esi; \
    popl %edi; \
    popl %ebp; \
    popl %eax
```

```
#define RESTORE_REGS \
```

```
    RESTORE_INT_REGS; \
1: popl %ds; \
2: popl %es; \
.section .fixup,"ax"; \
3: movl $0,(%esp); \
    jmp 1b; \
4: movl $0,(%esp); \
    jmp 2b; \
.previous; \
.section __ex_table,"a"; \
    .align 4; \
    .long 1b,3b; \
    .long 2b,4b; \
```

---

.previous

ENTRY(ret\_from\_fork)

```
    pushl %eax
    call schedule_tail
    GET_THREAD_INFO(%ebp)
    popl %eax
    jmp syscall_exit
```

这里主要完成现场恢复并返回。

ENTRY(system\_call)

```
    pushl %eax          # save orig_eax
    SAVE_ALL
    GET_THREAD_INFO(%ebp)
                        # system call tracing in operation

    /* Note, _TIF_SECCOMP is bit number 8, and so it needs testw and not testb */
    testw
    $(_TIF_SYSCALL_TRACE|_TIF_SYSCALL_AUDIT|_TIF_SECCOMP),TI_flags(%ebp)
    jnz syscall_trace_entry
    cmpl $(nr_syscalls), %eax
    jae syscall_badsys

syscall_call:
    call *sys_call_table(,%eax,4)
    movl %eax,EAX(%esp)    # store the return value

syscall_exit:
    cli                  # make sure we don't miss an interrupt
                        # setting need_resched or sigpending
                        # between sampling and the iret

    movl TI_flags(%ebp), %ecx
    testw $_TIF_ALLWORK_MASK, %cx    # current->work
    jne syscall_exit_work
```

---

```

restore_all:
    movl EFLAGS(%esp), %eax    # mix EFLAGS, SS and CS
    # Warning: OLDSS(%esp) contains the wrong/random values if we
    # are returning to the kernel.
    # See comments in process.c:copy_thread() for details.
    movb OLDSS(%esp), %ah
    movb CS(%esp), %al
    andl $(VM_MASK | (4 << 8) | 3), %eax
    cmpl $((4 << 8) | 3), %eax
    je ldt_ss                 # returning to user-space with LDT SS

restore_nocheck:
    RESTORE_REGS
    addl $4, %esp

1: iret

```

这一段中，主要是完成调用。**eax**放置的是系统调用号，因为eax有可能被使用，所以先保存其值。call \*sys\_call\_table(,%eax,4)这一句是计算调用的入口。

其中,sys\_call\_table 是 LINUX 的系统调用表,它存在目录 arch/i386/kernel/sys\_call\_table.S 下。

```

.data
ENTRY(sys_call_table)
    .long sys_restart_syscall /* 0 - old "setup()" system call, used for restarting */
    .long sys_exit
    .long sys_fork
    .long sys_read
    .long sys_write
    .long sys_open           /* 5 */
    .....
    .....
    .long sys_mq_timedreceive /* 280 */

```



---

```
.long sys_mq_notify
.long sys_mq_getsetattr
.long sys_ni_syscall    /* reserved for kexec */
.long sys_waitid
.long sys_ni_syscall    /* 285 */ /* available */
.long sys_add_key
.long sys_request_key
.long sys_keyctl
```

.代表当前地址， sys\_call\_table 代表数组首地址。这个表依次保存所有系统调用的函数指针， 以方便总的系统调用处理函数(system\_call)进行索引。

调用具体的实现在 kernel/sys.c 中。

```
asmlinkage long
sys_getuid16(void)
{
    return hig2lowuid(current_uid);
}
```

刚才我们提到，这一指令使用中断/异常向量号 128（即 16 进制的 80）将控制权转移给内核，那么中断向量是怎么形成的。它的定义在(arch/i386/kernel/traps.c)中。

```
void __init trap_init(void)
{
    .....
    set_trap_gate(0,&divide_error);
    set_trap_gate(1,&debug);
    set_intr_gate(2,&nmi);
    set_system_gate(3,&int3);    /* int3-5 can be called from all */
    set_system_gate(4,&overflow);
    set_system_gate(5,&bounds);
    set_trap_gate(6,&invalid_op);
    set_trap_gate(7,&device_not_available);
```

---

```

    set_trap_gate(8,&double_fault);
    set_trap_gate(9,&coprocessor_segment_overrun);
    set_trap_gate(10,&invalid_TSS);
    set_trap_gate(11,&segment_not_present);
    set_trap_gate(12,&stack_segment);
    set_trap_gate(13,&general_protection);
    set_intr_gate(14,&page_fault);
    set_trap_gate(15,&spurious_interrupt_bug);
    set_trap_gate(16,&coprocessor_error);
    set_trap_gate(17,&alignment_check);
    set_trap_gate(18,&machine_check);
    set_trap_gate(19,&simd_coprocessor_error);
    set_system_gate(&system_call);
    .....
}

```

上一句就是设置 `system_call` 的值。`SYSCALL_VECTOR` 的值就是 `0X80`。

### 那么概括起来，系统调用的过程大致如下：

#### (1) 系统调用初始化：

在 `traps.c` 中，系统在初始化程序 `trap_init()` 中，通过调用 `set_system_gate(0x80,*system_call)` 完成中断描述表的填充。这样当每次用户执行指令 `int 0x80` 时，系统能把控制转移到 `entry.S` 中的函数中去。

#### (2) 系统调用执行：

`system_call` 会根据用户传进来系统调用号，在系统调用表 `system_call` 中找到相应偏移地址的内核处理函数，进行相应的处理。当然在这个过程之前，要保存环境 (`SAVE_ALL`)。

#### (3) 系统调用的返回

系统调用处理完毕后，通过 `sys_call_exit` 返回。返回之前，程序会检查一些变量，相应地返回。不一定是返回到用户进程。真正返回到用户空间时，要恢复环境

---

(restore\_all)。

## 2. 用户程序中系统调用的过程

在前面提到 `system_call` 会根据用户传进来系统调用号，在系统调用表 `system_call` 中寻找到相应偏移地址的内核处理函数，进行相应的处理。那么系统调用号怎么产生，在 `include/asm-i386/unistd.h` 中可以看到系统调用号的定义。

```
#define __NR_restart_syscall    0
#define __NR_exit               1
#define __NR_fork               2
#define __NR_read               3
#define __NR_write              4
#define __NR_open               5
#define __NR_close              6
#define __NR_waitpid            7
#define __NR_creat              8
#define __NR_link               9
.....
#define __NR_mq_open            277
#define __NR_mq_unlink          (__NR_mq_open+1)
#define __NR_mq_timedsend       (__NR_mq_open+2)
#define __NR_mq_timedreceive    (__NR_mq_open+3)
#define __NR_mq_notify          (__NR_mq_open+4)
#define __NR_mq_getsetattr      (__NR_mq_open+5)
#define __NR_sys_kexec_load     283
#define __NR_waitid             284
/* #define __NR_sys_setaltroot  285 */
#define __NR_add_key            286
#define __NR_request_key        287
#define __NR_keyctl             288
```

---

```
#define NR_syscalls      289
```

此处的代码是从 2.6.11 中的代码，其中系统调用号已到了 288，并且与前面 `system_call` 中的相对应。每一个系统调用号前都是相应函数名加了 `__NR_`。

内核跟用户程序的交互，其实有标准 C 库作为它们之间的桥梁。标准 C 库把用户希望传递的参数装载到 CPU 的寄存器中，然后触发 0X80 中断。当从系统调用返回的时候 (`sys_call_exit`)，标准 C 库又接过控制权，处理返回值。

对于 `__NR_`，标准 C 库会作相应处理。转换成相应函数。对于系统函数的调用，有几个通用的宏在 `include/asm-i386/unistd.h` 中定义。

```
#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-(128 + 1))) { \
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)

#else

# define __syscall_return(type, res) return (type) (res)

#endif


#define __syscall0(type,name) \
type name(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_###name)); \
    __syscall_return(type,__res); \
}
```

---

```
}
```

这是无参函数调用的形式。

```
#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_###name),"b" ((long)(arg1))); \
__syscall_return(type,__res); \
}
```

这是含一个参数的调用形式,

.....

标准 C 库会把我们的调用如 pause()转换成相应的形式。

```
pause()
int pause(void)
{
long __res;
__asm__ volatile ("int $0x80"
: "=a" (__res)
: "" (__NR_pause));
__syscall_return(int,__res);
}
```

进入内核调用过程。

---

## 2.2. 添加新的系统调用 (2.6.18 内核)

### 1. 步骤 1: 准备

如果你安装的系统包含内核源文件，一般在/usr/src 路径下可以看到，那么可以直接跳到步骤 3 进行内核修改。

为避免编译失败造成无法进入系统，最好先修改/usr/src/linux 下的 Makefile 文件，将内核版本修改成自己的。

2.6.18 中 Makefile 文件头几行为：

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 18
EXTRAVERSION = .1
```

我们可以修改成自己版本 (2.6.18.1 - 1)：

```
VERSION = 2
PATCHLEVEL = 6
SUBLEVEL = 18
EXTRAVERSION = .1 - 1
```

### 2. 步骤 2: 下载源代码

如果系统不包含源文件，则需要在网站上下载系统源代码，(大概有四五十兆)。

网址：<http://kernel.org>

在官方网站上下到类似 linux-2.6.18.1.tar.gz 的代码后 (大概有四，五十兆)，放在/usr/src/ 的目录下，

然后解压：[root@localhost linux-2.6.18.1]# tar -xvf linux-(按 tab 键)

解压后会出现文件夹 linux - 2.6.18.1。

### 3. 步骤 3: 修改相应内核文件

#### (1) 修改 (添加) 源代码

---

第一个任务是编写加到内核中的源程序,即将要加到一个内核文件中去的一个函数,该函数的名称应该是新的系统调用名称前面加上 sys\_ 标志。假设新加的系统调用为 mycall(int number), 在 `/usr/src/linux—2.6.18.2/kernel/sys.c` 文件中添加源代码, 如下所示:

```
asmlinkage int sys_mycall(int number)
{
    return number;
}
```

作为一个最简单的例子, 我们新加的系统调用仅仅返回一个整型值。

**说明: 在以后的文档中, 我们所指的 `/usr/src/linux—2.6.18.2/` 都是指内核路径, 要根据自己的内核存放位置进行相应的改变。**

## (2) 连接新的系统调用

添加新的系统调用后, 下一个任务是使 Linux 内核的其余部分知道该程序的存在。为了从已有的内核程序中增加到新的函数的连接, 需要编辑两个文件。

在我们所用的 Linux 内核版本 (2.6.18) 中, 首先要修改的文件是 `unistd.h` (一般情况下有两个 `unistd.h` 需要修改):

`/usr/src/linux—2.6.18.2/include/asm-i386/unistd.h`

`/usr/include/asm/unistd.h`

该文件中包含了系统调用清单, 用来给每个系统调用分配一个唯一的号码。文件中每一行的格式如下:

```
#define __NR_name NNN
```

其中, name 用系统调用名称代替, 而 NNN 则是该系统调用对应的号码。应该将新的系统调用名称加到清单的最后, 并给它分配号码序列中下一个可用的系统调用号。我们的系统调用如下:

```
#define __NR_sync_file_vange 314
```

```
#define __NR_tee 315
```

```
#define __NR_vmsplice 316
```

```
#define NR_syscalls 317
```

---

最后一行说明内核自身的系统调用号已经使用到 316，共有 317 个（从 0 开始）。我们新添加的系统调用应该加到最后，并修改系统调用总数。修改如下：

```
#define __NR_sync_file_vange 314
#define __NR_tee 315
#define __NR_vmsplice 316
#define __NR_mycall 317 /*这是我们自己添加的系统调用*/

#define NR_syscalls 318
```

第二个要修改的文件是：

**/usr/src/linux—2.6.18.2/arch/i386/kernel/syscall\_table.S**

在 2.6 版本以前，需要修改的是 **/usr/src/linux/arch/i386/kernel/entry.s**，在该文件下有 **sys\_call\_table**，直接进行修改。

在 2.6 版本后，**entry.s** 文件中的 **sys\_call\_table** 独立出来，我们可以在该文件下看到类似于 **#include"syscall\_table.s"** 字样，说明我们正真需要修改 **syscall\_table.s**。在 **syscall\_table.s** 中有类似如下的清单：

**.long sys\_name**

该清单用来对 **sys\_call\_table[]** 数组进行初始化。该数组包含指向内核中每个系统调用的指针。这样就在数组中增加了新的**内核函数的指针**。我们在清单上与**系统调用号****相对应的位置**添加一行：

**.long sys\_mycall**

必须注意添加的行的位置，否则容易造成内核编译的失败。

#### 4. 步骤 4：开始对新的内核进行编译

在 2.6 内核之前，编译内核需要如下命令：

```
make mrproper
make dep
make clean
```



---

```
make bzImage  
make modules  
make modules_install
```

然后将新生成的 System.map 和 bzImage 拷贝到/boot 下，并建立连接。

在 2.6 中，这些已经得到简化，只要以下命令即可：

```
make mrproper  
make xconfig (可视化选项) (http://www.chinaunix.net/jh/4/16106.html)  
make  
make modules_install  
make install
```

说明：

其中 make mrproper 命令用来清除旧的配置等文件，避免编译内核时生成的文件不一致

make xconfig 用于选择内核编译配置，事实上还有 config,gconfig,menuconfig 等选项可以选用。这一步选择很重要，必须选对，编译内核时才能得到正确结果。对于 2.6.18.2 版本来说，必须要注意的是选中 ext3 项。

【配置内核 配置内核的选项有 N 多，为了避免繁琐的配置，可以在 /boot 的目录下找到一个类似 config-2.6.18-1.2798.fc 的配置文件，将他 copy 到 /usr/src/linux-2.6.18.1/下，并改名 config;

```
[root@localhost linux-2.6.18.1]# make menuconfig
```

用户可以直接保存退出。】

make 编译新内核(完成了以前的 make bzImage 和 make modules 命令)

make modules\_install 模块命令

make install 安装新内核（完成了以前需要手工进行的后处理工作）

如果执行成功，则在/boot/grub 中的 grub.conf 中会出现新内核的选项。

至此，新内核已经建立，新添加的系统调用已经成为操作系统的一部分，重启后选用新内核即可测试该新的系统调用。

---

## 2.3. 对新加的系统调用进行测试

### 1. 旧版本的测试方法

在应用程序中使用新添加的系统调用 `mycall`,为了测试,我们编写一个简单的例子 `test.c`.

```
/* test.c */

#include<linux/unistd.h>

_syscall2(int,mycall,int,ret) //系统调用函数声明

main
{
    printf("%d \n",mycall(100));    //100
}
```

在当前路径下编译执行该程序。

注意: 由于要编译系统源程序以及使用系统调用, 以上操作都应该在 **root** 用户下进行。

### 2. 新版本的测试方法

在 2.6.18 之后 `_syscallN` (0~7) 的定义与以前的版本不太一样, 不再使用 `_syscallN` 宏了。 `syscall` 调用接口从 2.6.18 开始移到应用层, 原来内核中使用 `_syscallN` 宏的方式来声明函数原型的方法不再有效:

如声明:

```
_syscall1(int, sysinfo, struct sysinfo *, info);
```

不再需要, 而是在程序中需要的时候直接调用:

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

\* 第一个 `number` 是后面要接的参数个数, 不是该系统调用的参数个数;

\* `number` 后面顺序接上该系统调用的所有参数即可

---

## 2.4. 添加新的系统调用 (3.2.4 内核)

### 1. 步骤 1: 准备

下载内核源代码，下载网址 <http://kernel.org>，直接选 linux-3.2.4 内核版本下载。下载后文件在 download 文件夹下面，下载的是 bz2 压缩包。然后后打开终端 (ctrl+alt+t)，输入命令：sudo tar xvjf ./download/linux-3.2.4.tar.bz2 -C/usr/src/，此命令将 linux-3.2.4 内核文件解压到 /usr/src/ 文件夹下面，会看到里面多了个 linux-3.2.4 文件夹，解压成功。

### 2. 步骤 2: 修改相应内核文件

由于以下修改的内核文件都是只读文件，需要 root 登录按相应路径找到才能将其改写。

root 登录方法:

打开终端：输入命令 sudo passwd root

按提示输入 sudo 密码

设置 root 密码

置成功后切换用户，选择其他用户登录，用户名为 root，密码为刚设的密码。

#### (1) 修改 (添加) 源代码

第一个任务是编写加到内核中的源程序，即将要加到一个内核文件中去的一个函数，该函数的名称应该是新的系统调用名称前面加上 sys\_ 标志。假设新加的系统调用为 mycall(int number)，在 /usr/src/linux-3.2.4/kernel//sys.c 文件中添加源代码，如下所示：

```
asmlinkage int sys_mycall(int number)
{
    return number;
}
```

作为一个最简单的例子，我们新加的系统调用仅仅返回一个整型值。

#### (2) 连接新的系统调用

---

添加新的系统调用后，下一个任务是使 Linux 内核的其余部分知道该程序的存在。为了从已有的内核程序中增加到新的函数的连接，需要编辑两个文件。

在我们所用的 Linux 内核版本（3.2.4）中，首先要修改的文件是 `/usr/src/linux-3.2.4/arch/x86/include/asm/unistd_32.h`。

该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码。文件中每一行的格式如下：

```
#define __NR_name NNN
```

其中，`name` 用系统调用名称代替，而 `NNN` 则是该系统调用对应的号码。应该将新的系统调用名称加到清单的最后，并给它分配号码序列中下一个可用的系统调用号。我们的系统调用如下：

```
#define __NR_sendmmsg      345
```

```
#define __NR_setns        346
```

```
#define __NR_process_vm_readv 347
```

```
#define __NR_process_vm_writev 348
```

```
#define NR_syscalls 349
```

最后一行说明内核自身的系统调用号已经使用到 348，共有 349 个（从 0 开始）。我们新添加的系统调用应该加到最后，并修改系统调用总数。修改如下：

```
#define __NR_sendmmsg      345
```

```
#define __NR_setns        346
```

```
#define __NR_process_vm_readv 347
```

```
#define __NR_process_vm_writev 348
```

```
#define __NR_mycall      349      /*这是我们自己添加的系统调用*/
```

---

```
#define NR_syscalls    350
```

第二个要修改的文件是:

```
/usr/src/linux-3.2.4/arch/x86/kernel/syscall_table_32.s
```

在 syscall\_table.s 中有类似如下的清单:

```
.long sys_name
```

该清单用来对 sys\_call\_table[]数组进行初始化。该数组包含指向内核中每个系统调用的指针。这样就在数组中增加了新的内核函数的指针。我们在清单上与系统调用号相对应的位置添加一行:

```
.long sys_mycall
```

必须注意添加的行的位置, 否则容易造成内核编译的失败。

## 2. 步骤 4: 开始对新的内核进行编译

先切换目录: `cd /usr/src/linux-3.2.4/`

命令: `sudo make mrproper`

`sudo make menuconfig` (第一次这步可能执行出错, 需要安装什么包如下命令安装: `sudo apt-get install libncurses5-dev`, 安装好后, 再 `make menuconfig`), 为了避免选择的麻烦, 直接按下键一直到最下面, 选中 `save`---回车就可以了, 退出到终端。

`sudo make` 简单的命令 (此步由 `make bzImage` 和 `make modules` 两步组成, 两步操作都要等很长时间, 还不如等一步, 之后就是让机器慢慢的编译内核去了)

```
sudo make modules_install
```

```
sudo make install
```

```
sudo update-grub
```

编译好了, 可以直接重启计算机了, 可以看到开机启动项里多了 `linux-3.2.4` 的选项, 选择进入此系统

说明:

---

其中 `make mrproper` 命令用来清除旧的配置等文件, 避免编译内核时生成的文件不一致;

`make xconfig` 用于选择内核编译配置, 事实上还有 `config`, `gconfig`, `menuconfig` 等选项可以选用。

## 2.5. 对新加的系统调用进行测试 (3.2.4 内核)

### 1. 测试方法

在应用程序中使用新添加的系统调用 `mycall`, 为了测试, 我们编写一个简单的例子 `test.c`.

```
/* test.c */
#include<stdio.h>
#include</usr/src/linux-3.2.4/arch/x86/include/asm//unistd_32.h>
#include<errno.h>
#include<sys/syscall.h>

int main(int argc, char **argv)
{
    int b=syscall(349,200); /*第一个参数系统调用号, 第二个参数, 给的任意数值
参数*/
    printf("%d\n",b);
    return 0;
}
```

在当前路径下编译执行该程序。

注意: `syscall()` 中的第一个参数 349 是系统调用号, 第二个参数 200 个是函数的参数。

---

## 第3章 设备驱动相关知识

### 3.1. 基础知识

系统调用是操作系统内核和应用程序之间的接口，而**设备驱动程序是操作系统内核和机器硬件之间的接口**。设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作。设备驱动程序是内核的一部分，它完成以下的功能：

- (1) **对设备初始化和释放。**
- (2) **把数据从内核传送到硬件和从硬件读取数据。**
- (3) **读取应用程序传送给设备文件的数据和回送应用程序请求的数据。**
- (4) **检测和处理设备出现的错误。**

Linux 支持三中不同类型的设备：字符设备 (character devices)、块设备 (block devices) 和网络设备 (network interfaces)。**字符设备和块设备的主要区别是**：在对字符设备发出读/写请求时，实际的硬件 I/O 一般就紧接着发生了，**块设备则不然，它利用一块系统内存作缓冲区**，当用户进程对设备请求能满足用户的要求，就返回请求的数据，如果不能，就调用请求函数来进行实际的 I/O 操作。块设备是主要针对磁盘等慢速设备设计的，以免耗费过多的 CPU 时间来等待。

用户进程是通过设备文件来与实际的硬件打交道，每个设备文件都都有其文件属性(c/b)，表示是字符设备还是块设备。另外每个文件都有两个设备号，第一个是主设备号，标识驱动程序，第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备，比如有两个软盘，就可以用从设备号来区分他们。设备文件的的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

**设备驱动程序工作的基本原理：**

用户进程利用系统调用对设备进行诸如 read/write 操作，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着

---

把控制权交给该函数。

最后，在用户进程调用驱动程序时，系统进入核心态，这时不再是抢先式调度。也就是说，系统必须在你的驱动程序的子函数返回后才能进行其他的工作。如果你的驱动程序陷入死循环，你只有重新启动机器了。

### 3.2. 添加新模块的基本步骤

#### 1. 写设备驱动源代码:

在设备驱动程序中有一个非常重要的结构 `file_operations`, 该结构的每个域都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如 `read/write` 操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

```
struct file_operations {
    int (*seek) (struct inode * , struct file * , off_t , int);
    int (*read) (struct inode * , struct file * , char , int);
    int (*write) (struct inode * , struct file * , off_t , int);
    int (*readdir) (struct inode * , struct file * , struct dirent * , int);
    int (*select) (struct inode * , struct file * , int , select_table *);
    int (*ioctl) (struct inode * , struct file * , unsigned int , unsigned long);
    int (*mmap) (struct inode * , struct file * , struct vm_area_struct *);
    int (*open) (struct inode * , struct file *);
    int (*release) (struct inode * , struct file *);
    int (*fsync) (struct inode * , struct file *);
    int (*fasync) (struct inode * , struct file * , int);
    int (*check_media_change) (struct inode * , struct file *);
    int (*revalidate) (dev_t dev);
}
```

编写设备驱动程序的主要工作是编写子函数，并填充 `file_operations` 的各个域。

例如:



---

```

Struct file_operations my_fops={
    .read=my_read,
    .write=my_write,
    .open=my_open,
    .release=my_release
}

```

然后再定义函数 my\_read,my\_write,my\_open,my\_release 相应的函数体。

例如:

```

static ssize_t my_open(struct inode *inode,struct file *file){
    static int counter=0;
    if(Device_Open)
        return -EBUSY;
    Device_Open++;
    /*写入设备的信息*/
    sprintf(msg,"the device has been called %d times\n",counter++);
    msg_ptr=msg;
    return 0;
}

```

同时对于可卸载的内核模块 (LKM) ,至少还有两个基本的模块:

例如本例中的:

```

static int __init my_init(void){
    int result;
    result=register_chrdev(0,"sky_driver",&my_fops);
    if(result<0){
        printk("error:can not register the device\n");
        return -1;
    }
    if(my_major==0){
        my_major=result;
    }
}

```

---

```

        printk("<1>hehe,the device has been registered!\n");
        printk("<1>the virtual device was assigned major number %d.\n",my_major);
        printk("<1>To talk to the driver,create a dev file with\n");
        printk("<1>'mknod/dev/my c %d 0\n",my_major);
        printk("<1>Remove the dev and the file when done\n");
    }
    return 0;
}

```

```

static void __exit my_exit(void){
    unregister_chrdev(my_major,"sky_driver");
    printk("<1>unloading the device\n");
}

```

my\_init 用于注册设备，获得设备的主设备号

调用 register\_chrdev(0,"sky\_driver(设备名)", &my\_fops);

my\_exit 用于注销设备

调用 unregister\_chrdev(my\_major, "sky\_driver(设备名)");

然后在程序尾再调用这两个函数

```

Module_init(my_init);
Module_exit(my_exit)
MODULE_LICENSE("GPL");

```

## 2. 编译

(1) 将设备驱动源文件复制到 /usr/src/linux/drivers/misc 下（这里的 /usr/src/linux 指的是源代码路径）

(2) 修改 Makefile,只要一句即可:obj-m +=sky\_driver.o

(3) 编译

---

在/usr/src/linux/drivers/misc 路径下执行

Make -C /usr/src/linux SUBDIRS=\$PWD modules

如果编译成功将得到.ko 文件

注意：此步编译过程中必须没有错误或者任何的警告，否则必须对错误和警告信息提示的位置进行修改，然后重新编译，直到没有任何错误或者警告（warning）为止。

### 3. 挂载内核中模块

命令：insmod ./sky\_driver.ko

此时 cat /proc/devices 或 dmesg 会看到在字符设备中有 254 sky\_driver。前面的是系统分配的主设备号，后面是设备注册名。

在 my\_init 函数中注册新的字符设备，所用系统调用是

```
Register_chrdev(0,"sky_driver",&my_fops);
```

函数中第一个参数是告诉系统，新注册的设备的主设备号由系统分配，

第二个参数是新设备注册时的设备名字，

第三个参数是指向 file\_operations 的指针，

执行该命令后调用 dmesg|tail 会看到在 my\_init 中写到的提示信息。

#### insmod 时出现 "Invalid module format" 的可能原因：

如果你 modprobe 自己编译的内核模块时出现：Invalid module format，那很有可能是以下原因引起的：

所用内核源码版本号与目前使用的内核不同；

编译目标不同，比如编译的是 i686，装好的是 i386；

使用编译器版本不同；

目前使用的内核不是自己编译出来的。

参考文献：<http://blog.csdn.net/hecant/archive/2007/10/31/1859606.aspx>

---

#### 4. 创建新的虚拟设备文件

命令: `mknod /dev/sky_driver C 254 0`

在此命令中, 第一个参数是新建设备文件的地址和名字, 第二个参数是指创建的是字符设备文件, 第三个参数是主设备号, 第四个参数是从设备号。

执行成功会在/dev 中看到一个新的设备文件 sky\_driver.

#### 5. 测试新的设备驱动

编写测试程序

#### 6. 卸载操作

删除模块

命令: `rmmod sky_driver`

删除新增的字符设备文件

命令: `rm /dev/sky_driver`

本部分参考文献:

<http://blog.csdn.net/sabalol/archive/2008/02/01/2076610.aspx>

<http://www.knowsky.com/340884.html>

## 第 4 章 /proc 文件相关知识

### 4.1. 实现原理

用户和应用程序可以通过 /proc 得到系统的信息，并可以改变内核的某些参数。由于系统的信息是动态改变的，所以用户或应用程序读取 proc 文件时，proc 文件系统是动态从系统内核读出所需信息并提交的。

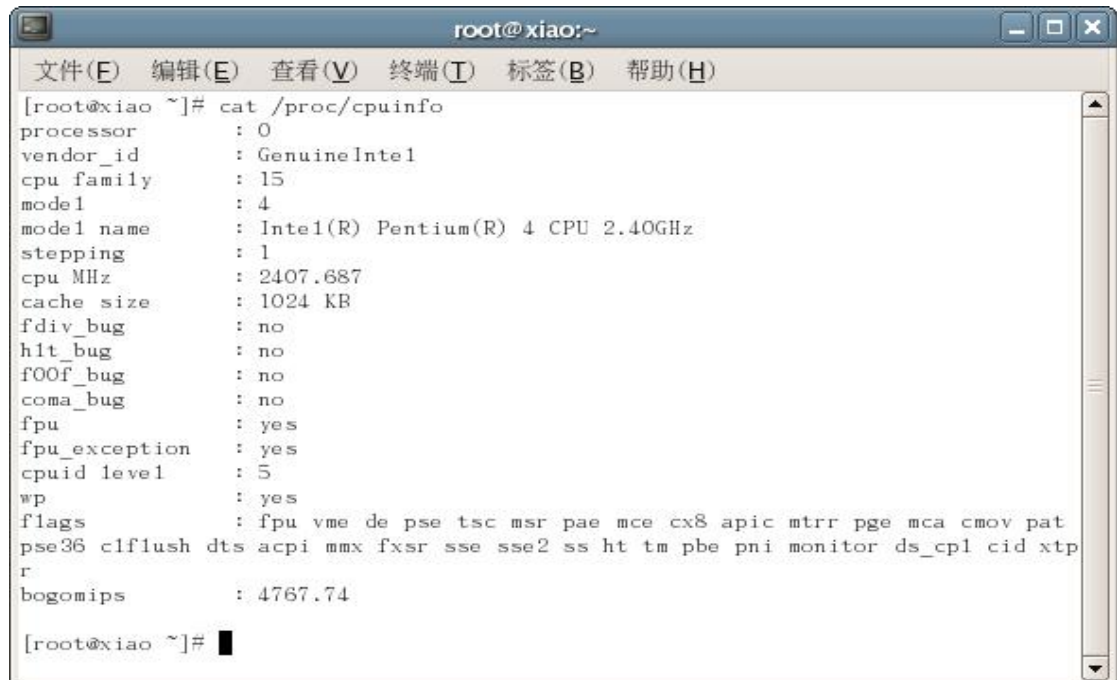
我们要显示系统信息，只需进行相应的文件操作。首先打开相应的文件，读取所需要的信息，将其写入一个缓冲区中，然后将缓冲区的内容加到 Gtk 的相应的控件上面去，最后将控件组合显示即可。

### 4.2. 实现方法

由于要以图形界面来实现,要用到 Linux 的图形库 Qt 或者是 Gtk,本例中使用 QT.

#### (1) CPU 信息:

使用 cat /proc/cpuinfo 你可以查看系统的内存信息



```
root@xiao:~# cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 15
model          : 4
model name     : Intel(R) Pentium(R) 4 CPU 2.40GHz
stepping       : 1
cpu MHz        : 2407.687
cache size     : 1024 KB
fdiv_bug       : no
hlt_bug        : no
f00f_bug       : no
coma_bug       : no
fpu            : yes
fpu_exception  : yes
cpuid level    : 5
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe pn1 monitor ds_cpl cid xtp
r
bogomips       : 4767.74

root@xiao:~#
```

由上图可知所需的信息所在文件中的位置，然后将其写入缓冲区，加载到

QT 相应的控件上:

```
QFile f("/proc/cpuinfo");  
f.open(IO_ReadOnly);/*打开相应的文件*/
```

得到 CPU 的类型: (GenuinoIntel):

```
QString *temp=new QString(buf);  
int from=temp->find("vendor_id",0);  
int end=temp->find("cpu family",0);  
QString *cpu_id=new QString(temp->mid(from+12,end-from-12));
```

同理, 可得到 cpu 的名称 (model name), 主频 (cpu MHZ), Cache 的大小 (cache size)

然后将其加载到相应的控件上:

```
cpuInfoLabel->setText(*cpu_id);  
cpunameLabel->setText(*cpu_name);  
frqLabel->setText(*cpu_mhz+" MHz");  
cacheLabel->setText(*cpu_cache);
```

## (2) 操作系统信息:

使用 cat /proc/version 可查看信息如下:



相应的代码如下:

```
void MainForm::osinfo()  
{  
    int i;  
    char buf[400];  
  
    QFile f("/proc/version");
```

---

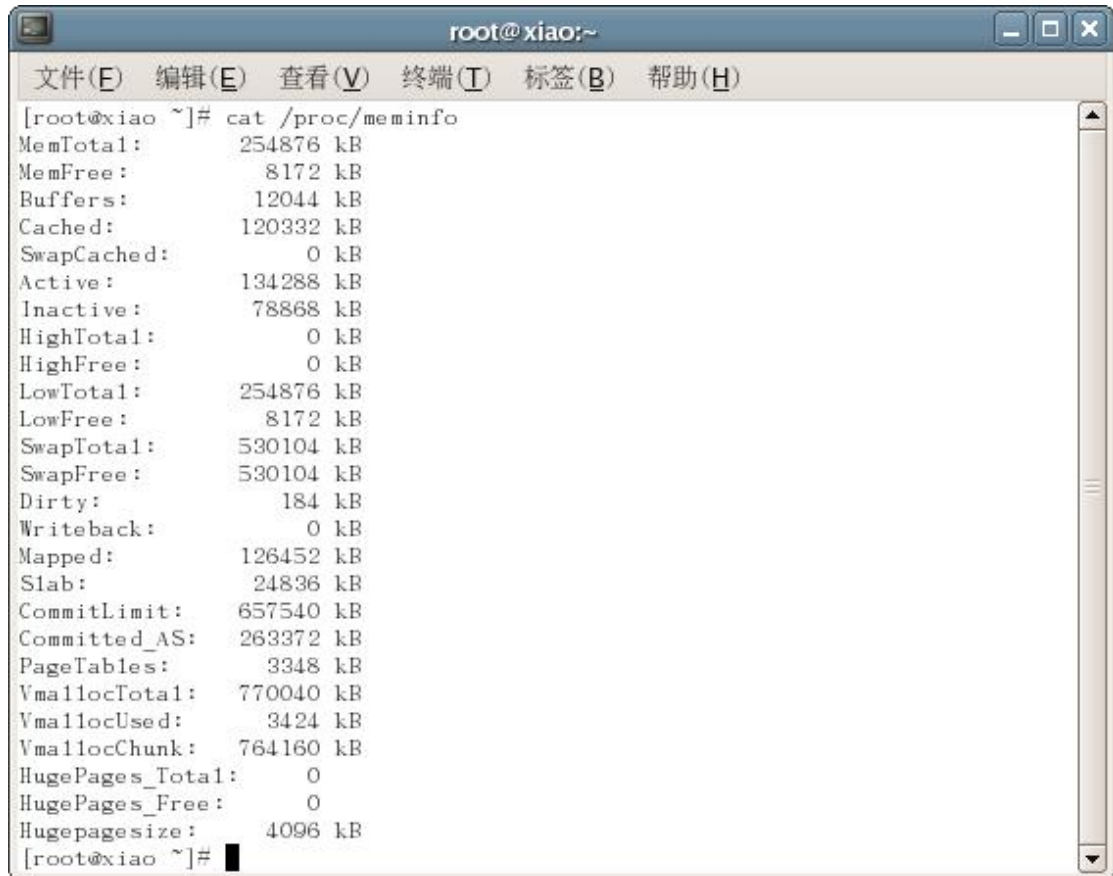
```

        f.open(IO_ReadOnly);
        for(i=0;i<400;i++){
            buf[i]=f.getch();
        }
    ////////////////得到 linux 版本信息
        QString *temp=new QString(buf);
        int from=temp->find("Linux version",0);
        int end=temp->find("(",0);
        QString *linux_version=new QString(temp->mid(from+14,end-from-15));
    ////////////////得到 gcc 版本信息
        from=temp->find("gcc",0);
        end=temp->find("#",0);
        QString *gcc_version=new QString(temp->mid(from+11,end-from-13));
    /*将信息加载到控件*/
        osversionLabel->setText(*linux_version);
        gccLabel->setText(*gcc_version);
    }

```

### (3) 内存信息:

使用 cat /proc/meminfo 可查看信息如下:

A terminal window titled 'root@xiao:~' with a menu bar containing '文件(E)', '编辑(E)', '查看(V)', '终端(T)', '标签(B)', and '帮助(H)'. The terminal displays the output of the command 'cat /proc/meminfo'. The output lists various memory statistics in kB, including MemTotal, MemFree, Buffers, Cached, SwapCached, Active, Inactive, HighTotal, HighFree, LowTotal, LowFree, SwapTotal, SwapFree, Dirty, Writeback, Mapped, Slab, CommitLimit, Committed\_AS, PageTables, VmallocTotal, VmallocUsed, VmallocChunk, HugePages\_Total, HugePages\_Free, and Hugepagesize. The prompt '[root@xiao ~]#' is visible at the bottom.

```
[root@xiao ~]# cat /proc/meminfo
MemTotal:      254876 kB
MemFree:       8172 kB
Buffers:       12044 kB
Cached:        120332 kB
SwapCached:    0 kB
Active:        134288 kB
Inactive:      78868 kB
HighTotal:     0 kB
HighFree:      0 kB
LowTotal:      254876 kB
LowFree:       8172 kB
SwapTotal:     530104 kB
SwapFree:      530104 kB
Dirty:         184 kB
Writeback:     0 kB
Mapped:        126452 kB
Slab:          24836 kB
CommitLimit:   657540 kB
Committed_AS:  263372 kB
PageTables:    3348 kB
VmallocTotal:  770040 kB
VmallocUsed:    3424 kB
VmallocChunk:  764160 kB
HugePages_Total: 0
HugePages_Free: 0
Hugepagesize:  4096 kB
[root@xiao ~]#
```

同理，可读取 proc 文件信息存入缓冲区：

Memtotal:254876KB      Memfree: 8172KB

由此可计算出主存已用空间和使用率

SwapTotal:530104kb      SwapFree:530104kb

由此可计算交换区已用空间和使用率

而这些信息都是不断变化的,所有设置定时器,不断地从文件中读取新的信息.

显示在界面上:

```
unused_mm_label->setText(*mem_free+"MB");
all_mm_label->setText(*mem_total+"MB");
all_swap->setText(*swap_total+"MB");
free_swap->setText(*swap_free+"MB");

int swap_use,mem_use;
swap_use=mem_total->toInt()-mem_free->toInt();
mem_use=swap_total->toInt()-swap_free->toInt();
```



```

        textLabel17->setText(QString::number(swap_use,10)+"MB");
        textLabel18->setText(QString::number(mem_use,10)+"MB");

    int mem_useage,swap_useage;

    bool okok;

    mem_useage=100-mem_free->toInt(&okok,10)*100/mem_total->toInt(&okok,10);

    progressBar1->setProgress(mem_useage);

    swap_useage=100-swap_free->toInt(&okok,10)*100/swap_total->toInt(&okok,10);

    progressBar2->setProgress(swap_useage);

}

```

#### (4) 进程信息:

当你进入/proc 目录时,你会发现很多以十进制数为标题的目录,它们都是记录系统中正在运行的每个用户级进程的信息,数字表示进程号(pid)./proc/pid/stat 目录下存储了该进程的相关信息。例如可用 cat 命令查看信息如下:



```

root@xiao:~
文件(E) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[root@xiao ~]# cat /proc/1/stat
1 (init) S 0 0 0 0 -1 8388864 5049 287697 93 691 28 86 648 270 16 0 1 0 138 1785
856 142 4294967 295 134512640 134540244 3221066320 0 582 0 0 1475401980 671819267
0 0 0 0 0 0 0
[root@xiao ~]#

```

其中,1 为进程的 pid, init 为进程名, S 表明为睡眠进程, 16 表示进程的优先级. 1785856 为进程占用的内存大小, 即  $1785856\text{B}/1024=1748\text{KB}$ .

要获得进程信息, 首先应该获得各进程的 pid, 将其存在一个 qlist 的缓冲区中:

```

QDir qd("/proc");
QStringList qlist=qd.entryList();
QString qs=qlist.join("\n");

```

---

再从该缓冲区中依次读出一个 pid,

```
a=qs.find("\n",find_start);
```

```
b=qs.find("\n",a+1);
```

```
find_start=b;
```

```
QString name_of_pro=qs.mid(a+1,b-a-1);
```

从/proc/pid/stat 目录下的进程信息读入缓冲区 buf 中去, 然后使用 find 在 buf 中获取需要显示的信息, 如进程名, 进程状态, 优先级, 占用内存大小等。

```
//get process name
```

```
QString *temp=new QString(buf);
```

```
QString temp_proc_name=temp->section(' ',1,1);
```

```
    QString temp_proc_name2
```

```
=temp_proc_name.mid(1,temp_proc_name.length()-2);
```

```
//get process state
```

```
QString temp_proc_state=temp->section(' ',2,2);
```

```
switch(temp_proc_state.at(0).latin1())
```

```
{
```

```
case 'S':number_of_sleep++;break;
```

```
case 'Z':number_of_zombie++;break;
```

```
case 'R':number_of_run++;break;
```

```
default:break;
```

```
}
```

```
//get process priority
```

```
QString temp_proc_priority=temp->section(' ',17,17);
```

```
//get process memory use
```

```
QString temp_proc_mem=temp->section(' ',22,22);
```

```
int ttt=temp_proc_mem.toInt()/1024;
```

最后将该进程的信息在界面上显示出来, 然后读取下一个进程的信息:

```
QListViewItem
```

```
abc=new
```

```

QListViewItem(listView2,name_of_pro,temp_proc_name2,
temp_proc_state,temp_proc_priority,QString::number(ttt,10)+" KB");
listView2->insertItem(&abc);

```

#### (5) 模块信息:

与获取进程信息的方式类似，模块信息存储在目录/proc/modules 下面，使用 cat 命令可查看信息如下：



```

root@xiao:~
文件(E) 编辑(E) 查看(V) 终端(T) 标签(B) 帮助(H)
[root@xiao ~]# cat /proc/modules
parport_pc 28933 1 - Live 0xd0af1000
lp 13001 0 - Live 0xd0ab7000
parport 40585 2 parport_pc,lp, Live 0xd09dd000
autofs4 29253 2 - Live 0xd0aae000
i2c_dev 11073 0 - Live 0xd09bb000
i2c_core 21569 1 i2c_dev, Live 0xd0a94000
rfcomm 42333 0 - Live 0xd0aa2000
l2cap 30661 5 rfcomm, Live 0xd0a8b000
bluetooth 56133 4 rfcomm,l2cap, Live 0xd09f4000
sunrpc 167813 1 - Live 0xd0abc000
ipt_REJECT 5569 1 - Live 0xd09da000
ipt_state 1857 1 - Live 0xd09c5000
ip_conntrack 41497 1 ipt_state, Live 0xd09e8000
iptable_filter 2881 1 - Live 0xd0822000
ip_tables 19521 3 ipt_REJECT,ipt_state,iptable_filter, Live 0xd09bf000
dm_mod 58101 0 - Live 0xd09ca000
video 15941 0 - Live 0xd091c000
button 6609 0 - Live 0xd08fe000
battery 9413 0 - Live 0xd0921000
ac 4805 0 - Live 0xd0903000
md5 4033 1 - Live 0xd0901000

```

其中,每一行为一个模块的信息,第一项为模块名,第二项为模块占用的内存大小,第三项为模块使用的次数。

也是将其内容读入缓冲区,使用 find 获得需要的信息 (如模块名, 占用内存大小, 使用次数), 然后创建一个条目在界面上显示出来。相关程序如下:

```

void mainForm::getmoduleinfo()
{
listView1->clear();

int i,j,k;

int number_of_modules=0;

char temp[20];

```

---

```
char temp2[20];
char temp3[20];
j=0;

char buf[20000];
for(i=0;i<20000;i++) buf[i]='\0';

t3.start(15000);

QFile f("/proc/modules");
f.open(IO_ReadOnly);

i=0;
while((buf[i]=f.getch())!=-1) i++;
for(i=0;i<20000;i++){
if (buf[i]=='\n') number_of_modules++;
}

for(i=1;i<number_of_modules;i++){
//////////get module name
for (k=0;k<20;k++) temp[k]='\0';
k=0;
while(buf[j]!=' '){
    temp[k++]=buf[j];
    j++;
}
j++;
//////////get module memory use
for (k=0;k<20;k++) temp2[k]='\0';
k=0;
```

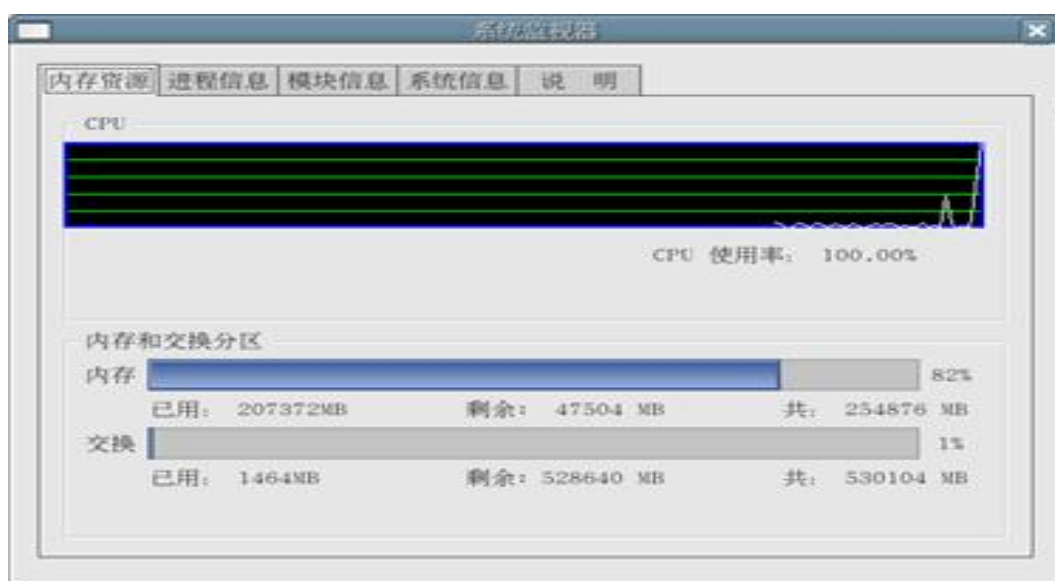
---

```
while(buf[j]!=' '){
    temp2[k++]=buf[j];
    j++;
}
j++;
//////////get nodule use time
for (k=0;k<20;k++) temp3[k]='\0';
k=0;
while(buf[j]!=' '){
    temp3[k++]=buf[j];
    j++;
}j++;

while(buf[j]!='\n') j++;
QListViewItem abc=new QListViewItem(listView1,QString(&temp[1]),
QString(temp2),QString(temp3));
listView1->insertItem(&abc);
} //for
}
```

其他功能如关机、重启等可自行查看相关资料，qt 的相关知识可查看其帮助文件。

最后实现的结果如下：



系统监视器

内存资源 进程信息 模块信息 系统信息 说明

PID	名称	状态	优先级	占用内存	
1	init	S	16	1748 KB	
135	pdflush	S	15	0 KB	
136	pdflush	S	15	0 KB	
137	kswapd0	S	16	0 KB	
138	aio/0	S	15	0 KB	
1388	scsi_eh_0	S	20	0 KB	
1389	usb-storage	S	15	0 KB	
1837	syslogd	S	16	1620 KB	
1839	klogd	S	16	1568 KB	
1849	portmap	S	15	1700 KB	
1867	rpc.statd	S	18	1744 KB	
1881	auditd	S	13	11912 KB	
1885	kauditd	S	10	0 KB	
1910	rpc.idmapd	S	16	4380 KB	
1923	hcid	S	18	2160 KB	

总进程数: 104

运行进程: 3

睡眠进程: 101

僵死进程: 0

结束进程(C) 刷新(H)



