第四篇　Linux系统高级编程

第18章　Linux内核模块编程

本章从一个简单的程序Hello World开始介绍Linux内核模块的编程。

18.1　一个简单程序Hello World

一个内核模块至少包括两个函数：一个是初始化函数 init_module，在模块被插入到内核时使用；另一个是清除函数 cleanup_module，在模块被从内核中移走时使用。一般情况下，函数init_module或者在内核中注册一个处理程序，或者使用自己的代码替换一个内核函数。函数cleanup_module的作用是清除init_module所作的任何事情，这样模块可以安全地卸载。

下面来看一个简单的程序。

```
/* hello.c */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Initialize the module */
int init_module()
{
  printk("Hello, world - this is the kernel speaking\n");
  /* If we return a non zero value, it means that init_module failed and
   * the kernel module can't be loaded */
  return 0;
}
/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
 # Makefile for a basic kernel module
CC=gcc
MODCFLAGS := -O6 -Wall -DCONFIG_KERNELD -DMODULE -D__KERNEL__ -DLinux

hello.o:        hello.c /usr/include/linux/version.h
                $(CC) $(MODCFLAGS) -c hello.c
                echo insmod hello.o to turn it on
                echo rmmod hello to turn if off
                echo
                echo X and kernel programming do not mix.
                echo Do the insmod and rmmod from outside X.
```

现在，你可以登录到超级用户下，然后执行 insmod hello 和rmmod hello。你可以在/proc/modules中查看新的内核模块。

18.2 设备文件

内核模块包括两种方法和一个进程通信。一种方法是使用设备文件，例如目录 /dev下的文件，另一种方法是使用文件系统。因为编写内核模块的一个主要原因就是为了驱动某种硬件设备，所以我们从设备文件开始。

设备文件的主要目的是允许进程和内核中的设备驱动程序通信，从而和物理设备通信。以下实现的方法。

每一个负责某硬件设备的设备驱动程序都有一个主设备号，设备驱动程序表和对应的主设备号可以在/proc/devices中查看到。每一个设备驱动程序控制的物理设备有一个从设备号。目录/dev下为每一个设备都对应了一个特殊的文件，叫做设备文件，而不管此设备是否安装到了系统中。

例如，如果你执行ls -l /dev/hd[ab]*，你将会看到机器中所有的 IDE硬盘的分区。请注意，这些硬盘分区的主设备号都是3，但从设备号却各不相同。

系统安装时，所有这些设备文件都是使用 mknod命令创建的。没有什么特别的原因将设备文件放在/dev目录下，这只是一种习惯。

设备可以分为两种类型：字符设备和块设备。通过使用 ls -1命令可以分辨设备是字符设备还是块设备。如果此命令输出的第一个字符是 b，那么它是一个块设备；如果第一个字符是 c，那么它是一个字符设备。

内核模块分为两个部分：用于注册设备的模块部分和设备驱动程序部分。函数 init_module通过调用module_register_chrdev来把设备驱动程序添加到内核的字符设备驱动表中，同时返回驱动程序可以使用的主设备号。函数cleanup_module用于清除注册的设备。

设备驱动程序由4个device_<action>函数组成。当某一个进程需要拥有这个主设备号的设备文件作某些工作时将会调用这些函数。内核通过数据结构 Fops可以了解调用它们的方法。该数据结构是在设备注册时给定的，它包含指向这4个函数的指针。

下面是chardev.c的源程序，该程序用来创建一个字符设备文件。

```
/* chardev.c
  * Create a character device (read only)
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* For character devices */
#include <linux/fs.h>       /* The character device definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does next to nothing at
                             * present, but may help for compatibility
                             * with future versions of Linux */
#define SUCCESS 0
/* Device Declarations ********************************************** */
/* The name for our device, as it will appear in /proc/devices */
#define DEVICE_NAME "char_dev"
/* The maximum length of the message from the device */
```

```
#define BUF_LEN 80
/* Is the device open right now? Used to prevent concurent access into
 * the same device */
static int Device_Open = 0;
/* The message the device will give when asked */
static char Message[BUF_LEN];
/* How far did the process reading the message get? Useful if the
 * message is larger than the size of the buffer we get to fill in
 * device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts to open the device
 * file */
static int device_open(struct inode *inode, struct file *file)
{
  static int counter = 0;
#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif
  /* We don't want to talk to two processes at the same time */
  if (Device_Open)
    return -EBUSY;
  /* If this was a process, we would have had to be more careful here.
   *
   * In the case of processes, the danger would be that one process
   * might have check Device_Open and then be replaced by the schedualer
   * by another process which runs this function. Then, when the first process
   * was back on the CPU, it would assume the device is still not open.
   * However, Linux guarantees that a process won't be replaced while it is
   * running in kernel context.
   *
   * In the case of SMP, one CPU might increment Device_Open while another
   * CPU is here, right after the check. However, in version 2.0 of the
   * kernel this is not a problem because there's a lock to guarantee
   * only one CPU will be kernel module at the same time. This is bad in
   * terms of performance, so it will probably be changed in the future,
   * but in a safe way.
   */
  Device_Open++;
  /* Initialize the message. */
  sprintf(Message,
    "If I told you once, I told you %d times - Hello, world\n",
    counter++);
  /* The only reason we're allowed to do this sprintf, is because the
   * maximum length of the message (assuming 32 bit integers - up to 10 digits
   * with the minus sign) is less than BUF_LEN, which is 80. BE CAREFUL NOT TO
   * OVERFLOW BUFFERS, ESPECIALLY IN THE KERNEL!!!
   */
  Message_Ptr = Message;
  /* Make sure that the module isn't removed while the file is open by
   * incrementing the usage count (the number of opened references to the
   * module, if it's not zero rmmod will fail)
```

```
   */
  MOD_INC_USE_COUNT;
  return SUCCESS;
}
/* This function is called when a process closes the device file. It
 * is not allowed to fail */
static void device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif
  /* We're now ready for our next caller */
  Device_Open --;
  /* Decrement the usage count, otherwise once you opened the file you'll
   * never get rid of the module.
   */
  MOD_DEC_USE_COUNT;
}
/* This function is called whenever a process which already opened the
 * device file attempts to read from it. */
static int device_read(struct inode *inode,
                       struct file *file,
                       char *buffer,   /* The buffer to fill with the data */
                       int length)     /* The length of the buffer
                                        * (mustn't write beyond that!) */
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;
#ifdef DEBUG
  printk("device_read(%p,%p,%p,%d)\n",
    inode, file, buffer, length);
#endif
  /* If we're at the end of the message, return 0 (which signifies end
   * of file) */
  if (*Message_Ptr == 0)
    return 0;
  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {
    /* Because the buffer is in the user data segment, not the kernel
     * data segment, assignment wouldn't work. Instead, we have to use
     * put_user which copies data from the kernel data segment to the user
     * data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }
#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
   /* Read functions are supposed to return the number of bytes actually
```

```
   * inserted into the buffer */
  return bytes_read;
}
/* This function is called when somebody tries to write into our device
 * file - currently unsupported */
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
{
#ifdef DEBUG
  printk ("device_write(%p,%p,%s,%d)",
    inode, file, buffer, length);
#endif
  return -EINVAL;
}
/* Module Declarations ************************************************ */
/* The major device number for the device. This is static because it
 * has to be accessible both for registration and for release. */
static int Major;
/* This structure will hold the functions to be called when
 * a process does something to the device we created. Since a pointer to
 * this structure is kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  NULL,   /* ioctl */
  NULL,   /* mmap */
  device_open,
  device_release  /* a.k.a. close */
};
/* Initialize the module - Register the character device */
int init_module()
{
  /* Register the character device (atleast try) */
  Major = module_register_chrdev(0,
                                 DEVICE_NAME,
                                 &Fops);
  /* Negative values signify an error */
  if (Major < 0) {
    printk ("Sorry, registering the character device failed with %d\n",
      Major);
    return Major;
  }
  printk ("Registeration is a success. The major device number is %d.\n",
    Major);
  printk ("If you want to talk to the device driver, you'll have to\n");
```

```
  printk ("create a device file. We suggest you use:\n");
  printk ("mknod <name> c %d 0\n", Major);
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(Major, DEVICE_NAME);
  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

18.3 /proc文件系统

在Linux系统中，系统内核和内核模块还有另一种向进程发送信息的方法——/proc文件系统。最初设计/proc文件系统的目的是为了方便地存取进程中的信息，而现在内核中所有希望发送信息的，例如保存模块表的/proc/modules和保存内存使用统计的/proc/meminfo都在使用/proc文件系统。

使用/proc文件系统的方法和使用设备驱动程序的方法十分相似。首先，创建一个包括/proc文件系统所需要的信息的数据结构，该数据结构包括指向处理程序的指针。然后，使用init_module在内核中注册此数据结构，要清除这些注册则使用 cleanup_module。

使用proc_register_dynamic的原因是我们并不希望事先决定文件的索引节点，而是允许内核来决定以避免冲突。

下面是程序procfs.c的源代码：

```
/* procfs.c -  create a "file" in /proc
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>
/* Put data into the proc fs file.
   Arguments
   =========
   1. The buffer where the data is to be inserted, if you decide to use
      it.
   2. A pointer to a pointer to characters. This is useful if you don't
      want to use the buffer allocated by the kernel.
   3. The current position in the file.
   4. The size of the buffer in the first argument.
   5. Zero (for future use?).
   Usage and Return Value
   ======================
   If you use your own buffer, like I do, put its location in the
   second argument and return the number of bytes used in the buffer.
```

```
   A return value of zero means you have no further information at this
   time (end of file). A negative return value is an error condition.
   For More Information
   ====================
   The way I discovered what to do with this function wasn't by reading
   documentation, but by reading the code which used it. I just looked to
   see what is using the get_info field of proc_dir_entry's (I used a
   combination of find and grep, if you're interested), and I saw that
   it is used in <kernel source directory>/fs/proc/array.c.
   If something is unknown about the kernel, this is usually the way
   to go. In Linux we have the great advantage of having the kernel source
   code for free - use it.
 */
int procfile_read(char *buffer, char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */
  /* This is static so it will still be in memory when we leave this
   * function */
  static char my_buffer[80];
  static int count = 1;
  /* We give all of our information in one go, so if the user asks us
   * if we have more information the answer should always be no.
   *
   * This is important because the standard read function from the library
   * would continue to issue the read system call until the kernel replies
   * that it has no more information, or until its buffer is filled.
   */
  if (offset > 0)
    return 0;
  /* Fill the buffer and get its length */
  len = sprintf(my_buffer, "For the %d%s time, go away!\n", count,
    (count % 100 > 10 && count % 100 < 14) ? "th" :
      (count % 10 == 1) ? "st" :
        (count % 10 == 2) ? "nd" :
          (count % 10 == 3) ? "rd" : "th" );
  count++;
  /* Tell the function which called us where the buffer is */
  *buffer_location = my_buffer;
  /* Return the length */
  return len;
}
struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    4, /* Length of the file name */
    "test", /* The file name */
    S_IFREG | S_IRUGO, /* File mode - this is a regular file which can
                        * be read by its owner, its group, and everybody
                        * else */
```

```
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode (linking, removing,
           * etc.) - we don't support any. */
    procfile_read, /* The read function for this file, the function called
                    * when somebody tries to read something from it. */
    NULL /* We could have here a function to fill the file's inode, to
          * enable us to play with permissions, ownership, etc. */
  };

/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);

  /* proc_root is the root directory for the proc fs (/proc). This is
   * where we want our file to be located.
   */
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.4 使用/proc输入

现在我们已经有了两种方法从内核模块中产生输出。一种方法是注册一个设备驱动程序，并且使用mknod创建一个设备文件，另一种方法是创建一个/proc文件。这就允许内核模块告诉我们它可以了解的任何事情。但目前为止我们却无法回应内核模块。向内核模块中发送信息的第一种方法是写入到/proc文件中。

因为/proc文件系统的主要作用是允许内核向进程报告它的状态，所以没有专门为输入设置的机制。数据结构proc_dir_entry中并不包括指向输入函数的指针，就像它包括指向输出函数的指针一样。为了能够写入到一个/proc文件中，需要使用标准的文件系统机制。

在Linux系统中有一个标准的文件系统注册机制。因为每一个文件系统都有处理索引节点和文件操作的函数，所以有一个特殊的数据结构来保存指向这些函数的指针，也就是数据结构inode_operations。在/proc中，每当注册一个新文件时，都允许指定使用哪一个数据结构inode_operations来存取它，这就是使用的机制，一个数据结构inode_operations中包括指向数据结构file_operations的指针，同时数据结构 file_operations中包括指向module_input 和module_output函数的指针。

应该注意的是，在内核中读和写的角色是互换的。读用于输出，而写用于输入，这是因为所说的读写是以用户的角度来说的。如果进程需要从内核中读取信息，内核就需要输出信息。如果进程需要向内核中写入信息，那么内核将接收此信息作为输入。

下面我们看看module_permission函数。每当一个进程试图使用/proc文件时都会调用此函数。它可以决定是否允许对文件的存取。

使用put_user 和 get_user的原因是Linux系统中的内存是分段的。这意味着指针不能指向内存中一个唯一的地址，只能指向内存段中唯一的地址。所以需要知道使用的内存段。系统中有一个内核使用的内存段，而每一个进程还有一个单独的内存段。

进程只能够存取自己的内存段，所以运行进程时没有必要担心内存段。当你编写一个内存模块时，一般情况下你希望存取内核内存段，这是由系统自动处理的。尽管如此，当需要将一个内存缓冲区中的内容在当前进程和内核中传递时，内核函数接收到一个指针，指向进程内存段中的内存缓冲区。宏put_user 和 get_user macros允许你存取该内存。

下面是procfs.c的源代码：

```
/* procfs.c -  create a "file" in /proc, which allows both input and
 * output. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>
/* The module's file functions ************************************ */
/* Here we keep the last message received, to prove that we can process
 * our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
/* Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is
 * this function */
static int module_output(struct inode *inode, /* The inode read */
                         struct file *file,   /* The file read */
                         char *buf, /* The buffer to put data to (in the
                                     * user segment) */
                         int len)  /* The length of the buffer */
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];
  /* We return 0 to indicate end of file, that we have no more information.
   * Otherwise, processes will continue to read from us in an endless loop. */
  if (finished) {
    finished = 0;
    return 0;
  }
  /* We use put_user to copy the string from the kernel's memory segment
   * to the memory segment of the process that called us. get_user, BTW, is
   * used for the reverse. */
  sprintf(message, "Last input:%s", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);
  /* Notice, we assume here that the size of the message is below len, or
   * it will be received cut. In a real life situation, if the size of the
   * message is less than len then we'd return len and on the second call
   * start filling the buffer with the len+1'th byte of the message. */
```

```
  finished = 1;
  return i;  /* Return the number of bytes "read" */
}
/* This function receives input from the user when the user writes to
 * the /proc file. */
static int module_input(struct inode *inode, /* The file's inode */
                        struct file *file,   /* The file itself */
                        const char *buf,     /* The buffer with the input */
                        int length)          /* The buffer's length */
{
  int i;
  /* Put the input into Message, where module_output will later be
   * able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
    Message[i] = get_user(buf+i);
  Message[i] = '\0';  /* we want a standard, zero terminated string */
  /* We need to return the number of input characters used */
  return i;
}
/* This function decides whether to allow an operation (return zero) or
 * not allow it (return a non-zero which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file permissions. The permissions
 * returned by ls -l are for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but only root (uid 0)
   * may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;
  /* If it's anything else, access is denied */
  return -EACCES;
}
/* The file is opened - we don't really care about that, but it does mean
 * we need to increment the module's reference count. */
int module_open(struct inode *inode, struct file *file)
{
  MOD_INC_USE_COUNT;
  return 0;
}
/* The file is closed - again, interesting only because of the reference
 * count. */
void module_close(struct inode *inode, struct file *file)
```

```
{
  MOD_DEC_USE_COUNT;
}
/* Structures to register as the /proc file, with pointers to all the
 * relevant functions. ********************************************** */
/* File operations for our proc file. This is where we place pointers
 * to all the functions called when somebody tries to do something to
 * our file. NULL means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,    /* Somebody opened the file */
    module_close    /* Somebody closed the file */
    /* etc. etc. etc. (they are all given in /usr/include/linux/fs.h).
     * Since we don't put anything here, the system will keep the default
     * data, which in UNIX is zeros (NULLs when taken as pointers). */
  };
/* Inode operations for our proc file. We need it so we'll have some
 * place to specify the file operations structure we want to use, and
 * the function we use for permissions. It's also possible to specify
 * functions to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
    module_permission /* check for permissions */
  };
/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
```

```
    0, /* Inode number - ignore, it will be filled by
      * proc_register_dynamic */
    7, /* Length of the file name */
    "rw_test", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR, /* File mode - this is a regular file which
                  * can be read by its owner, its group, and everybody
                  * else. Also, its owner can write to it.
                  *
                  * Actually, this field is just for reference, it's
                  * module_permission that does the actual check. It
                  * could use this field, but in our implementation it
                  * doesn't, for simplicity. */
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File, /* A pointer to the inode structure for
                      * the file, if we need it. In our case we
                      * do, because we need a write function. */
    NULL  /* The read function for the file. Irrelevant, because we put it
        * in the inode structure above */
  };
/* Module initialization and cleanup ******************************** */
/* Initialize the module - register the proc file */
int init_module()
{
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.5 与设备文件通信

设备文件是用来代表物理设备的。大部分的物理设备既可以用作输出，也可以用作输入，所以系统内核中必须有某种机制使得设备驱动程序接收进程发送给设备的信息。这可以通过打开一个设备文件来实现，该设备文件用于输出，同时进程向设备文件写入，就像写入到一个普通文件中一样。在下面的例子中，这是由device_write函数实现的。

但这还不够。想一下你有一个串行口连接到一个调制解调器中，最自然的事情就是使用设备文件向调制解调器中写入信息，然后从调制解调器中读取信息。但问题是如果你希望和串行口自身通信时该怎么办？

在UNIX系统中解决这个问题的办法是使用一个特殊的函数，叫做 ioctl。每一个设备都可以有它自己的ioctl命令，可以用来从进程向内核发送信息或者从内核向进程返回信息。函数ioctl包括三个参数：一个相应的设备文件的文件描述符、 ioctl数目以及长整型的参数，它可以用来传递任何信息。

参数ioctl数目包括主设备号、ioctl的类型、命令和参数的类型。它通常由宏调用在头文件中创建。在下面的例子中，这个头文件是chardev.h，使用它的程序是ioctl.c。

下面是chardev.c 的源代码：

```
/* chardev.c
 *
 * Create an input/output character device
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* For character devices */
#include <linux/fs.h>       /* The character device definitions are here */
#include <linux/wrapper.h>  /* A wrapper which does next to nothing at
                             * at present, but may help for compatibility
                             * with future versions of Linux */
/* Our own IOCTL numbers */
#include "chardev.h"
#define SUCCESS 0
/* Device Declarations ******************************************************** */
/* The name for our device, as it will appear in /proc/devices */
#define DEVICE_NAME "char_dev"
/* The maximum length of the message for the device */
#define BUF_LEN 80
/* Is the device open right now? Used to prevent concurent access into
 * the same device */
static int Device_Open = 0;
/* The message the device will give when asked */
static char Message[BUF_LEN];
/* How far did the process reading the message get? Useful if the
 * message is larger than the size of the buffer we get to fill in
 * device_read. */
static char *Message_Ptr;
/* This function is called whenever a process attempts to open the device
 * file */
static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_open(%p,%p)\n", inode, file);
#endif
  /* We don't want to talk to two processes at the same time */
  if (Device_Open)
    return -EBUSY;
  /* If this was a process, we would have had to be more careful here,
   * because one process might have checked Device_Open right before the
   * other one tried to increment it. However, we're in the kernel, so
   * we're protected against context switches.
   */
  Device_Open++;
  /* Initialize the message */
```

```
  Message_Ptr = Message;
  MOD_INC_USE_COUNT;
  return SUCCESS;
}
/* This function is called when a process closes the device file. It
 * is not allowed to fail */
static void device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
  printk ("device_release(%p,%p)\n", inode, file);
#endif
  /* We're now ready for our next caller */
  Device_Open --;
  MOD_DEC_USE_COUNT;
}
/* This function is called whenever a process which already opened the
 * device file attempts to read from it. */
static int device_read(struct inode *inode,
                       struct file *file,
                       char *buffer,   /* The buffer to fill with the data */
                       int length)     /* The length of the buffer
                                        * (mustn't write beyond that!) */
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;
#ifdef DEBUG
  printk("device_read(%p,%p,%p,%d)\n",
    inode, file, buffer, length);
#endif
  /* If we're at the end of the message, return 0 (which signifies end
   * of file) */
  if (*Message_Ptr == 0)
    return 0;
  /* Actually put the data into the buffer */
  while (length && *Message_Ptr)  {
    /* Because the buffer is in the user data segment, not the kernel
     * data segment, assignment wouldn't work. Instead, we have to use
     * put_user which copies data from the kernel data segment to the user
     * data segment. */
    put_user(*(Message_Ptr++), buffer++);
    length --;
    bytes_read ++;
  }
#ifdef DEBUG
   printk ("Read %d bytes, %d left\n",
     bytes_read, length);
#endif
   /* Read functions are supposed to return the number of bytes actually
    * inserted into the buffer */
  return bytes_read;
}
```

```
/* This function is called when somebody tries to write into our device
 * file. */
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
{
  int i;
  return 1;
#ifdef DEBUG
  printk ("device_write(%p,%p,%s,%d)",
    inode, file, buffer, length);
#endif
  for(i=0; i<length && i<BUF_LEN; i++)
    Message[i] = get_user(buffer+i);
  Message_Ptr = Message;
  /* Again, return the number of input characters used */
  return i;
}
/* This function is called whenever a process tries to do an ioctl on
 * our device file. We get two extra parameters (additional to the
 * inode and file structures, which all device functions get): the number
 * of the ioctl called and the parameter given to the ioctl function.
 *
 * If the ioctl is write or read/write (meaning output is returned to
 * the calling process), the ioctl call returns the output of this
 * function.
 */
int device_ioctl(struct inode *inode,
                 struct file *file,
                 unsigned int ioctl_num,    /* The number of the ioctl */
                 unsigned long ioctl_param) /* The parameter to it */
{
  int i;
  char *temp;
  /* Switch according to the ioctl called */
  switch (ioctl_num) {
    case IOCTL_SET_MSG:
      /* Receive a pointer to a message (in user space) and set that to
       * be the device's message. */
      /* Get the parameter given to ioctl by the process */
      temp = (char *) ioctl_param;
      /* Find the length of the message */
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
        ;
      /* Don't reinvent the wheel - call device_write */
      device_write(inode, file, (char *) ioctl_param, i);
      break;
    case IOCTL_GET_MSG:
      /* Give the current message to the calling process - the parameter
       * we got is a pointer, fill it. */
```

```
      i = device_read(inode, file, (char *) ioctl_param, 99);
      /* Warning - we assume here the buffer length is 100. If it's less
       * than that we might overflow the buffer, causing the process to
       * core dump.
       *
       * The reason we only allow up to 99 characters is that the NULL
       * which terminates the string also needs room. */
      /* Put a zero at the end of the buffer, so it will be properly
       * terminated */
      put_user('\0', (char *) ioctl_param+i);
      break;
    case IOCTL_GET_NTH_BYTE:
      /* This ioctl is both input (ioctl_param) and output (the return
       * value of this function) */
      return Message[ioctl_param];
      break;
  }
  return SUCCESS;
}
/* Module Declarations ************************************************ */
/* This structure will hold the functions to be called when
 * a process does something to the device we created. Since a pointer to
 * this structure is kept in the devices table, it can't be local to
 * init_module. NULL is for unimplemented functions. */
struct file_operations Fops = {
  NULL,   /* seek */
  device_read,
  device_write,
  NULL,   /* readdir */
  NULL,   /* select */
  device_ioctl,   /* ioctl */
  NULL,   /* mmap */
  device_open,
  device_release  /* a.k.a. close */
};
/* Initialize the module - Register the character device */
int init_module()
{
  int ret_val;
  /* Register the character device (atleast try) */
  ret_val = module_register_chrdev(MAJOR_NUM,
                                   DEVICE_NAME,
                                   &Fops);
  /* Negative values signify an error */
  if (ret_val < 0) {
    printk ("Sorry, registering the character device failed with %d\n",
      ret_val);
    return ret_val;
  }
  printk ("Registeration is a success. The major device number is %d.\n",
```

```
    MAJOR_NUM);
  printk ("If you want to talk to the device driver, you'll have to\n");
  printk ("create a device file. We suggest you use:\n");
  printk ("mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
  printk ("The device file name is important, because the ioctl program\n");
  printk ("assumes that's the file you'll use.\n");
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  int ret;
  /* Unregister the device */
  ret = module_unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
  /* If there's an error, report it */
  if (ret < 0)
    printk("Error in module_unregister_chrdev: %d\n", ret);
}
```

下面是chardev.h的源代码：

```
/* chardev.h - the header file with the ioctl definitions.
 *
 * The declarations here have to be in a header file, because they need
 * to be known both to the kernel module (in chardev.c) and the process
 * calling ioctl (ioctl.c)
 */
#ifndef CHARDEV_H
#define CHARDEV_H
#include <linux/ioctl.h>
/* The major device number. We can't rely on dynamic registration any
 * more, because ioctls need to know it. */
#define MAJOR_NUM 100
/* Set the message of the device driver */
#define IOCTL_SET_MSG _IOR(MAJOR_NUM, 0, char *)
/* _IOR means that we're creating an ioctl command number for passing
 * information from a user process to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device number we're using.
 *
 * The second argument is the number of the command (there could be
 * several with different meanings).
 *
 * The third argument is the type we want to get from the process to the
 * kernel.
 */
/* Get the message of the device driver */
#define IOCTL_GET_MSG _IOR(MAJOR_NUM, 1, char *)
 /* This IOCTL is used for output, to get the message of the device driver.
  * However, we still need the buffer to place the message in to be input,
  * as it is allocated by the process.
  */
```

```
/* Get the n'th byte of the message */
#define IOCTL_GET_NTH_BYTE _IOWR(MAJOR_NUM, 2, int)
 /* The IOCTL is used for both input and output. It receives from the
  * user a number, n, and returns Message[n]. */
/* The name of the device file */
#define DEVICE_FILE_NAME "char_dev"
#endif
```

下面是ioctl.c的源代码：

```
/* ioctl.c - the process to use ioctl's to control the kernel module
 *
 * Until now we could have used cat for input and output. But now we need
 * to do ioctl's, which require writing our own process.
 */
#include "chardev.h"    /* device specifics, such as ioctl numbers and
                         * the major device file. */

#include <fcntl.h>      /* open */
#include <unistd.h>     /* exit */
#include <sys/ioctl.h>  /* ioctl */
/* Functions for the ioctl calls */
ioctl_set_msg(int file_desc, char *message)
{
  int ret_val;
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
  if (ret_val < 0) {
    printf ("ioctl_set_msg failed:%d\n", ret_val);
    exit(-1);
  }
}
ioctl_get_msg(int file_desc)
{
  int ret_val;
  char message[100];
  /* Warning - this is dangerous because we don't tell the kernel how
   * far it's allowed to write, so it might overflow the buffer. In a
   * real production program, we would have used two ioctls - one to tell
   * the kernel the buffer length and another to give it the buffer to fill
   */
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
  if (ret_val < 0) {
    printf ("ioctl_get_msg failed:%d\n", ret_val);
    exit(-1);
  }
  printf("get_msg message:%s\n", message);
}
ioctl_get_nth_byte(int file_desc)
{
  int i;
```

```
  char c;
  printf("get_nth_byte message:");
  i = 0;
  while (c != 0) {
    c = ioctl(file_desc, IOCTL_GET_NTH_BYTE, i++);
    if (c < 0) {
      printf("ioctl_get_nth_byte failed at the %d'th byte:\n", i);
      exit(-1);
    }
    putchar(c);
  }
  putchar('\n');
}
/* Main - Call the ioctl functions */
main()
{
  int file_desc, ret_val;
  char *msg = "Message passed by ioctl\n";
  file_desc = open(DEVICE_FILE_NAME, 0);
  if (file_desc < 0) {
    printf ("Can't open device file: %s\n", DEVICE_FILE_NAME);
    exit(-1);
  }
  ioctl_get_nth_byte(file_desc);
  ioctl_get_msg(file_desc);
  ioctl_set_msg(file_desc, msg);
  close(file_desc);
}
```

18.6 启动参数

在前面的例子中，不得不硬性规定某些参数，例如/proc文件系统的文件名，或者设备的主设备号。这样应用起来十分的不方便，我们的目标是编写一个灵活易用的程序。

我们使用命令行参数为程序或者内核模块传递启动参数。在使用内核模块时，不使用参数argc 和 argv。可以在内核模块中定义全局变量，然后使用 insmod命令来写入这些变量。

在此内核模块中，我们定义了两个变量：str1和str2。所需要做的是编译内核模块，然后运行insmod str1=xxx str2=yyy即可。当调用init_module时，str1将指向字符串xxx，str2将指向字符串yyy。

下面是param.c的源程序：

```
/* param.c
 *
 * Receive command line parameters at module installation
 */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
```

```
#include <stdio.h>  /* I need NULL */
char *str1, *str2;
/* Initialize the module - show the parameters */
int init_module()
{
  if (str1 == NULL || str2 == NULL)
    printk("Next time, do insmod param str1=<something> str2=<something>\n");
  else
    printk("Strings:%s and %s\n", str1, str2);
  return 0;
}
/* Cleanup */
void cleanup_module()
{
}
```

18.7 系统调用

到现在为止，我们所做的只是使用已经定义好的内核机制来注册 /proc文件系统和设备处理程序，这对编写设备驱动程序是很有用的。但如果希望对系统作某些改动的话，这是不够的。

在系统中真正被所有进程都使用的内核通信方式是系统调用。例如当进程请求内核服务时，就使用的是系统调用。

一般情况下，进程是不能够存取系统内核的。它不能存取内核使用的内存段，也不能调用内核函数，CPU的硬件结构保证了这一点。只有系统调用是一个例外。进程使用寄存器中适当的值跳转到内核中事先定义好的代码中执行，（当然，这些代码是只读的）。在Intel结构的计算机中，这是由中断0x80实现的。

进程可以跳转到的内核中的位置叫做 system_call。在此位置的过程检查系统调用号，它将告诉内核进程请求的服务是什么。然后，它再查找系统调用表 sys_call_table，找到希望调用的内核函数的地址，并调用此函数，最后返回。

所以，如果希望改变一个系统调用的函数，需要做的是编写一个自己的函数，然后改变sys_call_table中的指针指向该函数，最后再使用 cleanup_module将系统调用表恢复到原来的状态。

下面的代码是这样一个例子。我们希望监视一个用户，每当该用户打开一个文件时就打印一条信息。我们用自己的程序 our_sys_open来替代打开文件的系统调用。此程序检查当前进程的uid，如果它与所监视用户的uid相等，则调用printk来显示将要打开的文件的名称。然后，它再调用原来的打开文件的系统调用，真正地打开文件。

在这个例子中，函数init_module用来替换sys_call_table表中的相应的位置，同时保存其原先的指针。函数 cleanup_module则用来将此变量恢复到中断表中。

下面是syscall.c的源代码：

```
/* syscall.c
 *
 * System call "stealing" sample
 */
/* The necessary header files */
/* Standard in kernel modules */
```

```
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
#include <sys/syscall.h>  /* The list of system calls */
#include <linux/sched.h>  /* For the current (process) structure, we need
                           * this to know who the current user is. */
/* The system call table (a table of functions). We just define this as
 * external, and the kernel will fill it up for us when we are insmod'ed
 */
extern void *sys_call_table[];
int uid;  /* UID we want to spy on - will be filled from the command line */
/* A pointer to the original system call. The reason we keep this, rather
 * than call the original function (sys_open), is because somebody else might
 * have replaced the system call before us. Note that this is not 100% safe,
 * because if another module replaced sys_open before us, then when we're
 * inserted we'll call the function in that module - and it might be removed
 * before we are. */
asmlinkage int (*original_call)(const char *, int, int);
/* The function we'll replace sys_open (the function called when you call
 * the open system call) with. To find the exact prototype, with the number
 * and type of arguments, we find the original function first (it's at
 * fs/open.c.
 * In theory, this means that we're tied to the current version of the
 * kernel. In practice, the system calls almost never change (it would
 * wreck havoc and require programs to be recompiled, since the system
 * calls are the interface between the kernel and the rest of the world).
 */
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
  int i = 0;
  char ch;
  /* Check if this is the user we're spying on */
  if (uid == current->uid) {  /* current->uid is the uid of the user who
                               ran the process which called the system
                               call we got */
    /* Report the file, if relevant */
    printk("Opened file: ");
    do {
      ch = get_user(filename+i);
      i++;
      printk("%c", ch);
    } while (ch != 0);
    printk("\n");
  }
  /* Call the original sys_open - otherwise, we lose the ability to open
   * files */
  return original_call(filename, flags, mode);
}
/* Initialize the module - replace the system call */
int init_module()
```

```
{
  /* Warning - too late for it now, but maybe for next time... */
  printk("I'm dangerous. I hope you did a sync before you insmod'ed me.\n");
  printk("My counterpart, cleapup_module(), is even more dangerous. If\n");
  printk("you value your file system, it will be \"sync; rmmod\" \n");
  printk("when you remove it.\n");
  /* Keep a pointer to the original function in original_call, and
   * then replace the system call in the system call table with
   * our_sys_open */
  original_call = sys_call_table[__NR_open];
  sys_call_table[__NR_open] = our_sys_open;
  /* To get the address of the function for system call foo, go to
   * sys_call_table[__NR_foo]. */
  return 0;
}
/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
  /* Return the system call back to normal */
  if (sys_call_table[__NR_open] != our_sys_open) {
    printk("Somebody else also played with the open system call\n");
    printk("The system may be left in an unstable state.\n");
  }
  sys_call_table[__NR_open] = original_call;
}
```

18.8 阻塞进程

当进程请求内核模块服务时，如果此时内核模块正忙，那么可以将进程放入睡眠状态直到模块空闲。

下面就是这样一个例子。文件/proc/sleep每次只可以被一个进程打开。如果文件已经打开，内核模块将调用 module_interruptible_sleep_on函数，此函数将进程的状态改为TASK_INTERRUPTIBLE，这意味着进程在被唤醒之前一直处于 WaitQ队列中。然后，此函数调用调度算法切换到另一个进程。

当进程处理完文件以后，它将关闭文件，然后调用 module_close函数。此函数唤醒队列中所有的等待进程。也可以使用一个信号，例如用 Ctrl+C (SIGINT)来唤醒一个进程。

下面是 sleep.c的源代码：

```
/* sleep.c - create a /proc file, and if several processes try to open it
 * at the same time, put all but one to sleep */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>
/* For putting processes to sleep and waking them up */
#include <linux/sched.h>
```

```
#include <linux/wrapper.h>
/* The module's file functions ************************************ */
/* Here we keep the last message received, to prove that we can process
 * our input */
#define MESSAGE_LENGTH 80
static char Message[MESSAGE_LENGTH];
/* Since we use the file operations struct, we can't use the special proc
 * output provisions - we have to use a standard read function, which is
 * this function */
static int module_output(struct inode *inode, /* The inode read */
                         struct file *file,   /* The file read */
                         char *buf, /* The buffer to put data to (in the
                                     * user segment) */
                         int len)  /* The length of the buffer */
{
  static int finished = 0;
  int i;
  char message[MESSAGE_LENGTH+30];
  /* Return 0 to signify end of file - that we have nothing more to say
   * at this point. */
  if (finished) {
    finished = 0;
    return 0;
  }
  /* If you don't understand this by now, you're hopeless as a kernel
   * programmer. */
  sprintf(message, "Last input:%s\n", Message);
  for(i=0; i<len && message[i]; i++)
    put_user(message[i], buf+i);
  finished = 1;
  return i;  /* Return the number of bytes "read" */
}
/* This function receives input from the user when the user writes to
 * the /proc file. */
static int module_input(struct inode *inode, /* The file's inode */
                        struct file *file,   /* The file itself */
                        const char *buf,     /* The buffer with the input */
                        int length)          /* The buffer's length */
{
  int i;
  /* Put the input into Message, where module_output will later be
   * able to use it */
  for(i=0; i<MESSAGE_LENGTH-1 && i<length; i++)
    Message[i] = get_user(buf+i);
  Message[i] = '\0';  /* we want a standard, zero terminated string */
  /* We need to return the number of input characters used */
  return i;
}
```

```
/* 1 if the file is currently open by somebody */
int Already_Open = 0;
/* Queue of processes who want our file */
static struct wait_queue *WaitQ = NULL;
/* Called when the /proc file is opened */
static int module_open(struct inode *inode,
                       struct file *file)
{
  /* If the file's flags include O_NONBLOCK, it means the process doesn't
   * want to wait for the file. In this case, if the file is already open,
   * we should fail with -EAGAIN, meaning "you'll have to try again",
   * instead of blocking a process which would rather stay awake. */
  if ((file->f_flags & O_NONBLOCK) && Already_Open)
    return -EAGAIN;
  /* This is the correct place for MOD_INC_USE_COUNT because if a process is
   * in the loop, which is within the kernel module, the kernel module must
   * not be removed. */
  MOD_INC_USE_COUNT;
  /* If the file is already open, wait until it isn't */
  while (Already_Open)
  {
    /* This function puts the current process, including any system calls,
     * such as us, to sleep. Execution will be resumed right after the
     * function call, either because somebody called wake_up(&WaitQ) (only
     * module_close does that, when the file is closed) or when a signal,
     * such as Ctrl-C, is sent to the process */
    module_interruptible_sleep_on(&WaitQ);
    /* If we woke up because we got a signal we're not blocking, return
     * -EINTR (fail the system call). This allows processes to be killed
     * or stopped. */
    if (current->signal & ~current->blocked) {
      /* It's important to put MOD_DEC_USE_COUNT here, because for processes
       * where the open is interrupted there will never be a corresponding
       * close. If we don't decrement the usage count here, we will be left
       * with a positive usage count which we'll have no way to bring down to
       * zero, giving us an immortal module, which can only be killed by
       * rebooting the machine. */
      MOD_DEC_USE_COUNT;
      return -EINTR;
    }
  }
  /* If we got here, Already_Open must be zero */
  /* Open the file */
  Already_Open = 1;
  return 0;  /* Allow the access */
}
/* Called when the /proc file is closed */
static void module_close(struct inode *inode,
                         struct file *file)
```

```
{
  /* Set Already_Open to zero, so one of the processes in the WaitQ will
   * be able to set Already_Open back to one and to open the file. All the
   * other processes will be called when Already_Open is back to one, so
   * they'll go back to sleep. */
  Already_Open = 0;
  /* Wake up all the processes in WaitQ, so if anybody is waiting for the
   * file, they can have it. */
  module_wake_up(&WaitQ);
  /* One less process interested in us */
  MOD_DEC_USE_COUNT;
}
/* This function decides whether to allow an operation (return zero) or
 * not allow it (return a non-zero which indicates why it is not allowed).
 *
 * The operation can be one of the following values:
 * 0 - Execute (run the "file" - meaningless in our case)
 * 2 - Write (input to the kernel module)
 * 4 - Read (output from the kernel module)
 *
 * This is the real function that checks file permissions. The permissions
 * returned by ls -l are for referece only, and can be overridden here.
 */
static int module_permission(struct inode *inode, int op)
{
  /* We allow everybody to read from our module, but only root (uid 0)
   * may write to it */
  if (op == 4 || (op == 2 && current->euid == 0))
    return 0;
  /* If it's anything else, access is denied */
  return -EACCES;
}
/* Structures to register as the /proc file, with pointers to all the
 * relevant functions. ********************************************** */
/* File operations for our proc file. This is where we place pointers
 * to all the functions called when somebody tries to do something to
 * our file. NULL means we don't want to deal with something. */
static struct file_operations File_Ops_4_Our_Proc_File =
  {
    NULL,  /* lseek */
    module_output,  /* "read" from the file */
    module_input,   /* "write" to the file */
    NULL,  /* readdir */
    NULL,  /* select */
    NULL,  /* ioctl */
    NULL,  /* mmap */
    module_open,      /* called when the /proc file is opened */
    module_close      /* called when it's classed */
  };
/* Inode operations for our proc file. We need it so we'll have some
 * place to specify the file operations structure we want to use, and
```

```
 * the function we use for permissions. It's also possible to specify
 * functions to be called for anything else which could be done to an
 * inode (although we don't bother, we just put NULL). */
static struct inode_operations Inode_Ops_4_Our_Proc_File =
  {
    &File_Ops_4_Our_Proc_File,
    NULL, /* create */
    NULL, /* lookup */
    NULL, /* link */
    NULL, /* unlink */
    NULL, /* symlink */
    NULL, /* mkdir */
    NULL, /* rmdir */
    NULL, /* mknod */
    NULL, /* rename */
    NULL, /* readlink */
    NULL, /* follow_link */
    NULL, /* readpage */
    NULL, /* writepage */
    NULL, /* bmap */
    NULL, /* truncate */
    module_permission /* check for permissions */
  };
/* Directory entry */
static struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    5, /* Length of the file name */
    "sleep", /* The file name */
    S_IFREG | S_IRUGO | S_IWUSR, /* File mode - this is a regular file which
                        * can be read by its owner, its group, and everybody
                        * else. Also, its owner can write to it.
                        *
                        * Actually, this field is just for reference, it's
                        * module_permission that does the actual check. It
                        * could use this field, but in our implementation it
                        * doesn't, for simplicity. */
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    &Inode_Ops_4_Our_Proc_File, /* A pointer to the inode structure for
                             * the file, if we need it. In our case we
                             * do, because we need a write function. */
    NULL  /* The read function for the file. Irrelevant, because we put it
          * in the inode structure above */
  };
/* Module initialization and cleanup ********************************** */
/* Initialize the module - register the proc file */
int init_module()
{
```

```
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
  /* proc_root is the root directory for the proc fs (/proc). This is
   * where we want our file to be located.
   */
}
/* Cleanup - unregister our file from /proc. This could get dangerous if
 * there are still processes waiting in WaitQ, because they are inside our
 * open function, which will get unloaded. I'll explain how to avoid removal
 * of a kernel module in such a case in chapter 9. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

18.9 替换printk

可以使用一个指向当前任务的指针来获得它的tty的数据结构，在此数据结构中可以找到一个指向字符串写函数的指针。

下面是printk.c的源程序：

```
/* printk.c - send textual output to the tty you're running on, regardless
 * of whether it's passed through X11, telnet, etc. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */
/* Necessary here */
#include <linux/sched.h>    /* For current */
#include <linux/tty.h>      /* For the tty declarations */
/* Print the string to the appropriate tty, the one the current task uses */
void print_string(char *str)
{
  struct tty_struct *my_tty;
  /* The tty for the current task */
  my_tty = current->tty;
  /* If my_tty is NULL, it means that the current task has no TTY you can
   * print to (this is possible, for example, if it's a daemon). In this
   * case, there's nothing we can do. */
  if (my_tty != NULL) {
    /* my_tty->driver is a struct which holds the TTY's functions, one of
     * which (write) is used to write strings to the tty. It can be used to
     * take a string either from the user's memory segment or the kernel's
     * memory segment.
     *
     * The function's first parameter is the tty to write to, because the
     * same function would normally be used for all tty's of a certain type.
     * The second parameter controls whether the function receives a string
     * from kernel memory (false, 0) or from user memory (true, non zero).
     * The third parameter is a pointer to a string, and the fourth parameter
     * is the length of the string.
```

```
    */
   (*(my_tty->driver).write)(my_tty,   /* The tty itself */
                              0, /* We don't take the string from user space */
                              str, /* String */
                              strlen(str));  /* Length */
   /* TTYs were originally hardware devices, which (usually) adhered strictly
    * to the ASCII standard. According to ASCII, to move to a new line
    * you need two characters, a carriage return and a line feed. In UNIX,
    * on the other hand, the ASCII line feed is used for both purposes -
    * so we can't just use \n, because it wouldn't have a carriage return and
    * the next line will start at the column right after the line feed.
    * BTW, this is the reason why the text file  is different between
    * UNIX and Windows. In CP/M and its derivatives, such as MS-DOS and
    * Windows, the ASCII standard was strictly adhered to, and therefore a
    * new line requires both a line feed and a carriage return.
    */
   (*(my_tty->driver).write)(my_tty,
                              0,
                              "\015\012",
                              2);
  }
}
/* Module initialization and cleanup ****************************** */
/* Initialize the module - register the proc file */
int init_module()
{
  print_string("Module Inserted");
  return 0;
}
/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  print_string("Module Removed");
}
```

18.10 调度任务

我们经常需要在一个特定的时间执行某一个任务。如果任务是由一个进程执行的，可以把它放入crontab中。如果任务是由一个内核模块执行的，那么有两种实现方法。一种是把一个进程放入crontab中，等待需要时由系统调用来唤醒。但这样做的效率很低。

另一种方法是创建一个函数，此函数可以在计时器中断时调用。再创建一个任务，把它放在数据结构tq_struct中，此数据结构中包括一个指向这个函数的指针。然后，使用 queue_task把任务放入到任务队列 tq_timer中，此队列中保存着下一次计时器中断时可以执行的任务。因为我们希望此函数可以持续地执行，所以需要把它不断地放回到队列 tq_timer中。

下面是sched.c的源程序：

```
/* sched.c - scheduale a function to be called on every timer interrupt. */
/* The necessary header files */
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
```

```
#include <linux/module.h>   /* Specifically, a module */
/* Necessary because we use the proc fs */
#include <linux/proc_fs.h>
/* We scheduale tasks here */
#include <linux/tqueue.h>
/* We also need the ability to put ourselves to sleep and wake up later */
#include <linux/sched.h>
/* The number of times the timer interrupt has been called so far */
static int TimerIntrpt = 0;
/* This is used by cleanup, to prevent the module from being unloaded while
 * intrpt_routine is still in the task queue */
static struct wait_queue *WaitQ = NULL;
static void intrpt_routine(void *);
/* The task queue structure for this task, from tqueue.h */
static struct tq_struct Task = {
  NULL,   /* Next item in list - queue_task will do this for us */
  0,      /* A flag meaning we haven't been inserted into a task queue yet */
  intrpt_routine, /* The function to run */
  NULL    /* The void* parameter for that function */
};
/* This function will be called on every timer interrupt. Notice the void*
 * pointer - task functions can be used for more than one purpose, each
 * time getting a different parameter. */
static void intrpt_routine(void *irrelevant)
{
  /* Increment the counter */
  TimerIntrpt++;
  /* If cleanup wants us to die */
  if (WaitQ != NULL)
    wake_up(&WaitQ);   /* Now cleanup_module can return */
  else
    queue_task(&Task, &tq_timer);  /* Put ourselves back in the task queue */
}
/* Put data into the proc fs file. */
int procfile_read(char *buffer, char **buffer_location, off_t offset,
                  int buffer_length, int zero)
{
  int len;  /* The number of bytes actually used */
  /* This is static so it will still be in memory when we leave this
   * function */
  static char my_buffer[80];
  static int count = 1;
  /* We give all of our information in one go, so if the anybody asks us
   * if we have more information the answer should always be no.
   */
  if (offset > 0)
    return 0;
  /* Fill the buffer and get its length */
  len = sprintf(my_buffer, "Timer was called %d times so far\n", TimerIntrpt);
  count++;
  /* Tell the function which called us where the buffer is */
```

```
  *buffer_location = my_buffer;
  /* Return the length */
  return len;
}
struct proc_dir_entry Our_Proc_File =
  {
    0, /* Inode number - ignore, it will be filled by
        * proc_register_dynamic */
    5, /* Length of the file name */
    "sched", /* The file name */
    S_IFREG | S_IRUGO, /* File mode - this is a regular file which can
                        * be read by its owner, its group, and everybody
                        * else */
    1,  /* Number of links (directories where the file is referenced) */
    0, 0,  /* The uid and gid for the file - we give it to root */
    80, /* The size of the file reported by ls. */
    NULL, /* functions which can be done on the inode (linking, removing,
           * etc.) - we don't support any. */
    procfile_read, /* The read function for this file, the function called
                    * when somebody tries to read something from it. */
    NULL /* We could have here a function to fill the file's inode, to
          * enable us to play with permissions, ownership, etc. */
  };
/* Initialize the module - register the proc file */
int init_module()
{
  /* Put the task in the tq_timer task queue, so it will be executed at
   * next timer interrupt */
  queue_task(&Task, &tq_timer);
  /* Success if proc_register_dynamic is a success, failure otherwise */
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
}
/* Cleanup */
void cleanup_module()
{
  /* Unregister our /proc file */
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
  /* Sleep until intrpt_routine is called one last time. This is necessary,
   * because otherwise we'll deallocate the memory holding intrpt_routine and
   * Task while tq_timer still references them. Notice that here we don't
   * allow signals to interrupt us.
   *
   * Notice that since WaitQ is now not NULL, this automatically tells
   * the interrupt routine it's time to die. */
 sleep_on(&WaitQ);
}
```