```
    gid_t                       i_gid;
    kdev_t                      i_rdev;
    off_t                       i_size;
    time_t                      i_atime;
    time_t                      i_mtime;
    time_t                      i_ctime;
    unsigned long               i_blksize;
    unsigned long               i_blocks;
    unsigned long               i_version;
    unsigned long               i_nrpages;
    struct semaphore            i_sem;
    struct inode_operations     *i_op;
    struct super_block          *i_sb;
    struct wait_queue           *i_wait;
    struct file_lock            *i_flock;
    struct vm_area_struct       *i_mmap;
    struct page                 *i_pages;
    struct dquot                *i_dquot[MAXQUOTAS];
    struct inode                *i_next, *i_prev;
    struct inode                *i_hash_next, *i_hash_prev;
    struct inode                *i_bound_to, *i_bound_by;
    struct inode                *i_mount;
    unsigned short              i_count;
    unsigned short              i_flags;
    unsigned char               i_lock;
    unsigned char               i_dirt;
    unsigned char               i_pipe;
    unsigned char               i_sock;
    unsigned char               i_seek;
    unsigned char               i_update;
    unsigned short              i_writecount;
    union {
        struct pipe_inode_info  pipe_i;
        struct minix_inode_info minix_i;
        struct ext_inode_info   ext_i;
        struct ext2_inode_info  ext2_i;
        struct hpfs_inode_info  hpfs_i;
        struct msdos_inode_info msdos_i;
        struct umsdos_inode_info umsdos_i;
        struct iso_inode_info   isofs_i;
        struct nfs_inode_info   nfs_i;
        struct xiafs_inode_info xiafs_i;
        struct sysv_inode_info  sysv_i;
        struct affs_inode_info  affs_i;
        struct ufs_inode_info   ufs_i;
        struct socket           socket_i;
        void                    *generic_ip;
    } u;
};
```

ipc_perm

该数据结构描述了一个system V IPC对象的访问许可。

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* sequence number */
};
```

irqaction

该数据结构描述系统中断处理器。

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

linux_binfmt

用于指代每一种Linux所能理解的二进制文件格式。

```
struct linux_binfmt {
  struct linux_binfmt * next;
  long *use_count;
  int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
  int (*load_shlib)(int fd);
  int (*core_dump)(long signr, struct pt_regs * regs);
};
```

mem_map_t

该数据结构包含有内存中物理页信息。

```
typedef struct page {
  /* these must be first (free area handling) */
  struct page        *next;
  struct page        *prev;
  struct inode       *inode;
  unsigned long      offset;
  struct page        *next_hash;
  atomic_t           count;
  unsigned           flags;     /* atomic flags, some possibly
                                   updated asynchronously */
  unsigned           dirty:16,
                     age:8;
```

```
  struct wait_queue  *wait;
  struct page        *prev_hash;
  struct buffer_head *buffers;
  unsigned long      swap_unlock_entry;
  unsigned long      map_nr;     /* page->map_nr == page - mem_map */
} mem_map_t;
```

mm_struct

该数据结构描述了一个任务或进程的虚存。

```
struct mm_struct {
  int count;
  pgd_t * pgd;
  unsigned long context;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack, start_mmap;
  unsigned long arg_start, arg_end, env_start, env_end;
  unsigned long rss, total_vm, locked_vm;
  unsigned long def_flags;
  struct vm_area_struct * mmap;
  struct vm_area_struct * mmap_avl;
  struct semaphore mmap_sem;
};
```

pci_bus

每个pci_bus数据结构指代一个系统的PCI总线。

```
struct pci_bus {
  struct pci_bus  *parent;     /* parent bus this bridge is on */
  struct pci_bus  *children;   /* chain of P2P bridges on this bus */
  struct pci_bus  *next;       /* chain of all PCI buses */

  struct pci_dev  *self;       /* bridge device as seen by parent */
  struct pci_dev  *devices;    /* devices behind this bridge */

  void    *sysdata;            /* hook for sys-specific extension */

  unsigned char  number;       /* bus number */
  unsigned char  primary;      /* number of primary bridge */
  unsigned char  secondary;    /* number of secondary bridge */
  unsigned char  subordinate;  /* max number of subordinate buses */
};
```

pci_dev

每一个pci_dev数据结构指代一个系统PCI设备（包括PCI-PCI桥和PCI-ISA桥）。

```
/*
 * There is one pci_dev structure for each slot-number/function-number
 * combination:
 */
struct pci_dev {
  struct pci_bus  *bus;      /* bus this device is on */
```

```
  struct pci_dev  *sibling;  /* next device on this bus */
  struct pci_dev  *next;     /* chain of all devices */

  void    *sysdata;          /* hook for sys-specific extension */

  unsigned int  devfn;       /* encoded device & function index */
  unsigned short  vendor;
  unsigned short  device;
  unsigned int  class;       /* 3 bytes: (base,sub,prog-if) */
  unsigned int  master : 1;  /* set if device is master capable */
  /*
   * In theory, the irq level can be read from configuration
   * space and all would be fine.  However, old PCI chips don't
   * support these registers and return 0 instead.  For example,
   * the Vision864-P rev 0 chip can uses INTA, but returns 0 in
   * the interrupt line and pin registers.  pci_init()
   * initializes this field with the value at PCI_INTERRUPT_LINE
   * and it is the job of pcibios_fixup() to change it if
   * necessary.  The field must not be 0 unless the device
   * cannot generate interrupts at all.
   */
  unsigned char  irq;        /* irq generated by this device */
};
```

request

该数据结构用于向系统中的块设备发申请。

```
struct request {
    volatile int rq_status;
#define RQ_INACTIVE            (-1)
#define RQ_ACTIVE              1
#define RQ_SCSI_BUSY           0xffff
#define RQ_SCSI_DONE           0xfffe
#define RQ_SCSI_DISCONNECTING  0xffe0
    kdev_t rq_dev;
    int cmd;        /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long current_nr_sectors;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    struct request * next;
};
```

rtable

每个rtable数据结构包含了将报文发往一个IP主机的信息，在IP路由缓存中用到该数据。

```
struct rtable
{
```

```
    struct rtable     *rt_next;
    __u32             rt_dst;
    __u32             rt_src;
    __u32             rt_gateway;
    atomic_t          rt_refcnt;
    atomic_t          rt_use;
    unsigned long     rt_window;
    atomic_t          rt_lastuse;
    struct hh_cache   *rt_hh;
    struct device     *rt_dev;
    unsigned short    rt_flags;
    unsigned short    rt_mtu;
    unsigned short    rt_irtt;
    unsigned char     rt_tos;
};
```

semaphore

该数据结构用于保护对临界区数据结构和代码的访问。

```
struct semaphore {
    int count;
    int waking;
    int lock ;                  /* to make waking testing atomic */
    struct wait_queue *wait;
};
```

sk_buff

当在协议层间传输数据时，用该数据结构描述数据信息。

```
struct sk_buff
{
  struct sk_buff      *next;       /* Next buffer in list                  */
  struct sk_buff      *prev;       /* Previous buffer in list              */
  struct sk_buff_head *list;       /* List we are on                       */
  int                 magic_debug_cookie;
  struct sk_buff      *link3;      /* Link for IP protocol level buffer chains
*/
  struct sock         *sk;         /* Socket we are owned by               */
  unsigned long       when;        /* used to compute rtt's                */
  struct timeval      stamp;       /* Time we arrived                      */
  struct device       *dev;        /* Device we arrived on/are leaving by  */
  union
  {
      struct tcphdr   *th;
      struct ethhdr   *eth;
      struct iphdr    *iph;
      struct udphdr   *uh;
      unsigned char   *raw;
      /* for passing file handles in a unix domain socket */
      void            *filp;
  } h;
```

```
    union
    {
        /* As yet incomplete physical layer views */
        unsigned char   *raw;
        struct ethhdr   *ethernet;
    } mac;

    struct iphdr        *ip_hdr;     /* For IPPROTO_RAW                        */
    unsigned long       len;         /* Length of actual data                  */
    unsigned long       csum;        /* Checksum                               */
    __u32               saddr;       /* IP source address                      */
    __u32               daddr;       /* IP target address                      */
    __u32               raddr;       /* IP next hop address                    */
    __u32               seq;         /* TCP sequence number                    */
    __u32               end_seq;     /* seq [+ fin] [+ syn] + datalen          */
    __u32               ack_seq;     /* TCP ack sequence number                */
    unsigned char       proto_priv[16];
    volatile char       acked,       /* Are we acked ?                         */
                        used,        /* Are we in use ?                        */
                        free,        /* How to free this buffer                */
                        arp;         /* Has IP/ARP resolution finished         */
    unsigned char       tries,       /* Times tried                            */
                        lock,        /* Are we locked ?                        */
                        localroute,  /* Local routing asserted for this frame */
                        pkt_type,    /* Packet class                           */
                        pkt_bridged, /* Tracker for bridging                   */
                        ip_summed;   /* Driver fed us an IP checksum           */
#define PACKET_HOST             0       /* To us
*/
#define PACKET_BROADCAST        1       /* To all
*/
#define PACKET_MULTICAST        2       /* To group
*/
#define PACKET_OTHERHOST        3       /* To someone else
*/
    unsigned short      users;       /* User count - see datagram.c,tcp.c      */
    unsigned short      protocol;    /* Packet protocol from driver.           */
    unsigned int        truesize;    /* Buffer size                            */
    atomic_t            count;       /* reference count                        */
    struct sk_buff      *data_skb;   /* Link to the actual data skb            */
    unsigned char       *head;       /* Head of buffer                         */
    unsigned char       *data;       /* Data head pointer                      */
    unsigned char       *tail;       /* Tail pointer                           */
    unsigned char       *end;        /* End pointer                            */
    void                (*destructor)(struct sk_buff *); /* Destruct function */
    __u16               redirport;   /* Redirect port                          */
};
```

sock

每个sock数据结构保存有关于BSD套接字的特定协议信息。

```
struct sock
{
  /* This must be first. */
  struct sock             *sklist_next;
  struct sock             *sklist_prev;

  struct options          *opt;
  atomic_t                wmem_alloc;
  atomic_t                rmem_alloc;
  unsigned long           allocation;       /* Allocation mode */
  __u32                   write_seq;
  __u32                   sent_seq;
  __u32                   acked_seq;
  __u32                   copied_seq;
  __u32                   rcv_ack_seq;
  unsigned short          rcv_ack_cnt;      /* count of same ack */
  __u32                   window_seq;
  __u32                   fin_seq;
  __u32                   urg_seq;
  __u32                   urg_data;
  __u32                   syn_seq;
  int                     users;            /* user count */
/*
 *    Not all are volatile, but some are, so we
 *     might as well say they all are.
 */
  volatile char           dead,
                          urginline,
                          intr,
                          blog,
                          done,
                          reuse,
                          keepopen,
                          linger,
                          delay_acks,
                          destroy,
                          ack_timed,
                          no_check,
                          zapped,
                          broadcast,
                          nonagle,
                          bsdism;
  unsigned long           lingertime;
  int                     proc;

  struct sock             *next;
  struct sock             **pprev;
  struct sock             *bind_next;
  struct sock             **bind_pprev;
  struct sock             *pair;
```

```
    int                     hashent;
    struct sock             *prev;
    struct sk_buff          *volatile send_head;
    struct sk_buff          *volatile send_next;
    struct sk_buff          *volatile send_tail;
    struct sk_buff_head     back_log;
    struct sk_buff          *partial;
    struct timer_list       partial_timer;
    long                    retransmits;
    struct sk_buff_head     write_queue,
                            receive_queue;
    struct proto            *prot;
    struct wait_queue       **sleep;
    __u32                   daddr;
    __u32                   saddr;          /* Sending source */
    __u32                   rcv_saddr;      /* Bound address */
    unsigned short          max_unacked;
    unsigned short          window;
    __u32                   lastwin_seq;    /* sequence number when we last
                                               updated the window we offer */
    __u32                   high_seq;       /* sequence number when we did
                                               current fast retransmit */
    volatile unsigned long  ato;            /* ack timeout */
    volatile unsigned long  lrcvtime;       /* jiffies at last data rcv */
    volatile unsigned long  idletime;       /* jiffies at last rcv */
    unsigned int            bytes_rcv;
/*
 *    mss is min(mtu, max_window)
 */
    unsigned short          mtu;            /* mss negotiated in the syn's */
    volatile unsigned short mss;            /* current eff. mss - can change
*/
    volatile unsigned short user_mss;       /* mss requested by user in ioctl
*/
    volatile unsigned short max_window;
    unsigned long           window_clamp;
    unsigned int            ssthresh;
    unsigned short          num;
    volatile unsigned short cong_window;
    volatile unsigned short cong_count;
    volatile unsigned short packets_out;
    volatile unsigned short shutdown;
    volatile unsigned long  rtt;
    volatile unsigned long  mdev;
    volatile unsigned long  rto;
    volatile unsigned short backoff;
    int                     err, err_soft;  /* Soft holds errors that don't
                                               cause failure but are the
cause
                                               of a persistent failure not
                                               just 'timed out' */
```

```
    unsigned char           protocol;
    volatile unsigned char  state;
    unsigned char           ack_backlog;
    unsigned char           max_ack_backlog;
    unsigned char           priority;
    unsigned char           debug;
    int                     rcvbuf;
    int                     sndbuf;
    unsigned short          type;
    unsigned char           localroute;       /* Route locally only */
/*
 *    This is where all the private (optional) areas that don't
 *    overlap will eventually live.
 */
    union
    {
          struct unix_opt   af_unix;
#if defined(CONFIG_ATALK) || defined(CONFIG_ATALK_MODULE)
        struct atalk_sock   af_at;
#endif
#if defined(CONFIG_IPX) || defined(CONFIG_IPX_MODULE)
        struct ipx_opt      af_ipx;
#endif
#ifdef CONFIG_INET
        struct inet_packet_opt  af_packet;
#ifdef CONFIG_NUTCP
        struct tcp_opt      af_tcp;
#endif
#endif
    } protinfo;
/*
 *    IP 'private area'
 */
    int                     ip_ttl;           /* TTL setting */
    int                     ip_tos;           /* TOS */
    struct tcphdr           dummy_th;
    struct timer_list       keepalive_timer;  /* TCP keepalive hack */
    struct timer_list       retransmit_timer; /* TCP retransmit timer */
    struct timer_list       delack_timer;     /* TCP delayed ack timer */
    int                     ip_xmit_timeout;  /* Why the timeout is running */
    struct rtable           *ip_route_cache;  /* Cached output route */
    unsigned char           ip_hdrincl;       /* Include headers ? */
#ifdef CONFIG_IP_MULTICAST
    int                     ip_mc_ttl;        /* Multicasting TTL */
    int                     ip_mc_loop;       /* Loopback */
    char                    ip_mc_name[MAX_ADDR_LEN]; /* Multicast device name
*/
    struct ip_mc_socklist   *ip_mc_list;      /* Group array */
#endif

/*
```

```
 *    This part is used for the timeout functions (timer.c).
 */
    int                     timeout;        /* What are we waiting for? */
    struct timer_list       timer;          /* This is the TIME_WAIT/receive
                                             * timer when we are doing IP
                                             */
    struct timeval          stamp;
 /*
  *    Identd
  */
    struct socket           *socket;
 /*
  *    Callbacks
  */
    void                    (*state_change)(struct sock *sk);
    void                    (*data_ready)(struct sock *sk,int bytes);
    void                    (*write_space)(struct sock *sk);
    void                    (*error_report)(struct sock *sk);

};
```

socket

每个socket数据结构保存一个BSD套接字的信息，但它不是独立存在的，而是 VFS inode 数据结构的一个部分。

```
struct socket {
  short                 type;         /* SOCK_STREAM, ...             */
  socket_state          state;
  long                  flags;
  struct proto_ops      *ops;         /* protocols do most everything */
  void                  *data;        /* protocol data                */
  struct socket         *conn;        /* server socket connected to   */
  struct socket         *iconn;       /* incomplete client conn.s     */
  struct socket         *next;
  struct wait_queue     **wait;       /* ptr to place to wait on      */
  struct inode          *inode;
  struct fasync_struct *fasync_list; /* Asynchronous wake up list    */
  struct file           *file;        /* File back pointer for gc     */
};
```

task_struct

每个task_struct数据结构描述了一个系统中的进程或任务。

```
struct task_struct {
/* these are hardcoded - don't touch */
  volatile long         state;          /* -1 unrunnable, 0 runnable, >0 stopped
*/
  long                  counter;
  long                  priority;
  unsigned              long signal;
  unsigned              long blocked;   /* bitmap of masked signals */
```

```
  unsigned              long flags;     /* per process flags, defined below */
  int errno;
  long                  debugreg[8];    /* Hardware debugging registers */
  struct exec_domain    *exec_domain;
/* various fields */
  struct linux_binfmt   *binfmt;
  struct task_struct    *next_task, *prev_task;
  struct task_struct    *next_run,  *prev_run;
  unsigned long         saved_kernel_stack;
  unsigned long         kernel_stack_page;
  int                   exit_code, exit_signal;
  /* ??? */
  unsigned long         personality;
  int                   dumpable:1;
  int                   did_exec:1;
  int                   pid;
  int                   pgrp;
  int                   tty_old_pgrp;
  int                   session;
  /* boolean value for session group leader */
  int                   leader;
  int                   groups[NGROUPS];
  /*
   * pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->p_pptr->pid)
   */
  struct task_struct    *p_opptr, *p_pptr, *p_cptr,
                        *p_ysptr, *p_osptr;
  struct wait_queue     *wait_chldexit;
  unsigned short        uid,euid,suid,fsuid;
  unsigned short        gid,egid,sgid,fsgid;
  unsigned long         timeout, policy, rt_priority;
  unsigned long         it_real_value, it_prof_value, it_virt_value;
  unsigned long         it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list     real_timer;
  long                  utime, stime, cutime, cstime, start_time;
/* mm fault and swap info: this can arguably be seen as either
   mm-specific or thread-specific */
  unsigned long         min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long         swap_address;
  unsigned long         old_maj_flt;    /* old value of maj_flt */
  unsigned long         dec_flt;        /* page fault count of the last time */
  unsigned long         swap_cnt;       /* number of pages to swap on next pass
*/
/* limits */
  struct rlimit         rlim[RLIM_NLIMITS];
  unsigned short        used_math;
  char                  comm[16];
/* file system info */
```

```
  int                   link_count;
  struct tty_struct     *tty;           /* NULL if no tty */
/* ipc stuff */
  struct sem_undo       *semundo;
  struct sem_queue      *semsleeping;
/* ldt for this task - used by Wine.  If NULL, default_ldt is used */
  struct desc_struct *ldt;
/* tss for this task */
  struct thread_struct tss;
/* filesystem information */
  struct fs_struct      *fs;
/* open file information */
  struct files_struct  *files;
/* memory management info */
  struct mm_struct      *mm;
/* signal handlers */
  struct signal_struct *sig;
#ifdef __SMP__
  int                   processor;
  int                   last_processor;
  int                   lock_depth;     /* Lock depth.
                                           We can context switch in and out
                                           of holding a syscall kernel lock...
*/
#endif
};
```

timer_list

该数据结构用于实现进程的真实时间定时器。

```
struct timer_list {
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires;
  unsigned long data;
  void (*function)(unsigned long);
};
```

tq_struct

每个tq_struct数据结构包含有队列列中一项工作的信息。

```
struct tq_struct {
    struct tq_struct *next;   /* linked list of active bh's */
    int sync;                 /* must be initialized to zero */
    void (*routine)(void *);  /* function to call */
    void *data;               /* argument to function */
};
```

vm_area_struct

每个vm_area_struct数据结构描述一个进程的虚内存空间。

```
struct vm_area_struct {
  struct mm_struct * vm_mm;  /* VM area parameters */
  unsigned long vm_start;
  unsigned long vm_end;
  pgprot_t vm_page_prot;
  unsigned short vm_flags;
/* AVL tree of VM areas per task, sorted by address */
  short vm_avl_height;
  struct vm_area_struct * vm_avl_left;
  struct vm_area_struct * vm_avl_right;
/* linked list of VM areas per task, sorted by address */
  struct vm_area_struct * vm_next;
/* for areas with inode, the circular list inode->i_mmap */
/* for shm areas, the circular list of attaches */
/* otherwise unused */
  struct vm_area_struct * vm_next_share;
  struct vm_area_struct * vm_prev_share;
/* more */
  struct vm_operations_struct * vm_ops;
  unsigned long vm_offset;
  struct inode * vm_inode;
  unsigned long vm_pte;      /* shared mem */
};
```

附录A　有用的Web和FTP站点

下列是一些有用的World Wide Web和ftp站点。

http://www.azstarnet.com/～axplinux：这是David Mosberger-Tang的Alpha AXP linux网络站点，所有的Alpha AXP How To均可在此找到，并且还包括大量的 Linux的指示器和Alpha AXP的特殊信息，如CPU数据表格。

http://www.redhat.com/Red Hat的网络站点，包括大量的有用的指示器。

ftp://sunsite.unc.edv：大量免费软件的主要汇集点。 pub/linux目录下有linux专用软件。

http://www.intel.com：Intel公司的网络站点，是查寻Intel芯片信息的好地方。

http://www.ssc.com/lj/index.html：The Linux Journal是一本非常好的Linux杂志，值得每年摘录其中优秀的文章。

http://www.blackdown.org/java/linux.html：关于Java和Linux的基本站点。

ftp://tsx-11.mit.edu/ftp/pub/Linux：MIT的 Linuxftp站点。

ftp://ftp.cs.helsinki.fi/pub/software/linux/kernel：Linux的核心资源。

http://www.linux.org.uk：UF Linux Use Group。

http://sunsite.unc.edu/mdw/linux.html：Linux Documentation Project主页。

http://www.digital.com：Digital Equipment Corporation的主要网页。

http://altavista.digital.com：DIGITAL公司的 Altavista的搜索引擎，是在网络和新闻群中搜寻信息的非常好的站点。

http://www.Linuxhq.com：The Linux HQ网络站点，存储最新的官方和非官方的patch，即帮助读者得到系统所需的最好的内核资源的建议和网络指示器。

http://www.amd.com：The AMD网络站点。

http://www.cyrix.com：Cyrix的网络站点。

附录B 词 汇 表

Argument：函数和例程的参数。

ARP：地址解析协议，用于把IP地址解析成物理硬件地址。

ASCII：美国标准信息交换码，字母表中每一个字母用一个 8位代码表示。

Bit：数据的一个位，表示0或1(开或关)。

Bottom Half Handler：在内核中的工作队列处理器。

Byte：8位数据。

C：一种高级程序语言，大多数的Linux内核用C语言编写。

CPU：中央处理单元，计算机的主要部件，包括微处理器和处理器。

Data Structure：内存中一组域数据的集合。

Device Driver：设备驱动程序，控制特定设备的软件。

DMA：直接内存访问。

ELF：可执行和可链接文件格式，现已由 UNIX系统实验室指定为Linux中最通用的对象文件格式。

EIDE：扩展IDE。

Executable Image：包含机器指令和数据的结构化文件，可将该文件载入处理器虚内存并执行。

Function：完成一项操作的一块软件。

IDE：集成硬盘电路。

Image：见Executable image。

IP：网际协议。

IPC：内部进程通信。

Interface：标准的例程调用和数据传输方式。

IRQ：中断请求队列。

ISA：工业标准体系结构，这是一种较为过时的标准数据总线接口。

Kernel Module：动态载入的内核函数。

Kilobytes：一千个数据字节，通常记为KB。

Megabytes：一百万个数据字节，通常记为MB。

Microprocessor：一种高度集成的CPU。

Module：一种包含CPU指令的文件，这些指令以汇编或C语言编成。

Object File：一种文件，包含已被编译但未链接的机器指令和数据。

Page：页，物理内存被均分为若干页。

Pointer：指向其他内存位置的内存数据。

Process：一种能够执行程序的实体。

Processor：Microprocessor的简写，等价于CPU。

PCI：外围构件互连，一种标准，定义计算机系统外围构件如何连接。

Peripheral：一种代替系统CPU工作的智能处理器，如IDE控制芯片。

Protocol：一种网络语言，用于两远程进程间传输应用数据。

Register：芯片中的一处区域，用于存储信息或指令。

Routine：类似于Function，只是不需要返回值。

SCSI：小型计算机系统接口。

shell：一个程序，起到操作系统和用户之间接口的作用，也称为 command shell，Linux中最通用的是bash shell。

SMP：对称多处理机，系统中有多个处理器，平均分工。

Socket：一个套接字代表网络连接中的一端， Linux支持BSD Socket接口。

System V：1983生产的一种变型UNIX，包含了System V IPC机

TCP：传输控制协议。

Task Queue：Linux内核用于延迟工作的一种机制。

UDP：用户数据报协议。

Virtual memory：一种硬件和软件机制，用于支持大于实际值的系统物理内存。

第1章 Hello, World

如果第一个程序员是一个山顶洞人，它在山洞壁 (第一台计算机)上凿出的第一个程序应该是用羚羊图案构成的一个字符串“ Hello, World ”。罗马的编程教科书也应该是以程序“ Salut, Mundi ”开始的。我不知道如果打破这个传统会带来什么后果，至少我还没有勇气去做第一个吃螃蟹的人。

内核模块至少必须有两个函数：init_module和cleanup_module。第一个函数是在把模块插入内核时调用的；第二个函数则在删除该模块时调用。一般来说， init_module可以为内核的某些东西注册一个处理程序，或者也可以用自身的代码来取代某个内核函数 (通常是先干点别的什么事，然后再调用原来的函数)。函数cleanup_module的任务是清除掉init_module所做的一切，这样，这个模块就可以安全地卸载了。

```
ex hello.c

/* hello.c
 * Copyright (C) 1998 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif



/* 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");
}
```

1.1 内核模块的Makefiles文件

内核模块并不是一个独立的可执行文件，而是一个对象文件，在运行时内核模块被链接到内核中。因此，应该使用 -c命令参数来编译它们。还有一点需要注意，在编译所有内核模块时，都将需要定义好某些特定的符号。

- _ _KERNEL_ _—— 这个符号告诉头文件：这个程序代码将在内核模式下运行，而不要作为用户进程的一部分来执行。
- MODULE —— 这个符号告诉头文件向内核模块提供正确的定义。
- LINUX —— 从技术的角度讲，这个符号不是必需的。然而，如果程序员想要编写一个重要的内核模块，而且这个内核模块需要在多个操作系统上编译，在这种情况下，程序员将会很高兴自己定义了LINUX这个符号。这样一来，在那些依赖于操作系统的部分，这个符号就可以提供条件编译了。

还有其它的一些符号，是否包含它们要取决于在编译内核时使用了哪些命令参数。如果用户不太清楚内核是怎样编译的，可以查看文件 /usr/include/linux/config.h。

- _ _SMP_ _—— 对称多处理。如果编译内核的目的是为了支持对称多处理，在编译时就需要定义这个符号 (即使内核只是在一个 CPU上运行也需要定义它)。当然，如果用户使用对称多处理，那么还需要完成其它一些任务 (参见第12章)。
- CONFIG_MODVERSIONS —— 如果CONFIG_MODVERSIONS可用，那么在编译内核模块时就需要定义它，并且包含头文件 /usr/include/linux/modversions.h。还可以用代码自身来完成这个任务。

```
ex Makefile

# Makefile for a basic kernel module

CC=gcc
MODCFLAGS := -Wall -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.
```

完成了以上这些任务以后，剩下唯一要做的事就是切换到根用户下 (你不是以root身份编译内核模块的吧？别玩什么惊险动作哟！)，然后根据自己的需要插入或删除 hello模块。在执行完insmod命令以后，可以看到新的内核模块在 /proc/modules中。

顺便提一下，Makefile建议用户不要从X执行insmod命令的原因在于，当内核有个消息需要使用printk命令打印出来时，内核会把该消息发送给控制台。当用户没有使用 X时，该消息

将发送到用户正在使用的虚拟终端 (用户可以用Alt-F<n>来选择当前终端)，然后用户就可以看到这个消息了。而另一方面，当用户使用 X时，存在两种可能性。一种情况是用户用命令xterm -C打开了一个控制台，这时输出将被发送到那个控制台；另一种情况是用户没有打开控制台，这时输出将送往虚拟终端 7 —— 被X所“ 覆盖 ”的一个虚拟终端。

当用户的内核不太稳定时，没有使用 X的用户更有可能取得调试信息。如果没有使用 X，printk将直接从内核把调试消息发送到控制台。而另一方面，在X中printk的消息将被送给一个用户模式的进程(xterm -C)。当那个进程获得CPU时间时，它将把该消息传送给X服务器进程。然后，当X服务器获得CPU时间时，它将显示该消息 —— 但是一个不稳定的内核通常意味着系统将要崩溃或者重新启动，所以用户不希望推迟错误信息显示的时间，因为该信息可能会向用户解释什么地方出了问题，如果显示的时刻晚于系统崩溃或重启的时刻，用户将会错过这个重要的信息。

1.2 多重文件内核模块

有时候在多个源文件间划分内核模块是很有意义的。这时用户需要完成下面三件任务：

1) 除了一个源文件以外，在其它所有源文件中加入一行 #define _ _ NO_VERSION_ _。这点很重要，因为module.h中通常会包含有kernel_version的定义(kernel_version是一个全局变量，它表明该模块是为哪个内核版本所编译的)。如果用户需要version.h文件，那么用户必须自己把它包含在源文件中，因为在定义了 _ _NO_VERSION_ _的情况下， module.h是不会为用户完成这个任务的。

2) 像平常一样编译所有的源文件。

3) 把所有的对象文件组合进一个文件中。在 x86下，可以使用命令：

ld -m elf_i386 -r -o〈模块名称〉.o (第一个源文件).o (第二个源文件).o来完成这个任务。

下面是这种内核模块的一个例子。

ex start.c

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */


/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* 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;
}
ex stop.c
/* stop.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version. This
 * file includes just the stop routine.
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */

#define __NO_VERSION__      /* This isn't "the" file
                             * of the kernel module */
#include <linux/module.h>   /* Specifically, a module */

#include <linux/version.h>   /* Not included by
                              * module.h because
                              * of the __NO_VERSION__ */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk("Short is the life of a kernel module\n");
}
ex Makefile

# Makefile for a multifile kernel module
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLINUX

hello.o:    start.o stop.o
            ld -m elf_i386 -r -o hello.o start.o stop.o

start.o:    start.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c start.c

stop.o:             stop.c /usr/include/linux/version.h
            $(CC) $(MODCFLAGS) -c stop.c
```

第2章　字符设备文件

我们现在就可以吹牛说自己是内核程序员了。虽然我们所写的内核模块还什么也干不了，但我们仍然为自己感到骄傲，简直可以称得上趾高气扬。但是，有时候在某种程度上我们也会感到缺少点什么，简单的模块并不是太有趣。

内核模块主要通过两种方法与进程打交道。一种方法是通过设备文件（例如在目录/dev中的文件），另一种方法是使用proc文件系统。既然编写内核模块的主要原因之一就是支持某些类型的硬件设备，那么就让我们从设备文件开始吧。

设备文件最初的用途是使进程与内核中的设备驱动程序通信，并且通过设备驱动程序再与物理设备(调制解调器、终端等等)通信。下面我们要讲述实现这一任务的方法。

每个设备驱动程序都被赋予一个主编号，主要用于负责某几种类型的硬件。可以在/proc/devices中找到驱动程序以及它们对应的主编号的列表。由设备驱动程序管理的每个物理设备都被赋予一个从编号。这些设备中的每一个，不管是否真正安装在计算机系统上，都将对应一个特殊的文件，该文件称为设备文件，所有的设备文件都包含在目录 /dev中。

例如，如果执行命令 ls -l /dev/hd[ab]*，用户将可以看到与一个计算机相连接的所有的IDE硬件分区。注意，所有的这些硬盘分区都使用同一个主编号： 3，但是从编号却各不相同。需要强调的是，这里假设用户使用的是 PC体系结构。我并不知道在其它体系结构上运行的Linux的设备是怎么样的。

在安装了系统以后，所有的设备文件都由命令 mknod创建出来。从技术的角度上讲，并没有什么特别的原因一定要把这些设备文件放在目录 /dev中，这只不过是一个有用的传统习惯而已。如果读者创建设备文件的目的只不过是为了试试看，就像本章的练习一样，那么把该设备文件放置在编译内核模块的目录中可能会更有意义一些。

设备一般分为两种类型：字符设备和块设备。它们的区别在于块设备具有一个请求缓冲区，所以块设备可以选择按照何种顺序来响应这些请求。这对于存储设备来说是很重要的。在存储设备中，读或写相邻的扇区速度要快一些，而读写相互之间离得较远的扇区则要慢得多。另一个区别在于块设备只能以成块的形式接收输入和返回输出（块的大小根据设备类型的变化而有所不同），而字符设备则可以随心所欲地使用任意数目的字节。当前大多数设备都是字符设备，因为它们既不需要某种形式的缓冲，也不需要按照固定的块大小来进行操作。如果想知道某个设备文件对应的是块设备还是字符设备，用户可以执行命令 ls -l，查看一下该命令的输出中的第一个字符，如果第一个字符是“ b ”，则对应的是块设备；如果是“ c ”，则对应的是字符设备。

模块分为两个独立的部分：模块部分和设备驱动程序部分。前者用于注册设备。函数init_module调用module_register_chrdev，把该设备驱动程序加入到内核的字符设备驱动程序表中，它还会返回供驱动程序所使用的主编号。函数 cleanup_module则取消该设备的注册。

注册某设备和取消它的注册是这两个函数最基本的功能。内核中的东西并不是按照它们自己的意愿主动开始运行的，就像进程一样，而是由进程通过系统调用来调用，或者由硬件

设备通过中断来调用，或者由内核的其它部分调用 (只需调用特定的函数)，它们才会执行。因此，如果用户往内核中加入了代码，就必须把它作为某种特定类型事件的处理程序进行注册；而在删除这些代码时，用户必须取消它的注册。

设备驱动程序一般是由四个 device_<action>函数所组成的，如果用户需要处理具有对应主编号的设备文件，就可以调用这四个函数。通过 file_operations结构Fops内核可以知道调用哪些函数。因为该结构的值是在注册设备时给定的，它包含了指向这四个函数的指针。

在这里我们还需要记住的一点是：无论如何不能乱删内核模块。原因在于如果设备文件是由进程打开的，而我们删去了该内核模块，那么使用该文件就将导致对正确的函数 (读/写)原来所处的存储位置的调用。如果我们走运，那里没有装入什么其它的代码，那我们至多得到一些难看的错误信息，而如果我们不走运，在原来的同一位置已经装入了另一个内核模块，这就意味着跳转到了内核中另一个函数的中间，这样做的后果是不堪设想的，起码不会是令人愉快的。

一般来说，如果用户不愿意让某件事发生，可以让执行这件事的函数返回一个错误代码(一个负数)。而对cleanup_module来说这是不可能的，因为它是一个 void函数。一旦调用了cleanup_module，这个模块就死了。然而，还有一个计数器记录了有多少个其它的内核模块正在使用该内核模块，这个计数器称为引用计数器 (就是位于文件/proc/modules信息行中的最后那个数值)。如果这个数值不为零， rmmod将失败。模块的引用计数值可以从变量mod_use_count_中得到。因为有些宏是专门为处理这个变量而定义的(如MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT)，我们宁愿使用这些宏，也不愿直接对mod_use_count_进行操作，这样一来，如果将来实现方法有所变化，我们也会很安全。

them, rather than mod_use_count_ directly, so we'll be safe if the implementation c
in the future.

```
ex chardev.c

/* chardev.c
 * Copyright (C) 1998-1999 by Ori Pomerantz
 *
 * 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 */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 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 */


/* In 2.2.3 /usr/include/linux/version.h includes
 * a macro for this, but 2.0.35 doesn't - so I add
 * it here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* Conditional compilation. LINUX_VERSION_CODE is
 * the code (as per KERNEL_VERSION) of this version. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for put_user */
#endif



#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

  /* This is how you get the minor device number in
 * case you have more than one physical device using
 * the driver. */
printk("Device: %d.%d\n",
   inode->i_rdev >> 8, inode->i_rdev & 0xFF);

/* 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 version 2.2 changed it.
 * Unfortunately, I don't have access to an SMP box
 * to check how it works with SMP.
 */

Device_Open++;

/* Initialize the message. */
sprintf(Message,
  "If I told you once, I told you %d times - %s",
```

```
  counter++,
  "Hello, world\n");
/* 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 doesn't have a return value in
 * version 2.0.x because it can't fail (you must ALWAYS
 * be able to close a device). In version 2.2.x it is
 * allowed to fail - but we won't let it.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                           struct file *file)
#endif
{
#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;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
```

```
#endif
}
/* This function is called whenever a process which
 * have already opened the device file attempts to
 * read from it. */


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(struct file *file,
    char *buffer,    /* The buffer to fill with data */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
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!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

  /* 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 - unsupported in this example. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
    const char *buffer,    /* The buffer */
    size_t length,   /* The length of the buffer */
    loff_t *offset)  /* Our offset in the file */
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  return -EINVAL;
}



/* Module Declarations ***************************** */

/* The major device number for the device. This is
 * global (well, static, which in this context is global
 * within this file) 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,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
```

```
  NULL,   /* flush */
#endif
  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 ("%s device failed with %d\n",
        "Sorry, registering the character",
        Major);
    return Major;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success.",
          Major);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("We suggest you use:\n");
  printk ("mknod <name> c %d <minor>\n", Major);
  printk ("You can try different minor numbers %s",
          "and see what happens.\n");

  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 unregister_chrdev: %d\n", ret);
}
```

多内核版本源文件

系统调用是内核提供给进程的主要接口，它并不随着内核版本的变化而有所改变。当然

可能会加入新的系统调用，但是老的系统调用是永远不会改变的。这主要是为了提供向后兼容性的需要 —— 新的内核版本不应该使原来工作正常的进程出现问题。在大多数情况下，设备文件也应该保持不变。另一方面，内核里面的内部接口则可以变化，并且也确实随着内核版本的变化而改变了。

Linux内核的版本可以划分为稳定版本(n.<偶数>.m)和开发版本(n.<奇数>.m)两种。开发版本中包括所有最新最酷的思想，当然其中也可能有一些将来会被认为是馊主意的错误，有些将会在下一个版本中重新实现。因此，用户不能认为在这些版本之间接口也会保持一致 (这就是我为什么懒得在本书中介绍它们的原因，这需要做大量的工作，而且很快就会过时被淘汰)。另一方面，在稳定的版本中，我们可以无视错误修正版本 (带数字m的版本)而认为接口是保持不变的。

这个MPG版本包含对Linux内核版本2.0.x和版本2.2.x的支持。因为这两个版本之间存在差异，这就要求用户根据内核版本号来进行条件编译。为了做到这一点，可以使用宏 LINUX_VERSION_CODE。在内核版本a.b.c中，该宏的值将会是$2^{16}a+2^{8}b+c$。为了获取特定内核版本的值，我们可以使用宏 KERNEL_VERSION。在2.0.35中该宏没有定义，如果需要的话我们可以自己定义它。

第3章　/proc文件系统

在Linux中，内核和内核模块还可以通过另一种方法把信息发送给进程，这种方法就是/proc 文件系统。最初/proc 文件系统是为了可以轻松访问有关进程的信息而设计的 (这就是它的名称的由来)，现在每一个内核部分只要有些信息需要报告，都可以使用 /proc 文件系统，例如/proc /modules包含一个模块的列表，/proc /meminfo包含有关内存使用的统计信息。

使用/proc 文件系统的方法其实与使用设备驱动程序的方法是非常类似的 —— 用户需要创建一个结构，该结构包含了 /proc文件所需要的所有信息，包括指向任意处理程序函数的指针 (在我们本章的例子中只有一个处理程序函数，当有人试图读 /proc文件时将调用这个函数)。然后，init_module将向内核注册这个结构，而 cleanup_module将取消它的注册。

在程序中之所以需要使用 proc_register_dynamic，是因为我们不想事先判断文件所使用的索引节点编号，而是让内核去决定，这样可以避免编号冲突。一般文件系统都是位于磁盘上的，而不是仅仅存在于内存中 (/proc是存在于内存中的)，在那种情况下，索引节点编号是一个指向某个磁盘位置的指针，在那个位置上存放了该文件的索引节点 (简写为inode)。索引节点包含该文件的有关信息，例如文件的访问权限，以及指向某个或者某些磁盘位置的指针，在这个或者这些磁盘位置中，存放着文件的数据。

因为在文件打开或者关闭时该模块不会被调用，所以我们无需在该模块中使用MOD_INC_USE_COUNT和MOD_DEC_USE_COUNT。如果文件打开以后模块被删除了，没有任何措施可以避免这一后果。在下一章中，我们将学习到一种比较难于实现，但却相对方便的方法，可以用于处理/proc文件，我们也可以使用那个方法来防止这个问题。

```
ex procfs.c

/* procfs.c -  create a "file" in /proc
 * Copyright (C) 1998-1999 by Ori Pomerantz
 */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


/* Necessary because we use the proc fs */
```

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



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* 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 uses
   the get_info field of proc_dir_entry struct (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. */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* 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);
}
```

第4章　把/proc用于输入

到目前为止，可以通过两种方法从内核模块产生输出：我们可以注册一个设备驱动程序，并使用mknod命令创建一个设备文件；我们还可以创建一个 /proc文件。这样内核模块就可以告诉我们各种各样的信息。现在唯一的问题是我们没有办法来回答它。如果要把输入信息发送给内核模块，第一个方法就是把这些信息写回到 /proc文件中。

因为编写proc文件系统的主要目的是为了让内核可以把它的状态报告给进程，对于输入并没有提供相应的特别措施。结构 proc_dir_entry中并不包含指向输入函数的指针，它只包含指向输出函数的指针。如果需要输入，为了把信息写到 /proc文件中，用户需要使用标准的文件系统机制。

Linux为文件系统注册提供了一个标准的机制。因为每个文件系统都必须具有自己的函数专门用于处于索引节点和文件操作，所以 Linux提供了一个特殊的结构inode_operations，该结构存放指向所有这些函数的指针，其中包含一个指向结构 file_operations的指针。在/proc中，无论何时注册一个新文件，用户都可以指定使用哪个 inode_operations结构来访问它。这就是我们所使用的机制。结构 inode_operations包含指向结构 file_operations的指针，而结构file_operations又包含指向module_input和module_output函数的指针。

注意　在内核中读和写的标准角色是互换的，读函数用于输出，而写函数则用于输入，记住这点很重要。之所以会这样，是因为读和写实际上是站在用户的观点来说的—如果一个进程从内核读信息，内核需要做的是输出这些信息，而如果进程向内核写信息，内核当然会把它当作输入来接收。

这里另外一个有趣的地方是module_permission函数。只要进程试图对/proc文件干点什么，这个函数就将被调用，它可以判断是允许对文件进行访问，还是拒绝这次访问。目前这种判断还只是基于操作本身以及当前所使用的 uid来作出(当前所使用的 uid可以从current得到，current是一个指针，指向包含有关当前运行进程的信息结构)，但是函数module_permission还可以基于用户所选择的任意条件来作出允许或是拒绝访问的判断，例如其它还有什么进程正在使用这个文件、日期和时间、或者我们最近接收到的输入。

在程序中我们之所以使用put_user和get_user，主要是因为Linux内存是分段的(在Intel体系结构下；有些其它的处理器可能会有所不同)。这就意味着一个指针并不能指向内存中的某个唯一的位置，而只能指向一个内存段。为了能够使用指针，用户必须知道它指向的是哪个内存段。内核只对应一个内存段，且每个进程都对应一个内存段。

进程所能访问的唯一的内存段就是它自己的内存段，所以在写将要当作进程来运行的常规程序时，程序员不需要考虑有关分段的问题，而当用户编写内核模块时，通常用户需要访问内核的内存段，该内存段是由系统自动处理的。然而，如果内存缓冲区中的内容需要在当前运行的进程和内核之间传送，内核函数将会接收到一个指针，该指针指向进程段中的内存缓冲区。宏put_user和get_user使用户可以访问那块内存。

ex procfs.c

```
/* procfs.c -  create a "file" in /proc, which allows
 * both input and output. */

/* Copyright (C) 1998-1999 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
/* Necessary because we use proc fs */
#include <linux/proc_fs.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif

/* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,   /* The file read */
```

```
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
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 */
#endif
{
 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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
```

```
    struct file *file,   /* The file itself */
    const char *buf,     /* The buffer with input */
    size_t length,       /* The buffer's length */
    loff_t *offset)      /* offset to file - ignore */
#else
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 */
#endif
{
  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++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buf+i);
  /* In version 2.2 the semantics of get_user changed,
   * it not longer returns a character, but expects a
   * variable to fill up as its first argument and a
   * user segment pointer to fill it from as the its
   * second.
   *
   * The reason for this change is that the version 2.2
   * get_user can also read an short or an int. The way
   * it knows the type of the variable it should read
   * is by using sizeof, and for that it needs the
   * variable itself.
   */
#else
    Message[i] = get_user(buf+i);
#endif
  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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}


/* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush, added here in version 2.2 */
#endif
    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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  /* In version 2.2, proc_register assign a dynamic
   * inode number automatically if it is zero in the
   * structure, so there's no more need for
   * proc_register_dynamic
   */
  return proc_register(&proc_root, &Our_Proc_File);
```

```
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* Cleanup - unregister our file from /proc */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第5章　把设备文件用于输入

设备文件一般代表物理设备，而大多数物理设备既可用于输出，也可用于输入，所以Linux必须提供一些机制，以便内核中的设备驱动程序可以从进程获得输出信息，并把它发送到设备。要做到这一点，可以为输出打开设备文件，并且向它写信息，就像写普通的文件一样。在下面的例子中，这些任务是由 device_write完成的。

当然仅有上面这种方法是不够的。假设用户有一个与调制解调器相连接的串行口 (即使用户拥有的是一个内置式的调制解调器，从 CPU角度来看，它还是由一个与调制解调器相连的串行口来实现的，所以这样的用户也无需过多地苛求自己的想象力)。用户将会做的最自然的事情就是使用设备文件来把信息写到调制解调器 (调制解调器命令或者数据将会通过电话线来传送)，并且利用设备文件从调制解调器读信息 (命令的响应或者数据也是通过电话线接收的)。然而，这会带来一个很明显的问题：如果用户需要与串行口本身交换信息的话，用户该怎么办？例如用户可能发送有关数据发送和接收的速率的值。

在Unix中，可以使用一个称为 ioctl的特殊函数来解决这个问题 (ioctl是输入输出控制的英文缩写)。每个设备都有属于自己的 ioctl命令，可以是读 ioctl(从进程把信息发送到内核)、写 ioctl(把信息返回到进程)、都有或者都没有。调用 ioctl函数必须带上三个参数：适当的设备文件的文件描述符， ioctl编号以及另外一个长整型的参数，用户可以使用这个长整型参数来传送任何信息。

ioctl编号是由主设备编号、 ioctl类型、命令以及参数的类型这几者编码而成。这个 ioctl编号通常是由一个头文件中的宏调用 (取决于类型的不同，可以是 _IO、_IOR、_IOW或者 _IOWR)来创建的。然后，将要使用 ioctl的程序以及内核模块都必须通过 #include命令包含这个头文件。前者包含这个头文件是为了生成适当的 ioctl，而后者是为了能理解它。在下面的例子中，头文件的名称是 chardev.h，而使用它的程序是 ioctl.c。

如果用户希望在自己的内核模块中使用 ioctl，最好是接受正式的 ioctl的约定，这样如果偶尔得到别人的 ioctl，或者如果别人获得了你的 ioctl，你可以知道那些地方出现了错误。如果读者想知道更多的信息，可以查询 Documentation/ioctl -number.txt下的内核源代码树。

```
ex chardev.c

/* chardev.c
 *
 * Create an input/output character device
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* For character devices */

/* The character device definitions are here */
#include <linux/fs.h>

/* A wrapper which does next to nothing at
 * at present, but may help for compatibility
 * with future versions of Linux */
#include <linux/wrapper.h>
/* Our own ioctl numbers */
#include "chardev.h"


/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif



#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)\n", 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.
   *
   * This is NOT the right attitude to take, because we
   * might be running on an SMP box, but we'll deal with
   * SMP in a later chapter.
   */

  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 doesn't have a return value because
 * it cannot fail. Regardless of what else happens, you
 * should always be able to close a device (in 2.0, a 2.2
```

```
 * device file could be impossible to close). */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static int device_release(struct inode *inode,
                          struct file *file)
#else
static void device_release(struct inode *inode,
                          struct file *file)
#endif
{
#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;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;
#endif
}



/* This function is called whenever a process which
 * has already opened the device file attempts to
 * read from it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_read(
    struct file *file,
    char *buffer, /* The buffer to fill with the data */
    size_t length,     /* The length of the buffer */
    loff_t *offset) /* offset to the file */
#else
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!) */
#endif
{
  /* Number of bytes actually written to the buffer */
  int bytes_read = 0;

#ifdef DEBUG
  printk("device_read(%p,%p,%d)\n",
    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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t device_write(struct file *file,
                            const char *buffer,
                            size_t length,
                            loff_t *offset)
#else
static int device_write(struct inode *inode,
                        struct file *file,
                        const char *buffer,
                        int length)
#endif
{
  int i;

#ifdef DEBUG
  printk ("device_write(%p,%s,%d)",
    file, buffer, length);
#endif

  for(i=0; i<length && i<BUF_LEN; i++)
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    get_user(Message[i], buffer+i);
#else
    Message[i] = get_user(buffer+i);
#endif

  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;
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  char ch;
#endif

  /* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, temp);
      for (i=0; ch && i<BUF_LEN; i++, temp++)
        get_user(ch, temp);
```

```
#else
      for (i=0; get_user(temp) && i<BUF_LEN; i++, temp++)
    ;
#endif

      /* Don't reinvent the wheel - call device_write */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      device_write(file, (char *) ioctl_param, i, 0);
#else
      device_write(inode, file, (char *) ioctl_param, i);
#endif
      break;

    case IOCTL_GET_MSG:
      /* Give the current message to the calling
       * process - the parameter we got is a pointer,
       * fill it. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      i = device_read(file, (char *) ioctl_param, 99, 0);
#else
      i = device_read(inode, file, (char *) ioctl_param,
                      99);
#endif
      /* 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,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  NULL,  /* flush */
#endif
  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 ("%s failed with %d\n",
            "Sorry, registering the character device ",
            ret_val);
    return ret_val;
  }

  printk ("%s The major device number is %d.\n",
          "Registeration is a success",
          MAJOR_NUM);
  printk ("If you want to talk to the device driver,\n");
  printk ("you'll have to create a device file. \n");
  printk ("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\n");
  printk ("the ioctl program assumes that's the\n");
  printk ("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);
}
ex chardev.h

/* 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
ex ioctl.c

/* 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.
 */

/* Copyright (C) 1998 by Ori Pomerantz */


/* device specifics, such as ioctl numbers and the
 * major device file. */
#include "chardev.h"


#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);
}
```

第6章 启动参数

在前面所给出的许多例子中，我们不得不把一些东西硬塞进内核模块中，如 /proc文件的文件名或者设备的主设备编号，这样我们就可以使用该设备的 ioctl命令。但这是与 Unix和Linux的宗旨背道而驰的，Unix和Linux提倡编写用户所习惯的易于使用的程序。

在程序或者内核模块开始工作之前，如果希望告诉它一些它所需要的信息，可以使用命令行参数。如果是内核模块，我们不需要使用 argc和argv —— 相反，我们还有更好的选择。我们可以在内核模块中定义一些全局变量，然后使用 insmod命令，它将替我们给这些变量赋值。

在下面这个内核模块中，我们定义了两个全局变量： str1和str2。用户所需做的全部工作就是编译该内核模块，然后运行 insmod str1=xxx str2=yyy。当调用init_module时，str1将指向字符串“ xxx ”，而str2将指向字符串“ yyy ”。

在版本2中，对这些参数不进行类型检查。如果 str1或者str2的第一个字符是一个数字，则内核将用该整数的值填充这个变量，而不会用指向字符串的指针去填充它。如果用户对此不太确定，那么就必须亲自去检查一下。

而另一方面，在版本2.2中，用户使用宏MACRO_PARM告诉insmod自己希望参数、它的名称以及类型是什么样的。这就解决了类型问题，并且允许内核模块接受那些以数字开头的字符串。

```
ex param.c
/* param.c
 *
 * Receive command line parameters at module installation
 */

/* Copyright (C) 1998-99 by Ori Pomerantz */



/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif


#include <stdio.h>  /* I need NULL */
```

```
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Emmanuel Papirakis:
 *
 * Parameter names are now (2.2) handled in a macro.
 * The kernel doesn't resolve the symbol names
 * like it seems to have once did.
 *
 * To pass parameters to a module, you have to use a macro
 * defined in include/linux/modules.h (line 176).
 * The macro takes two parameters. The parameter's name and
 * it's type. The type is a letter in double quotes.
 * be a string.
 */


char *str1, *str2;


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(str1, "s");
MODULE_PARM(str2, "s");
#endif


/* Initialize the module - show the parameters */
int init_module()
{
  if (str1 == NULL || str2 == NULL) {
    printk("Next time, do insmod param str1=<something>");
    printk("str2=<something>\n");
  } else
    printk("Strings:%s and %s\n", str1, str2);

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  printk("If you try to insmod this module twice,");
  printk("(without rmmod'ing\n");
  printk("it first), you might get the wrong");
  printk("error message:\n");
  printk("'symbol for parameters str1 not found'.\n");
#endif

  return 0;
}
```

```
}


/* Cleanup */
void cleanup_module()
{
}
```

第7章 系统调用

到现在为止，我们所做过的唯一的工作就是使用一个定义好的内核机制来注册 /proc文件和设备处理程序。如果用户仅仅希望做一个内核程序员份内的工作，例如编写设备驱动程序，那么以前我们所学的知识已经足够了。但是如果用户想做一些不平凡的事，比如在某些方面，在某种程度上改变一下系统的行为，那应该怎么办呢？答案是，几乎全部要靠自己。

这就是内核编程之所以危险的原因。在编写下面的例题时，我关掉了系统调用 open。这将意味着我不能打开任何文件，不能运行任何程序，甚至不能关闭计算机。我只能把电源开关拔掉。幸运的是，我没有删除掉任何文件。为了保证自己也不丢失任何文件，请读者在执行insmod和rmmod之前先运行sync。

现在让我们忘记/proc文件，忘记设备文件，他们只不过是无关大雅的细节问题。实现内核通信机制的“真正”进程是系统调用，它是被所有进程所使用的进程。当某进程向内核请求服务时(例如打开文件、产生一个新进程、或者请求更多的内存)，它所使用的机制就是系统调用。如果用户希望以一种有趣的方式改变内核的行为，也需要依靠系统调用。顺便说一下，如果用户想知道程序使用的是哪个系统调用，可以运行命令 strace <command> <arguments>。

一般来说，进程是不能访问内核的。它不能访问内核存储，它也不能调用内核函数。CPU的硬件保证了这一点(那就是为什么称之为“保护模式”的原因)。系统调用是这条通用规则的一个特例。在进行系统调用时，进程以适当的值填充注册程序，然后调用一条特殊的指令，而这条指令是跳转到以前定义好的内核中的某个位置 (当然，用户进程可以读那个位置，但却不能对它进行写的操作)。在Intel CPU下，以上任务是通过中断0x80来完成的。硬件知道一旦跳转到这个位置，用户的进程就不再是在受限制的用户模式下运行了，而是作为操作系统内核来运行——于是用户就被允许干所有他想干的事。

进程可以跳转到的那个内核中的位置称为 system_call。那个位置上的过程检查系统调用编号(系统调用编号可以告诉内核进程所请求的是什么服务)。然后，该过程查看系统调用表(sys_call_table)，找出想要调用的内核函数的地址，然后调用那个函数。在函数返回之后，该过程还要做一些系统检查工作，然后再返回到进程 (或者如果进程时间已用完，则返回到另一个进程)。如果读者想读这段代码，可以查看源文件 arch/<architecture>/kernel/entry.S，它就在ENTRY(system_call)那一行的后面。

这样看来，如果我们想要改变某个系统调用的工作方式，我们需要编写自己的函数来实现它(通常是加入一点自己的代码，然后再调用原来的函数)，然后改变sys_call_table表中的指针，使它指向我们的函数。因为我们的函数将来可能会被删除掉，而我们不想使系统处于一个不稳定的状态，所以必须用 cleanup_module使sys_call_table表恢复到它的原始状态，这是很重要的。

本章的源代码就是这样一个内核模块的例子。我们想要“侦听”某个特定的用户。无论何时，只要那个用户一打开某个文件，程序就会用 printk打印出一个消息。为了做到这一点，我们用自己的函数代替了用来打开文件的那个系统调用。我们的函数称为 our_sys_open。该函

数查看当前进程的uid(用户的ID)，如果它就是我们想要侦听的 uid，它就调用printk，显示出将要打开的文件的名称。接下来，不管当前进程的 uid是不是想要侦听的uid，该函数都使用同样的参数调用原来的open函数，真正地打开那个文件。

init_module函数替换sys_call_table表中相应的位置，并把原来的指针存放在一个变量中；而cleanup_module函数则使用那个变量把一切都恢复成原来正常的状态。这种方法是具有一定的危险性的，因为可能有两个内核模块都修改同一个系统调用。现在假设有两个内核模块A和B。A模块打开文件的系统调用是 A_open，而B的系统调用是B_open。当把A插入到内核中时，系统调用被换成了A_open，它在被调用时将会调用原来的sys_open。接下来，当B被插入到内核中时，系统调用将被替换成 B_open，它在被调用时，将会调用它自以为是原始系统调用的那个系统调用，即A_open。

现在假设B先被删除，那么一切都将正常 —— 系统调用将被恢复成A_open，而A_open会调用原始的系统调用。然而，如果 A先被删除然后B被删除，系统将会崩溃。删除 A时系统调用被恢复成原始的sys_open, B_open将被忽略。接着，当删除 B时，B将把系统调用恢复成它自认为是原始的系统调用，即 A_open，而A_open已经不再位于内存中了。乍一看上去，好象用户可以检查系统调用是不是打开文件函数，如果是就不做任何修改 (这样在删除B时它就不会改变系统调用)，似乎这样可以避免问题的发生。但这样做将会导致一个更为严重的问题。当A被删除时，它看到系统调用已经被改变为 B_open，而不再指向A_open，所以A在从内存中删除前不会将系统调用恢复为 sys_open。不幸的是， B_open仍将试图调用 A_open，而A_open已不在内存中了，这样，甚至还不到删除 B时系统就将崩溃。

我认为有两种方法可以解决这个问题。第一个方法是把系统调用恢复为原始的值：sys_open。不幸的是，sys_open不是/proc/ksyms中内核系统表的一部分，所以我们不能访问它。另一个方法是一旦装入了模块，马上设立一个引用计数器，以防止根用户把它 rmmod掉。对于产品模块来说，这样做是很好的，但却不适合于作为教学的例子 —— 这就是我没有在这里实现它的原因。

```
ex syscall.c

/* syscall.c
 *
 * System call "stealing" sample
 */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
```

```
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>  /* The list of system calls */

/* For the current (process) structure, we need
 * this to know who the current user is. */
#include <linux/sched.h>



/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif



/* 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[];


/* UID we want to spy on - will be filled from the
 * command line */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* 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.
 *
 * Another reason for this is that we can't get sys_open.
 * It's a static variable, so it is not exported. */
```

```
asmlinkage int (*original_call)(const char *, int, int);



/* For some reason, in 2.2.3 current->uid gave me
 * zero, not the real user ID. I tried to find what went
 * wrong, but I couldn't do it in a short time, and
 * I'm lazy - so I'll just use the system call to get the
 * uid, the way a process would.
 *
 * For some reason, after I recompiled the kernel this
 * problem went away.
 */
asmlinkage int (*getuid_call)();
/* 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
 * processes).
 */
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 == getuid_call()) {
   /* getuid_call is the getuid system call,
    * which gives the uid of the user who
    * ran the process which called the system
    * call we got */

    /* Report the file, if relevant */
    printk("Opened file by %d: ", uid);
    do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
      get_user(ch, filename+i);
#else
      ch = get_user(filename+i);
#endif
      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 ");
    printk("sync before you insmod'ed me.\n");
    printk("My counterpart, cleanup_module(), is even");
    printk("more dangerous. If\n");
    printk("you value your file system, it will ");
    printk("be \"sync; rmmod\" \n");
    printk("when you remove this module.\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]. */

    printk("Spying on UID:%d\n", uid);

    /* Get the system call for getuid */
    getuid_call = sys_call_table[__NR_getuid];

    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 ");
        printk("open system call\n");
        printk("The system may be left in ");
        printk("an unstable state.\n");
    }

    sys_call_table[__NR_open] = original_call;
}
```

第8章 阻塞处理

如果有人想让你做一些目前无法做到的事，你会怎么处理呢？如果打扰你的是某个人的话，你可以说的唯一的一句话是：“现在不行，我很忙，你走吧！”但是如果是进程让内核模块做一些目前它无法处理的事，内核模块却可以有另一种处理方法。内核可以让进程睡眠，直到能够为它服务为止。毕竟，随时都会有进程被内核置为睡眠状态或者唤醒（这就是多个进程看上去好象同时在一个CPU上运行的道理）。

本章的内核模块就是这样的一个例子。该文件（名为/proc/sleep）每次只能被一个进程所打开。如果文件早已经被打开，内核模块将调用 module_interruptible_sleep_on。这个函数把任务的状态改为TASK_INTERRUPTIBLE（任务是内核数据结构，它包含着有关它所处的进程和系统调用的信息，如果存在系统调用的话），把它加入到WaitQ当中，这就意味着任务在被唤醒之前将不会运行。WaitQ是等待访问文件的任务队列。然后，函数调用上下文调度程度，切换到另一个拥有CPU时间的进程。

在进程结束了文件操作以后，进程将关闭文件，并调用 module_close。该函数将唤醒队列中的所有进程（没有只唤醒一个进程的机制），然后函数返回，刚刚关闭文件的那个进程就可以继续运行了。如果那个进程的时间片用完了，则调度程度将会及时判断出这一点，并将 CPU的控制权交给另一个进程。最后，等待队列中的某个进程将会被调度程序授予 CPU的控制权。该进程将从紧接着调用 module_interruptible_sleep_on后面的那个点开始执行。然后它会设置一个全局变量，告诉其它所有进程文件依然打开着，然后该进程将继续执行。当其它进程获得CPU时间片时，它们将看到那个全局变量，于是继续睡眠。

更为有趣的是，并不是只有 module_close才能唤醒那些等待访问文件的进程。一个信号，例如Ctrl+C（SIGINT）也可以唤醒进程。在那种情况下，我们希望立刻用 EINTR返回。这是很重要的，只有这样用户才能在进程接收到文件之前杀死那个进程。

还需要记住一点，有时候进程并不想睡眠；它们希望或者立刻拿到它们想要的东西，或者直接告诉它们这是不可能的。这样的进程在打开文件时使用标志 O_NONBLOCK。内核在进行该操作时，将会返回错误代码 EAGAIN来作响应，否则该操作就将阻塞，就像下面例子中的打开文件操作一样。本章的源目录中有一个程序 cat_noblock，可以用于带O_NONBLOCK标志打开一个文件。

```
ex sleep.c

/* sleep.c - create a /proc file, and if several
 * processes try to open it at the same time, put all
 * but one to sleep */


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */
```

```
/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 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>
/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif


#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>  /* for get_user and put_user */
#endif


/* 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_output(
    struct file *file,   /* The file read */
    char *buf, /* The buffer to put data to (in the
                * user segment) */
    size_t len,  /* The length of the buffer */
    loff_t *offset) /* Offset in the file - ignore */
#else
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 */
  #endif
  {
    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. */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
static ssize_t module_input(
    struct file *file,    /* The file itself */
    const char *buf,      /* The buffer with input */
    size_t length,        /* The buffer's length */
    loff_t *offset)       /* offset to file - ignore */
#else
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 */
#endif
{
  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++)
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
     get_user(Message[i], buf+i);
#else
```

```
  Message[i] = get_user(buf+i);
#endif
/* we want a standard, zero terminated string */
  Message[i] = '\0';
  /* 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)
  {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    int i, is_sig=0;
#endif

    /* 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. */


/*
 * Emmanuel Papirakis:
 *
 * This is a little update to work with 2.2.*. Signals
 * now are contained in two words (64 bits) and are
 * stored in a structure that contains an array of two
 * unsigned longs. We now have to make 2 checks in our if.
 *
 * Ori Pomerantz:
 *
 * Nobody promised me they'll never use more than 64
 * bits, or that this book won't be used for a version
 * of Linux with a word size of 16 bits. This code
 * would work in any case.
 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)

    for(i=0; i<_NSIG_WORDS && !is_sig; i++)
      is_sig = current->signal.sig[i] &
        ~current->blocked.sig[i];
    if (is_sig) {
#else
    if (current->signal & ~current->blocked) {
#endif
      /* 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 */
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
int module_close(struct inode *inode, struct file *file)
#else
void module_close(struct inode *inode, struct file *file)
#endif
{
  /* 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);

  MOD_DEC_USE_COUNT;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return 0;  /* success */
#endif
}



/* 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 reference 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
    NULL,   /* flush */
#endif
    module_close      /* called when it's classed */
  };



/* Inode operations for our proc file. We need it so
 * we'll have somewhere 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 */
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif

  /* 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 10. */
void cleanup_module()
{
  proc_unregister(&proc_root, Our_Proc_File.low_ino);
}
```

第9章　替换printk

在第1章，我曾经提到过X编程和内核模块编程不能混为一谈。这句话在开发内核模块时是正确的。但是在实际应用中，用户可能希望向运行 tty命令的那个模块发送消息。在释放内核模块以后，这对于识别错误是非常重要的。因为该内核模块将会被所有使用 tty命令的模块所使用。

通过使用current可以做到这一点， current是一个指向当前正在运行的任务的指针，通过使用它可以获得当前任务的tty结构。然后查看tty结构，可以找到指向写字符串的函数的指针，我们就是用这个指针向tty写字符串的。

```
ex printk.c

/* printk.c - send textual output to the tty you're
 * running on, regardless of whether it's passed
 * through X11, telnet, etc. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work
*/
#include <linux/module.h>   /* Specifically, a module */
/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 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");
}
```

第10章 任务调度

常常有一些“内务处理”任务需要我们定时或者经常去做。如果任务是由进程去完成的，我们可以把它放在crontab文件中。如果任务要由内核模块来完成，那么我们将面临两种选择。第一种方案是把一个进程放在 crontab文件中，该进程在需要的时候通过系统调用唤醒模块。例如，通过打开一个文件来做到这一点。第三种方案需要在 crontab中运行一个新进程，把一个新的可执行程序读入内存，而这一切都只是为了唤醒一个早已经位于内存中的内核模块这样做效率是非常低的。

除了以上这两种方法以外，我们还可以创建一个函数，在每次时钟中断时调用一次那个函数。为了做到这一点，需要创建一个任务，存放在结构 tq_struct中，而该结构将存放指向函数的指针。接着，我们使用 queue_task把那个任务放置到一个称为 tq_timer的任务列表中，该任务列表中的任务都将在下次时钟中断时执行。由于我们希望该函数能继续执行下去，无论该函数何时被调用，我们一定要把它放回到 tq_timer中，这样在下一次时钟中断时它还可以执行。

在这里还需要记住一点。当使用 rmmod命令删除一个模块时，首先需要检查它的引用计数器值。如果计数器的值为 0，则调用module_cleanup。然后，该模块以及它的所有函数就被从内存中删除掉了。没有人会记得检查一下看看时钟的任务列表中是否碰巧包含了一个指向这些函数中某个函数的指针，而该函数已经不再是可用的了。很长一段时间以后 (这是从计算机的角度来说的，从人的角度来看这段时间不值一提，有可能比百分之一秒还短)，内核发生了一次时钟中断，并试图调用任务列表中的函数。不幸的是，该函数早已经不在那里了。在大多数情况下，该函数原来所在的内存页还没有被使用，这时用户得到的将是一段难看的错误信息。但如果那个内存位置已经存放了一些新的其它的代码，事情就变得非常糟糕了。不幸的是，我们还没有一种简便的方法可以从任务列表中取消一个任务的注册。

由于cleanup_module不能返回错误代码(它是一个void函数)，解决的方法是根本不让它返回，相反，它调用 sleep_on或者module_sleep_on，把rmmod进程置为睡眠状态。而在此之前，它会设置一个全局变量，通知那些将要在时钟中断时调用的函数停止连接。然后，在下一次时钟中断发生时， rmmod进程将被唤醒，我们的函数已经不在队列中了，这时就可以安全地删除模块。

```
ex sched.c

/* sched.c - scheduale a function to be called on
 * every timer interrupt. */


/* Copyright (C) 1998 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
```

```
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* 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>

/* In 2.2.3 /usr/include/linux/version.h includes a
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
/* 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
```

```
 * 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
    /* Put ourselves back in the task queue */
    queue_task(&Task, &tq_timer);
}
/* 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 */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  return proc_register(&proc_root, &Our_Proc_File);
#else
  return proc_register_dynamic(&proc_root, &Our_Proc_File);
#endif
}


/* 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.
   *
   * Since WaitQ is now not NULL, this automatically
   * tells the interrupt routine it's time to die. */
 sleep_on(&WaitQ);
}
```

第11章　中断处理程序

除了第10章以外，到目前为止我们在内核中所做的所有工作都是为了响应进程的请求，或者是处理一个特殊的文件，发送 ioctl，或者是发出一个系统调用。但是内核的任务并不仅仅是为了响应进程请求。另一个任务也是同样重要的，那就是内核还需要和与机器相连接的硬件进行通信。

在CPU和计算机的其它硬件之间有两种相互交互的方式。第一种是 CPU向硬件发出命令，另一种是硬件需要告诉 CPU一些事情。第二种方式称之为中断，相比较而言，中断要难实现得多，这是因为它需要在硬件方便的时候进行处理，而不是在 CPU方便的时候去处理。一般来说，每个硬件设备都会有一个数量相当少的 RAM，如果在可以读它们的信息的时候用户没有去读，则这些信息将丢失。

在Linux下，硬件中断称为IRQ(即Interrupt Request的简称，中断请求)。IRQ分为两种类型：短的和长的，短IRQ是指需要的时间周期非常短，在这段时间内，机器的其它部分将被阻塞，并且不再处理其它的中断。而长 IRQ是指需要的时间相对长一些，在这段时间内也可能会发生别的中断(但是同一个设备上不会再产生中断)。如果有可能的话，中断处理程序还是处理长IRQ要好一些。

当CPU接收到一个中断时，它将停下手中所有的工作 (除非它正在处理一个更为重要的中断，在那种情况下，只有处理完那个更重要的中断以后， CPU才会去处理这个中断)，在栈中保存某些特定的参数，并调用中断处理程序。这就意味着在中断处理程序内部有些特定的事情是不允许的，因为系统处于一个未知的状态下。为了解决这个问题，中断处理程序应该做那些需要立刻去做的事情，通常是从硬件读一些信息或者向硬件发送一些信息，在稍后的某时刻再调度去做新信息的处理工作 (这称为“ 底半处理 ”)并返回。内核必须保证尽快调用底半处理 —— 在进行底半处理工作时，内核模块中允许做的所有工作都可以做。

要实现这一点，可以调用 request_irq，这样一来，在接收到相关 IRQ时，就可以调用用户的中断处理程序 (在Intel平台上共有16种IRQ)。request_irq接收IRQ编号、函数名称、标志、/proc/interrupts的名称以及传送给中断处理函数的一个参数。这里提到的标志包括 SA_SHIRQ和SA_INTERRUPT，前者表明用户愿意与其它中断处理程序分享 IRQ(通常是因为同一个 IRQ上有多个硬件设备)；而后者表明这是个高速中断。只有当这个 IRQ上还没有处理程序，或者两者都愿意共享这个IRQ时，request_irq函数才会成功。

接下来，我们从中断处理程序内部与硬件进行通信，并使用 tq_immediate和mark_bh(BH_IMMEDIATE)调用queue_task_irq，以调度底半处理。在版本2中我们不能使用标准的queue_task，这是因为中断有可能正好发生在其它的 queue_task中间。我们之所以需要使用mark_bh，是因为以前版本的Linux只有一个由32个底半处理所组成的队列，而现在它们中的一个(BH_IMMEDIATE)已经被用作底半处理的链接列表，这是为那些没有得到分配给它们的底半处理入口的驱动程序所准备的。

Intel体系结构的键盘

警告 本章余下的内容完全是与Intel相关的。如果读者不是在Intel平台上运行，这些内容将不能正常工作。读者甚至不要去编译这些代码。

在写本章的示例代码时我遇到了一个问题。一方面，为了让所给出的例子能有用，它必须可以在所有人的计算机上运行，且能得到有意义的结果。但是另一方面，内核中早已经为所有的通用设备准备了设备驱动程序，而那些设备驱动程序与我将要编写的驱动程序是不能共存的。最后我找到了解决的办法，就是为键盘中断写驱动程序，并且首先关闭常规的键盘中断驱动程序。因为它被定义成内核源文件 (特别是指drivers/char/keyboard.c)中的静态符号，所以没有办法恢复它。如果用户珍惜自己的文件系统的话，在执行 insmod命令插入这个代码以前，请先在另一个终端上执行 sleep 120;reboot。

该代码把它自己绑定在IRQ 1上，在Intel体系结构下，IRQ1是控制键盘的IRQ。这样，当它接收到键盘中断时，它读键盘的状态 (在程序中使用 inb (0x64)来实现)，并查看代码，该代码是由键盘所返回的值。然后在内核认为适合的时候，它立刻运行 got_char，该函数将给出所使用的键的代码 (即查看到的代码的前七位)，并给出该键是被按下 (如果第八位为 0)还是被松开(如果第八位为1)。

```
ex intrpt.c

/* intrpt.c - An interrupt handler. */


/* Copyright (C) 1998 by Ori Pomerantz */



/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h>   /* We're doing kernel work */
#include <linux/module.h>   /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* We want an interrupt */
#include <linux/interrupt.h>

#include <asm/io.h>


/* In 2.2.3 /usr/include/linux/version.h includes a
```

```
 * macro for this, but 2.0.35 doesn't - so I add it
 * here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif



/* Bottom Half - this will get called by the kernel
 * as soon as it's safe to do everything normally
 * allowed by kernel modules. */
static void got_char(void *scancode)
{
  printk("Scan Code %x %s.\n",
    (int) *((char *) scancode) & 0x7F,
    *((char *) scancode) & 0x80 ? "Released" : "Pressed");
}
/* This function services keyboard interrupts. It reads
 * the relevant information from the keyboard and then
 * schedules the bottom half to run when the kernel
 * considers it safe. */
void irq_handler(int irq,
                 void *dev_id,
                 struct pt_regs *regs)
{
  /* This variables are static because they need to be
   * accessible (through pointers) to the bottom
   * half routine. */
  static unsigned char scancode;
  static struct tq_struct task =
        {NULL, 0, got_char, &scancode};
  unsigned char status;

  /* Read keyboard status */
  status = inb(0x64);
  scancode = inb(0x60);

  /* Scheduale bottom half to run */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  queue_task(&task, &tq_immediate);
#else
  queue_task_irq(&task, &tq_immediate);
#endif
  mark_bh(IMMEDIATE_BH);
}



/* Initialize the module - register the IRQ handler */
int init_module()
{
```

```
    /* Since the keyboard handler won't co-exist with
     * another handler, such as us, we have to disable
     * it (free its IRQ) before we do anything. Since we
     * don't know where it is, there's no way to
     * reinstate it later - so the computer will have to
     * be rebooted when we're done.
     */
    free_irq(1, NULL);

    /* Request IRQ 1, the keyboard IRQ, to go to our
     * irq_handler. */
    return request_irq(
    1,  /* The number of the keyboard IRQ on PCs */
    irq_handler,  /* our handler */
    SA_SHIRQ,
    /* SA_SHIRQ means we're willing to have other
     * handlers on this IRQ.
     *
     * SA_INTERRUPT can be used to make the
     * handler into a fast interrupt.
     */
    "test_keyboard_irq_handler", NULL);
}


/* Cleanup */
void cleanup_module()
{
  /* This is only here for completeness. It's totally
   * irrelevant, since we don't have a way to restore
   * the normal keyboard interrupt so the computer
   * is completely useless and has to be rebooted. */
  free_irq(1, NULL);
}
```

第12章　对称多处理

要提高硬件的性能，最简单的(同时也是最便宜的)方法是把多个CPU放到一个主板上。实现这一点，可以有两种方法。一是使不同的CPU执行不同的任务(即非对称多处理)，或者使这些CPU并行运行，执行同样的任务(即对称多处理，简写为SMP)。进行非对称多处理非常需要对计算机将要执行的任务有特别的了解，而在像Linux这样的操作系统中，这是不可能的。另一方面，对称多处理相对要容易实现一些。这里所说的“相对容易”就是相对的意思——并不是指它真的容易实现。在对称多处理环境中，CPU共享同一个内存，这样做的后果是一个CPU中运行的代码可能会影响另一个CPU所使用的内存。用户不再可以肯定自己在前面一行中设置了值的那个变量仍然保持着那个值——另一个CPU可能在用户不注意的时候对那个变量进行了处理。很显然，要编出这样的程序是不可能的。

在进程编程时，一般来说上面这个问题就不是什么问题了，因为进程在某个时刻一般是只在一个CPU上运行的。而另一方面，内核则可以被运行在不同CPU上的不同进程所调用。

在版本2.0.x中，这个也不是什么问题，因为整个内核就是一个大的自旋锁(spinlock)。这意味着如果某个CPU在内核中而另一个CPU试图进入内核(假设是由于系统调用)，则后到的那个CPU必须等待，直到第一个CPU处理完，这使得Linux SMP比较安全，但是效率却相当低。

在版本2.2.x中，几个CPU可以同时位于内核中，这是模块编程人员需要留意的地方。我已经就SMP的问题向其它高手求助了，希望在本书的下一个版本中将会包含更多的信息。

第13章　常 见 错 误

在读者踌躇满志地准备动手编写内核模块以前，我还要提醒大家注意一些事情。如果是因为我没有警告过你而发生了不愉快的情况，请把你遇到的问题告诉我。我将把你买此书所付的钱全部还给你。

1) 使用标准库　不能这样做，在内核模块中用户只能使用内核函数，也就是可以在/proc/ksyms中找到的那些函数。

2) 关闭中断　用户可能需要在短时间内暂时关闭中断，那没什么问题，但是事后如果忘了打开它们，系统将会瘫痪，用户将不得不把电源拔掉。

3) 无视危险的存在　可能不需要提醒读者，但是最后我还是要说，只是为了以防万一。

附录A　2.0和2.2之间的差异

实际上我对整个内核了解得并不是很透彻，没有透彻到能够列出所有变化的地步。在转换本书例子的过程中(或者更确切地说是对Emmanuel Papirakis所做的转换进行修改)，我遇到了下面的差异，我把它们全部列举了出来，以便帮助模块编程人员 (特别是那些曾经学习过本书的前一版本，并熟悉我所使用的技术的读者)转换到新的版本。

希望进行转换工作的用户还可以访问如下网址：

http://www.atnf.csiro.au/~rgooch/linux/docs/porting_to_2.2.html.

1) asm/uaccess.h　如果需要使用put_user或者get_user，用户必须用#include包含这个文件。

2) get_user　在版本2.2中，get_user既可以接收指向用户内存的指针，也可以接收内核内存中的变量，以便填充用户信息。之所以这样，是因为在版本 2.2中get_user可以一次读两个或四个字节，如果我们所读的变量是两个字节或四个字节长的话， get_user可以读入它。

3) file_operations　该结构已经在open函数和close函数之间加入了一个刷新函数。

4) file_operations中的close函数　在版本2.2中，close函数返回一个整数，所以它可以失败。

5) file_operations中的read和write函数　这两个函数的头部已经改变了。在版本 2.2中，它们不再返回整数，而是返回一个 ssize_t类型的值，它们的参数列表也不同了。索引节点不再作为一个参数，但同时却加入了文件中的偏移量作为参数。

6) proc_register_dynamic　该函数已经不再存在了。取而代之的是，用户可以调用proc_register并把结构的索引结点字段置为0。

7) 信号　在任务结构中的信号不再是32位长的整数值，而是变成了由 _NSIG_WORDS整数组成的一个数组。

8) queue_task_irq　即使用户希望从中断处理程序内部调度任务来执行，他也应该使用queue_task，而不要使用queue_task_irq。

9) 模块参数　用户不再是只把模块参数定义为全局变量了。在 2.2中，用户必须同时使用MODULE_PARM来定义它们的类型。这是一个很大的改进，因为它允许模块可以接收以数字开头的字符串参数，而不会搞混淆。

10) 对称多处理　内核不再是一个大的自旋锁了，这就意味着内核模块在使用 SMP时必须小心。

附录B 其他资源

如果笔者愿意的话，可以轻易地在本书中再加入几章。可以加入一章介绍如何创建新的文件系统，或者介绍如何增加新的协议栈 (但这是不必要的 —— 因为读者很难找到Linux所不支持的协议栈)。当然还可以加入一些内核机制的解释，而这些机制我们并没有接触过，例如引导机制或者磁盘接口。

然而，笔者没有这么做，因为笔者写这本书的目的是为了探索内核模块编程的奥秘，希望教给用户一些内核模块编程的通用技术。对于那些对内核编程有着强烈兴趣的读者，笔者建议他们阅读如下网址的内核资源列表： http://jungla.dit.upm.es/~jmseyas /linux/kernel /hackers-docs.html。正如Linus所说的那样，学习内核的最佳方法就是自己阅读代码。

如果读者希望得到更多的短内核模块的例子，我建议阅读 Phrack杂志，即使用户对安全性不是太感兴趣，作为一个程序员也应该培养这方面的兴趣。 Phrack杂志上的内核模块都是一些很好的例子，介绍了用户可以在内核中所做的有关工作。而且这些例子都很短，读者不用费太大的劲就可以弄懂它们。

希望我能帮助读者成为一个更好的程序员，或者至少培养了在这方面的兴趣。如果读者写出了有用的内核模块，希望也能按照 GPL把它们出版出来，这样我也可以使用它们。

附录C　给出你的评价

这是一本“自由”的书。在GNU公共许可证所规定的内容以外，读者不受任何限制。如果读者想为这本书做点什么，我有以下几点建议：

- 给我寄一张明信片，地址是：

  Ori Pomerantz

  Apt. #1032

  2355 N Hwy 360

  Grand Prairie, TX 75050

  USA

  如果希望收到我的致谢回函，请在明信片上写清你的电子邮箱地址。
- 给自由软件组织捐献一些钱，或者一些时间 (更佳)。编写一个程序或者写作一本书，并按照GPL的条款出版，教别人怎样使用自由软件，例如Linux或者Perl。
- 向别人解释一下自私与社会生活是格格不入的，与帮助其它人是格格不入的。我很高兴我写了这本书，并且我相信出版这本书将来一定会获得回报。同时，我写这本书是为了帮助读者(如果我做到了的话)。记住，快乐的人常常比不快乐的人对自己更有用，而有才华的人常常比低能儿要有帮助的多。
- 高兴起来。如果我将遇见你，而且你是个愉快的人，这次会面将会给我留下美好的印象，而且愉快的个性也会使你变得对我更有用。

第1章　Linux操作系统

1991年3月，Linus Benedict Torvalds为他的AT386计算机买了一个多任务操作系统：Minix。他使用这个操作系统来开发自己的多任务系统，并称之为 Linux。1991年9月，他向Internet网上的其他一些Minix用户发电子邮件，发布了第一个系统原型，这样就揭开了 Linux工程的序幕。从那时起，有许多程序员都开始支持 Linux。他们增加设备驱动程序，开发应用程序，他们的目标是符合 POSIX标准。现在的Linux功能已经非常强大了，但是 Linux更吸引人的地方在于，它是免费的 (当然并不像免费啤酒那样，不是完全免费)。现在人们正在把Linux移植到其他平台上。

第2章　Linux内核

Linux的基础就是它的内核。用户可以替换某个库，或者将所有库都进行替换，但是只要Linux内核存在，它就还是Linux。内核包括设备驱动程序、内核管理、进程管理以及通信管理。内核高手总是遵循POSIX规则，该规则有时会使编程变得简单，有时会使它变得比较复杂。如果用户的程序在一个新的Linux内核版本上行为发生了变化，可能是因为实现了一个新的POSIX规则。如果读者想了解更多的关于Linux内核编程的信息，可以阅读《Linux Kernel Hacker ’s Guide》。

第3章 Linux libc包

Libc：ISO 8859.1；位于<linux/param.h>中；包括YP函数、加密函数、一些基本的影子过程(默认情况不包含)，……在libcompt中有一些为了保持兼容性而提供的老过程 (默认情况下不激活)；提供英文、法文，或者德文的错误信息；在 libcurses中有一些具有bsd 4.4lite兼容性的屏幕处理过程；在libbsd中有一些bsd兼容的过程；在libtermcap中有一些屏幕处理过程；在libdbm中有用于数据库管理的过程；在 libm中有数学过程；在crtO.o???中有执行程序的入口，在libieee???有一些字节信息(请别再笑话我了，能不能给我提供一些信息？)，在libgmon中是用户空间的配置信息。

我希望由某位Linux libc开发人员来编写本章。现在我能说的唯一的一句话是 a.out可执行格式将会变化成elf(可执行并可链接)格式(出版者注：这个变化已经发生了)，而后者又意味着在创建共享库方面的一个变化。当前这两种格式 (a.out和elf)都支持。

Linux libc包的绝大部分都是遵守库 GNU公共许可证的，尽管有些文件是遵守特殊的版权规定的，例如 crtO.o。对商业版本来说，这就意味着一个限制 ,即禁止静态链接可执行程序。在这里动态链接可执行程序又是一个特殊的例外。 FSF(自由软件基金会)的Richard Stallman说过：

在我看来，我们应该明确地允许发行不带伴随库的动态链接可执行程序，只要组成该可执行程序的对象文件按照第 5节的规定是不受限制的。… …，所以我决定现在就允许这样做。实际上，要更新LGPL将需要等到我有时间的时候，并且需要检查一下新版本。

第4章 系统调用

系统调用是向操作系统(内核)所作出的一次申请，请求操作系统做一次硬件 /系统相关的操作，或者是作一次只有系统才能做的操作。以Linux 1.2为例，它总共定义了140个系统调用。有些系统调用(如close ( ))是在Linux libc中实现的。这种实现常常需要调用一个宏，而该宏最后会调用syscall ( )。传送给syscall ( )的参数是系统调用的编号，在编号后面的参数是其他一些必需的变元。如果通过实现一个新的 libc库而更新了<sys/syscall.h>，则真正的系统调用编号可以在<linux/unistd.h>中找到。如果新的系统调用在libc中还没有代理程序，用户可以使用syscall ( )。下面给出一个例子，可以像下面一样使用 syscall ( )来关闭一个文件(不提倡)：

```
#include <syscall.h>

extern int syscall(int, ...);

int my_close(int filedescriptor)
{
   return syscall(SYS_close, filedescriptor);
}
```

在i386体系结构下，除了系统调用编号以外，系统调用只能带 5个以下的变元，这是由于受到了硬件寄存器数目的限制。如果读者是在另一个体系结构下运行 Linux的，可以检查一下<asm/unistd.h>中的_syscall宏，看看硬件支持多少个变元，或者说开发人员选择支持多少个变元。这些_syscall可以用于取代syscall ( )，但是并不提倡用户这么做，这是因为由这些宏扩展而成的完整的函数有可能在库中早已存在了。

所以，只有内核高手才能去使用 _syscall宏。为了说明这一点，下面给出一个使用 _syscall宏的close ( )的例子：

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

_syscall1(int, close, int, filedescriptor);
```

在_syscall1宏扩展以后，得到函数close ( )。这样，我们就有两个close ( )了，一个在libc中，另一个在我们的程序中。如果系统调用失败， syscall ( )或者_syscall宏的返回值是－1；而如果系统调用成功，则返回值将是 0或者更大的数值，如果系统调用失败，我们可以查看一下全局变量errno，看看到底发生了什么。

下面这些系统调用在BSD和Sys V中是可用的，但Linux 1.2不支持：

audit ( )、audition ( )、auditsvc ( )、fchroot ( )、getauid ( )、getdents ( )、getmsg ( )、mincore ( )、poll ( )、putmsg ( )、setaudit ( )、setauid ( )。

第5章 “瑞士军刀”：ioctl

ioctl代表输入/输出控制，它用于通过文件描述符来操作字符设备。 ioctl的格式如下所示：

ioctl (unsigned int fd, unsigned int request, unsigned long argument)

如果出错则返回值为－ 1，如果请求成功则返回值将大于或者等于 0，这就像其他系统调用一样。内核能区分特殊文件和普通文件。特殊文件一般可以在 /dev和/proc中找到。它们与普通文件的区别在于，它们隐藏了驱动程序的接口，并不是一个包含着文本或二进制数据的真正的(常规的)文件。这是Unix的特点，它允许用户对每一个文件都可以使用普通的读 /写操作。但是，如果用户想要对特殊文件或者普通文件进行更多的处理，用户可以使用……对了，就是ioctl。用户把ioctl用于特殊文件的机会比用于普通文件的机会要多得多，但是在普通文件上也可以使用ioctl。

第6章　Linux进程间通信

下面我们详细地介绍在Linux操作系统中实现的IPC(进程间通信)机制。

6.1　介绍

Linux IPC(进程间通信)机制为多个进程之间相互通信提供了一种方法。对 Linux C程序员来说，有许多种IPC的方法：

- 半双工Unix管道
- FIFO(命名管道)
- SYSV形式的消息队列
- SYSV形式的信号量集合
- SYSV形式的共享内存段
- 网络套接字(Berkeley形式)(本书不作介绍)
- 全双工管道(STREAMS管道)(本书不作介绍)

如果使用得当的话，这些机制将为任意 Unix系统(包括Linux)上的客户机/服务器开发提供一个坚强的框架。

6.2　半双工Unix管道

6.2.1　基本概念

简单地说，管道就是一种把一个进程的标准输出与另一个进程的标准输入相连接的方法。管道是最古老的IPC工具，自从Unix操作系统诞生以来，管道就已经存在了。它们提供了一种进程之间单向通信的方法(这就是术语“半双工”的由来)。

管道这一特征已经被广泛地使用了，甚至在 Unix命令行中(在Shell)中也用到了管道。

图3-6-1显示了建立一个管道，把 ls的输出当作sort的输入，以及把sort的输出当作lp输入进行处理的一系列过程。数据是通过一个半双工管道传输的，从管道的左边传输到管道的右边。

尽管我们中的大多数人都曾经非常频繁地在 shell脚本编程时使用管道，但在我们这么做的时候常常懒得想一下在内核一级到底发生了一些什么事。

在进程创建管道时，内核创建两个文件描述符，以供管道使用。其中一个描述符用于允许输入管道的路径(写)；而另一个用于从管道获得数据 (读)。如果仅仅是这样，管道将没有什么实用性，创建管道的进程只能用该管道与它自己通信。下图显示了在创建管道以后进程和内核的状态。

注意　出版者注：作者尚未提供3-6-1图。

从上图可以很容易地看出这两个文件描述符是如何连接在一起的。如果进程通过管道

(fd0)发送数据，它可以从fd1获得(读)那个信息。然而，相对上面这个简单的草图来说，用户希望实现的目标要大得多。当一个管道把进程与它自己相连接时，在管道中传输的数据需要经过内核。特别是在Linux下，管道实际上是用一个合法索引节点在内部表示的。当然，这个索引节点自己也驻留在内核中，并且不在任何物理文件系统的范围之内，这个特点将为用户提供一些方便的I/O方法，稍后读者就能了解到这一点。

到目前为止，管道显得毫无用处。然而，如果我们只想和自己通信，我们为什么要不辞辛苦地去创建管道呢？就目前来说，创建进程一般都是产生一个子进程。因为子进程将从父母那里继承所有打开的文件描述符，我们现在已经得到了一个多进程通信的框架(在父与子之间通信)。我们上面这个简单的草图经过升级以后如图3-6-2所示。

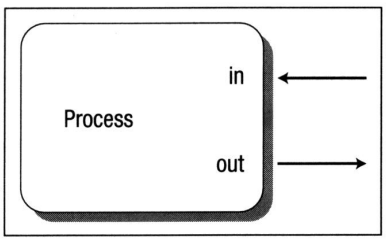

图 3-6-2

现在读者可以看到，两个进程都能访问组成管道的那两个文件描述符。就在这个时候需要作出一个关键的决定，即希望数据向哪个方向传输？是子进程向父进程发送信息，还是反过来？一旦决定以后，两个进程彼此都同意这个决定，并把它们各自所不关心的管道的某一端关掉。为了方便进行介绍，这里假设子进程进行某些处理操作，并通过管道把消息发送回父进程。经过修改以后，图3-6-2变成了图3-6-3。

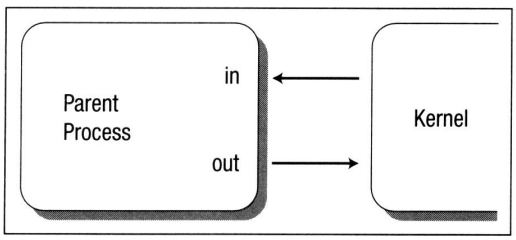

图 3-6-3

管道的创建工作到此全部完成！唯一剩下要做的事就是使用管道了。为了直接访问管道，可以使用那些用来完成低级文件I/O的系统调用(记住，管道实际上在内部是表示成合法的索引节点的)。

为了把数据发送到管道，我们使用 write ( )系统调用；而为了从管道接收数据，我们可以使用 read ( )系统调用。记住，低级文件I/O系统调用是使用文件描述符的！然而，读者需要注意的是，有些特定的系统调用，例如 lseek ( )，对管道的文件描述符是无效的。

6.2.2 用C语言创建管道

与简单的shell例子比较起来，用高级编程语言 C创建管道要常见一些。为了用 C语言创建一个简单的管道，用户可以使用 pipe ( )系统调用。这个系统调用只需要一个变元，它是由两个整数组成的一个数组。如果调用成功，该数组将会包含管道所使用的两个新的文件描述符。在创建管道以后，进程一般会产生一个新进程 (记住子进程将继承打开的文件描述符)。

```
SYSTEM CALL: pipe();

  PROTOTYPE: int pipe( int fd[2] );
    RETURNS: 0 on success
             -1 on error: errno = EMFILE (no free descriptors)
                                  EMFILE (system file table is full)
                                  EFAULT (fd array is not valid)
```

```
NOTES: fd[0] is set up for reading, fd[1] is set up for writing
```

数组中的第一个函数(元素0)是为了读操作而创建和打开的，而第二个函数 (元素1)是为了写操作而创建和打开的。直观地说， fd1的输出变成了fd0的输入。再说一遍，通过管道传输的所有数据都将经过内核。

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

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

记住，在C语言中，数组的名称实际上变成了指向它的第一个成员的指针，在上面的程序中，fd相当于&fd[0]。一旦创建了管道，就可以产生子进程了：

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

如果父进程希望从子进程接收到数据，它应该关闭 fd1，而子进程应该关闭fd0。如果父进程希望把数据发送到子进程，它应该关闭 fd0，而子进程应该关闭fd1。因为描述符是在父进程和子进程之间共享的，所以一定要确定我们所关闭的管道的那一端是我们所不关心的。从技术的角度来看，如果管道中不需要的那一端没有显式地关闭的话，将永远不会返回 EOF。

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

```
main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
        .
        .
}
```

正如以前所提到的那样，管道一旦创建好以后，它的文件描述符就可以像普通文件的文件描述符一样处理了。

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
```

```
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, strlen(string));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

常常会有这样的情况，用户把子进程的文件描述符复制到标准输入或输出。这样，子进程可以用exec ( )来执行另一个程序，而那个程序将得到标准的输入或输出流。请看下面的 dup ( )系统调用：

```
SYSTEM CALL: dup();
  PROTOTYPE: int dup( int oldfd );
    RETURNS: new descriptor on success
             -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                  EBADF (newfd is out of range)
                                  EMFILE (too many descriptors for the process)
Notes: The old descriptor is not closed! Both may be used interchangeably
```

尽管老的文件描述符和新创建的文件描述符可以互换使用，但一般首先关闭一个标准的输入/输出流。系统调用dup ( )使用编号最低且未被使用的描述符作为新的描述符。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close up standard input of the child */
                close(0);

                /* Duplicate the input side of pipe to stdin */
```

```
        dup(fd[0]);
        execlp("sort", "sort", NULL);
        .
}
```

尽管文件描述符0(stdin)被关闭了，对dup ( )的调用会把管道的输入描述符 (fd0)复制到它的标准输入上，然后再调用 execlp ( )，以便让排序程序的文本段 (代码)覆盖子进程的文本段 (代码)。因为新执行的程序从它们的创建者那里继承了标准的输入 /输出流，它实际上就继承了管道的输入端作为自己的标准输入！这样一来，原来的父进程发送到管道的任何数据都将传送到排序工具那里。

用户还可以使用另一个系统调用 dup2 ( )。这个特殊的系统调用最早是由第 7版的Unix提供的，在BSD的发行版本中一直沿续了下来，现在已被 POSIX标准列入规范要求。

```
SYSTEM CALL: dup2();

PROTOTYPE: int dup2( int oldfd, int newfd );
  RETURNS: new descriptor on success
           -1 on error: errno = EBADF (oldfd is not a valid descriptor)
                                EBADF (newfd is out of range)
                                EMFILE (too many descriptors for the process)
```

注意 旧的文件描述符是用dup2 ( )关闭的！

使用这个特殊的调用，用户可以在一次系统调用中完成关闭操作以及文件描述符的复制工作。另外，该系统调用保证是原子性的，这就意味着它永远不会被一个突如其来的信号所中断。在把控制权还给内核进行信号调度以前，整个操作将会全部完成。如果用户使用原来的dup ( )系统调用，则程序员在调用它之前需要执行一次 close ( )操作。这就需要进行两次系统调用，在两次系统调用之间的短暂时间里，存在一点点危险的可能性。如果在那个稍纵即逝的瞬间到达了一个信号，则文件描述符的复制将会失败。当然， dup2 ( )会为用户解决这个问题。

Consider:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Close stdin, duplicate the input side of pipe to stdin */
                dup2(0, fd[0]);
                execlp("sort", "sort", NULL);
                .
                .
        }
```

6.2.3 简便方法

如果读者认为以上这种方法是一种创建和利用管道的间接方法，觉得它不够直观和简便

的话，可以选择使用下面这种方法：

```
LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type);
  RETURNS: new file stream on success
           NULL on unsuccessful fork() or pipe() call
Notes: creates a pipe, and performs fork/exec operations using "command."
```

以上这个标准库函数通过在内部调用 pipe ( )创建了一个半双工管道，然后它生成一个子进程，执行Bourne shell，并在shell中执行“ command ”变元。数据流动的方向由第二个变元“ type ”所决定。它的值可以是“ r ”或者“ w ”。“ r ”代表“ 读 ”，而“ w ”代表“ 写 ”，但不能既为读又为写！在Linux下，管道打开的方式是由“ type ”变元的第一个字符决定的。这样一来，如果用户把“ type ”的值置为“ rw ”，则管道只能以“ 读 ”方式打开。

可能读者会认为这个库函数非常实用，但是读者要知道，这么做所付出的代价也不小。在使用pipe ( )系统调用并自己处理fork/exec操作时，用户对管道有着很好的控制能力，但是使用这个库函数使用户失去了控制权。然而，由于这里直接使用 Bourne shell，所以允许在“ command ”变元中使用shell元字符扩展(包括通配符)。

使用popen ( )创建的管道必须用 pclose ( ) 关闭。到现在为止，读者可能已经意识到popen/pclose与标准文件流I/O函数fopen ( )和fclose ( )有着惊人的相似性。

```
LIBRARY FUNCTION: pclose();

PROTOTYPE: int pclose( FILE *stream );
  RETURNS: exit status of wait4() call
           -1 if "stream" is not valid, or if wait4() fails
Notes: waits on the pipe process to terminate, then closes the stream.
```

在由popen ( )所生成的进程上， pclose ( )函数执行一次wait4 ( )命令。当它返回时，它将摧毁管道和文件流。在这里读者可以再次体会到， pclose ( )和普通的基于流的文件I/O fclose ( )函数非常相似。

请看下面的例子。该例为排序命令打开了一个管道，接着开始处理一个字符串数组的排序：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: popen1.c
 *****************************************************************************/

#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                   "charlie", "delta"};
```

```
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen("sort", "w")) == NULL)
{
        perror("popen");
                exit(1);
        }

        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }

        /* Close the pipe */
        pclose(pipe_fp);

        return(0);
}
```

因为popen ( )使用了shell来完成它的任务，所以可以使用所有的 shell扩展字符和元字符！此外，popen ( )还可以利用一些更高级的技术，例如重定向，甚至还可以利用输出管道技术。请看下面的实例调用：

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

下面的这个小程序是popen ( )的另一个例子，它打开了两个管道 (一个连接到ls命令，另一个连接到sort命令)：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen2.c
 *************************************************************************/

#include <stdio.h>

int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }
```

```
    /* Create one way pipe line with call to popen() */
    if (( pipeout_fp = popen("sort", "w")) == NULL)
    {
            perror("popen");
            exit(1);
    }

    /* Processing loop */
    while(fgets(readbuf, 80, pipein_fp))
            fputs(readbuf, pipeout_fp);

    /* Close the pipes */
    pclose(pipein_fp);
    pclose(pipeout_fp);

    return(0);
}
```

为了对popen ( )进行最后的说明。我们创建了一个普通的程序。该程序在执行的命令和文件名称之间打开一个管道：

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: popen3.c
 *************************************************************************/

#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];

        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }

        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }
/* Create one way pipe line with call to popen() */
if (( pipe_fp = popen(argv[1], "w")) == NULL)
{
        perror("popen");
```

```
        exit(1);
    }

    /* Processing loop */
    do {
        fgets(readbuf, 80, infile);
        if(feof(infile)) break;

        fputs(readbuf, pipe_fp);
    } while(!feof(infile));

    fclose(infile);
    pclose(pipe_fp);

    return(0);
}
```

使用以下的命令试着运行这个程序：

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

6.2.4 管道的原子操作

如果说一个操作是“原子性”的，那么就不能以任何理由来中断该操作。整个操作都是一次性完成的。在/usr/include/posix1_lim.h中，POSIX标准对管道上的原子操作的最大缓冲区大小是这样规定的：

```
#define _POSIX_PIPE_BUF         512
```

在一次原子操作中，最多可以把512个字节写到管道，或者最多从管道读取512个字节。如果超过了这个范围，操作就将被分开，不能称之为原子操作了。然而，在Linux下，原子操作的限制在“linux/limits.h”中是这样定义的：

```
#define PIPE_BUF        4096
```

读者不难看出，Linux调整了POSIX规定的很少数目的字节数，甚至可以说是大大地调整了这个数目。当包含多个进程时，管道的原子操作变得非常重要。例如，如果写到管道的字节数目超过了一个操作所能达到的原子性限制，并且有多个进程都在写管道，则数据将会被“交叉”或者“分块”。换句话说，一个进程可能会在另一个进程进行写的时候把数据插入到管道中。

6.2.5 关于半双工管道需要注意的几个问题

- 通过创建两个管道，并在子进程中正确地重新分配好文件描述符，用户就可以创建双向管道。
- pipe ( )调用必须在调用fork ( )以前进行，否则子进程将无法继承文件描述符 (对popen ( )来说也是如此)。

• 使用半双工管道互相连接的任意进程必须位于一个相关的进程家族里。因为管道必须受到内核的限制，所以如果进程没有在管道创建者的家族里面，则该进程将无法访问管道。这一点是与命名管道(FIFO)有区别的。

6.3 命名管道

6.3.1 基本概念

命名管道的工作方式与普通的管道非常相似，但也有一些明显的区别。

• 在文件系统中命名管道是以设备特殊文件的形式存在的。
• 不同家族的进程可以通过命名管道共享数据。
• 因为所有的I/O工作都是由共享进程处理的，文件系统中保留命名管道是为了将来使用。

6.3.2 创建FIFO

有许多种方法可以创建命名管道。其中前两者可以直接用 shell来完成。

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

上面这两个命令执行同样的操作，只有一个地方存在差异。 mkinfo命令提供了一个挂钩，可以在创建 FIFO文件之后直接改变它的许可。而如果用户使用 mknod，则需要立刻调用chmod命令。

在物理文件系统中，用户可以很快找到 FIFO文件，因为在长目录列表中， FIFO文件有一个“ p ”指示符。

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

同时读者还可以注意到，在文件名的后面有一条竖线 (“ 管道符号 ”)。由此不难体会到运行Linux无所不在的好处。

为了用C语言创建FIFO，用户可以使用mknod ( )系统调用：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

the file “/tmp/MYFIFO” is created as a FIFO file. The request
ough they are affected by the umask setting as follows:

```
final_umask = requested_permissions & ~original_umask
```

rick is to use the umask() system call to temporarily zap the

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

关于mknod ( )更详细的介绍，用户可以参考 man的帮助信息，现在请看用C语言创建FIFO的一个简单的例子：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

在上面这个例子中，文件 /tmp/MYFIFO是当作一个FIFO文件而创建的。所申请的访问许

可权限是“ 0666 ”，尽管该权限会受到如下定义的掩码的影响：

```
final_umask = requested_permissions & ~original_umask
```

一个常见的技巧是使用umask ( )系统调用暂时屏蔽掉掩码的值：

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

此外， mknod ( )的第三个参数可以忽略，除非用户创建的是设备文件，在那种情况下，mknod ( )应该指明设备文件的主编号和从编号。

6.3.3 FIFO操作

对FIFO来说， I/O操作与普通的管道 I/O操作基本上是一样的，但也存在着一个主要的区别。在FIFO中，必须使用一个“ open ”系统调用或者库函数来物理地建立联接到管道的通道。而对于半双工管道而言，这是不必要的，因为管道是驻留在内核中的，而不是驻留在物理文件系统上的。在下面的例子中，我们将把管道当作一个流来看待，使用 fopen ( )来打开它，并使用fclose ( )来关闭它。

请看下面这个简单的服务器进程：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoserver.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
```

```
        }

        return(0);
}
```

因为FIFO是默认阻塞的，所以在编译了这个服务器进程之后，可以在后台运行它：

```
$ fifoserver&
```

我们稍后就将讨论FIFO的阻塞动作。首先，请看上面这个服务器进程的简单的客户前端，如下所示：

```
/*****************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: fifoclient.c
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

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

6.3.4 FIFO上的阻塞动作

一般来说FIFO总是处于阻塞状态。换句话说，如果 FIFO是为了读操作而打开的，则进程将“阻塞”，直到某些其他进程打开该FIFO并且写数据。这个阻塞动作反过来也是成立的。如果用户不希望发生阻塞，可以在 open ( )调用中使用O_NONBLOCK标志，以关闭默认的阻塞动作。

在上面那个简单的服务器进程的例子中，我们只是把它放在后台，让它在那里进行它的

阻塞动作。用户还可以选择跳转到另一个虚拟控制台，并运行客户前端，来回切换以查看所导致的动作。

6.3.5 SIGPIPE信号

最后需要注意的一点是，管道必须有人读并有人写。如果某个进程企图往管道中写数据，而没有进程去读该管道，则内核将向该进程发送 SIGPIPE信号。当在管道操作中涉及到两个以上的进程时，这个信号是非常必要的。

6.4 系统V IPC

6.4.1 基本概念

在Unix系统V中，AT&T引入了三种新的IPC方法(消息队列、信号量和共享内存)。POSIX委员会还没有完全制定好这些工具的标准，但大多数实现已经支持这三种 IPC方法。此外，Berkeley (BSD)使用套接字来作为其IPC的主要方法，而不是采用系统V的方法。Linux可以同时使用这两种IPC方法(BSD和系统V)。在稍后的某章中将会介绍套接字。

在Linux中，系统V IPC的实现是由Krishna Balasubramanian完成的，他的电子邮箱地址是balasub@cis.ohio_state.edu。

1. IPC标识符

每一个IPC对象都有一个独一无二的IPC标识符与之联系。这里所说的“ IPC对象 ”是指单个的消息队列、信号量集合或者共享内存段。在内核中这个标识符可以用于唯一地标识一个IPC对象。例如，为了访问某个特定的共享内存段，用户唯一需要的是赋给那个内存段的独一无二的ID值。

标识符的唯一性是与对象的类型相关的。为了说明这一点，假设有一个数字标识符“ 12345 ”。当然不可能会有两个消息队列都对应到这个标识符。但还是存在一种明显的可能性，即某个消息队列和某个共享内存段都具有这个数字标识符。

2. IPC关键字

为了获得一个唯一的ID号。必须使用关键字，客户进程和服务器进程双方都必须同意这个关键字，这是构造应用程序的客户/服务器框架的第一步。

当你给某人打电话时，你必须知道他的电话号码。此外，电话公司必须知道如何把你发出的呼叫转达到它最终的目的地。一旦对方提起电话，开始作出响应，联接就建立起来了。

在系统V IPC方法中，“ 电话 ”直接对应到使用的对象的类型，而“ 电话公司 ”，或者说寻找路由的方法，则与IPC关键字有着直接的联系。

关键字对不同的对象可以具有相同的值，它是通过把关键字的值硬编码进应用程序来实现的。这样做有一个缺点，就是关键字可能早已经在使用了。用户通常可以使用 ftok ( )函数来为客户进程和服务器进程生成关键字的值。

```
LIBRARY FUNCTION: ftok();
PROTOTYPE: key_t ftok ( char *pathname, char proj );
  RETURNS: new IPC key value if successful
           -1 if unsuccessful, errno set to return of stat() call
```

ftok ( )返回的关键字值是这样生成的：把索引节点号、第一个变元中文件的次设备编号，以及第二个变元中的工程标识符 (一个字符)组合起来，这就是返回值。这样的返回值不能保证是唯一的，但应用程序可以查找到是否存在冲突，如果有的话则重新试着生成关键字。

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在上面这个短的代码片段中，目录 /tmp/myapp与 ’ a ’ 这个标识符组合在一起。另一个通用的例子是使用当前目录：

```
key_t   mykey;
mykey = ftok(".", 'a');
```

采取什么样的关键字生成算法完全是由应用程序编程人员决定的。只要可以防止竞争条件、死锁等等，无论采取什么方法都可以。为了方便进行介绍，这里将使用 ftok ( )方法。如果用户能确定每个客户进程都是从独一无二的“ home ”目录运行的，则所产生的关键字应该能够满足需要。

无论关键字的值是如何产生的，它将用于后续的 IPC系统调用中，用来创建 IPC对象或者获取对它的访问权限。

3. ipcs命令

ipcs命令可以用来获取所有系统 V IPC对象的状态。这个工具的 Linux版本也是由Krishna Balasubramanian编写的。

```
ipcs      -q:    Show only message queues
ipcs      -s:    Show only semaphores
ipcs      -m:    Show only shared memory
ipcs --help:     Additional arguments
```

在默认的情况下，所有三种类型的 IPC对象都将显示出来。请看下面这个 ipcs命令的输出示例：

```
------ Shared Memory Segments --------
shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
semid      owner      perms      nsems      status
 ------ Message Queues --------
 msqid      owner      perms      used-bytes  messages
 0          root       660        5           1
```

从上面这个输出中读者可以找到一个具有标识符“ 0 ”的消息队列，它是由 root用户所拥有的，具有八进制形式的访问权限 660，即-rw-rw--。在此队列中有一条消息，而那个消息它的大小总共为5个字节。

ipcs命令是一个功能非常强大的工具，它使用户可以研究内核中 IPC对象的存储机制。读者应该学好它，用好它。

4. ipcrm命令

ipcrm命令可以用来从内核删除一个 IPC对象。IPC对象可以通过在用户代码中调用系统调用来删除(稍后将介绍这方面的内容)。然而，用户常常会需要手工删除 IPC对象，尤其是在开发环境下，ipcrm命令的用法非常简单：

```
ipcrm <msg | sem | shm>  <IPC ID>
```

只需指定想要删除的对象是消息队列(msg)、信号量集合(sem)还是共享内存段(shm)即可。IPC ID号可以使用ipcs命令来获得。用户必须指明对象的类型，因为在相同类型的对象之中，标识符是各不相同的(参见前面的讨论)。

6.4.2 消息队列

1. 基本概念

消息队列的最佳定义是：内核地址空间中的内部链表。消息可以顺序地发送到队列中，并以几种不同的方式从队列中获取。当然，每个消息队列都是由 IPC标识符所唯一标识的。

2. 内部和用户数据结构

要完成理解象系统 V IPC这样复杂的问题，关键是要彻底熟悉内核的几个内部数据结构。甚至对那些最基本的操作来说，直接访问这些结构中的某几个结构也是必要的，而其他的结构则停留在一个更低的级别上。

3. 消息缓冲区

我们要介绍的第一个结构是 msgbuf结构。这个特殊的数据结构可以认为是消息数据的模板。虽然定义这种类型的数据结构是程序员的职责，但是读者绝对必须知道实际上存在msgbuf类型的结构。在linux/msg.h中，它的定义如下所示：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;         /* type of message */
    char mtext[1];      /* message text */
};
```

在msgbuf结构中有两个成员：

• mtype —— 它是消息类型，以正数来表示。这个数必须为一个正数！

• mtext —— 它就是消息数据。

因为用户可以给特定的消息赋予一个类型，这样用户就能够在一个消息队列上进行消息的多路传送。例如，用户必须赋予客户进程一个幻数，这个幻数可以用作服务器进程所发送的消息的消息类型。而服务器本身也可以使用其他的一些数，客户可以使用这些数向它发送消息。在另一种情况下，应用程序可以把错误消息标记为具有消息类型号 1的消息，而请求消息的类型为2，等等。这种可能性是不胜枚举的。

读者需要注意的另一点是，不要被被消息数据元素 (mtext)的描述性太强的名称所误导，这个域并不是只能存放字符数据，它还能存放任意形式的任意数据。因为应用程序编程人员可以重新定义msgbuf结构，所以mtext域实际上是随机性很强的。请看下面的例子：

```
struct my_msgbuf {
        long    mtype;          /* Message type */
        long    request_id;     /* Request identifier */
        struct  client info;    /* Client information structure */
};
```

在这个定义中也有消息类型，这点与以前的定义一样，但是结构的剩余部分则被替换成两个其他的元素，其中有一个是另一种结构！这就是消息队列的可爱之处。这样一来，不论是哪种类型的数据，内核均无需作转换工作，任意信息均可以发送。

然而，对于给定消息的最大的大小，确实存在一个内部的限制。在 Linux中，它在

linux/msg.h中是这样定义的：

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

消息总的大小不能超过4,056个字节，这其中包括mtype成员，它的长度是4个字节(long类型)。

4. 内核msg结构

内核把消息队列中的每个消息都存放在 msg结构的框架中。该结构是在 linux/msg.h中定义的，如下所示：

```
/* one msg structure for each message */
struct msg {
    struct msg *msg_next;   /* next message on queue */
    long  msg_type;
    char *msg_spot;         /* message text address */
    short msg_ts;           /* message text size */
};
```

- msg_next —— 这是一个指针，指向消息队列中的下一个消息。在内核寻址空间中，它们是当作一个链表存储的。
- msg_type —— 这是消息类型，它的值是在用户结构 msgbuf中赋予的。
- msg_spot —— 这是一个指针，指向消息体的开始处。
- msg_ts —— 这是消息文本(消息体)的长度。
- 内核msgid_ds结构 —— IPC对象分为三类，每一类都有一个内部数据结构，该数据结构是由内核维护的。对于消息队列而言，它的内部数据结构是 msqid_ds结构。对于系统上创建的每个消息队列，内核均为其创建、存储和维护该结构的一个实例。该结构在linux/msg.h中定义，如下所示：

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

虽然读者可能很少会关心这个结构的大部分成员，为了叙述的完整性，下面还是对每个成员都给出一个简短的介绍：

- msg_perm —— 它是ipc_perm结构的一个实例， ipc_perm结构是在linux/ipc.h中定义的。该成员存放的是消息队列的许可权限信息，其中包括访问许可信息，以及队列的创建者

的有关信息(如uid等等)。

- msg_first —— 链接到队列中的第一个消息(列表头部)。
- msg_last —— 链接到队列中的最后一个消息(列表尾部)。
- msg_stime —— 发送到队列的最后一个消息的时间戳(time_t)。
- msg_rtime —— 从队列中获取的最后一个消息的时间戳。
- msg_ctime —— 对队列进行最后一次变动的时间戳(稍后将作详细介绍)。
- wwait和rwait —— 是两个指针，指向内核的等待队列。如果消息队列上的某次操作使进程进入睡眠状态，则需要使用这两个成员！ (类似的操作包括：队列已满，进程等待一次打开操作)。
- msg_cbytes —— 在队列上所驻留的字节的总数(即所有消息的大小的总和)。
- msg_qnum —— 当前处于队列中的消息数目。
- msg_qbytes —— 队列中能容纳的字节的最大数目。
- msg_lspid —— 发送最后一个消息的进程的PID。
- msg_lrpid —— 接收最后一个消息的进程的PID。

5. 内核ipc_perm结构

内核把IPC对象的许可权限信息存放在ipc_perm类型的结构中。例如在前面描述的某个消息队列的内部结构中， msg_perm成员就是ipc_perm类型的，它的定义是在文件linux/ipc.h中，如下所示：

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

以上所有的成员都具有相当的自扩展性。对象的创建者以及所有者 (它们可能会有不同)的有关信息，以及对象的IPC关键字都是存放在该结构中的。八进制形式的访问模式也是存放在这里的，它是以一种无符号短整型的形式存储的。最后，时间片使用序列编号存放在最后面，每次通过系统调用关闭IPC对象(摧毁)时，这个值将被增加一，至多可以增加到能驻留在系统中的IPC对象的最大数目。用户需要关心这个值吗？答案是“不”。

有关这个问题，在Richard Stevens所著的《Unix Network Programming》一书的第125页中作了精辟的讨论。该书还介绍了ipc_perm结构的存在和行为在安全性方面的原因。

6. 系统调用：msgget ( )

为了创建一个新的消息队列，或者访问一个现有的队列，可以使用系统调用 msgget ( )。

```
SYSTEM CALL: msgget();

PROTOTYPE: int msgget ( key_t key, int msgflg );
  RETURNS: message queue identifier on success
           -1 on error: errno = EACCESS (permission denied)
```

```
EEXIST (Queue exists, cannot create)
EIDRM (Queue is marked for deletion)
ENOENT (Queue does not exist)
ENOMEM (Not enough memory to create queue)
ENOSPC (Maximum queue limit exceeded)
```

msgget ( )的第一个变元是关键字的值(在我们的例子中该值是调用 ftok ( )的返回值)。这个关键字的值将被拿来与内核中其他消息队列的现有关键字值相比较。比较之后，打开或者访问操作依赖于msgflg变元的内容。

• IPC_CREAT —— 如果在内核中不存在该队列，则创建它。

• IPC_EXCL —— 当与IPC_CREAT一起使用时，如果队列早已存在则将出错。

如果只使用了IPC_CREAT, msgget ( )或者返回新创建消息队列的消息队列标识符，或者会返回现有的具有同一个关键字值的队列的标识符。如果同时使用了 IPC_EXCL和IPC_CREAT，那么将可能会有两个结果。或者创建一个新的队列，或者如果该队列存在，则调用将出错，并返回－1。IPC_EXCL本身是没有什么用处的，但在与IPC_CREAT组合使用时，它可以用于保证没有一个现存的队列为了访问而被打开。

有个可选的八进制许可模式，它是与掩码进行 OR操作以后得到的。这是因为从功能上讲，每个IPC对象的访问许可权限与Unix文件系统的文件许可权限是相似的！

下面我们创建一个包装程序，它可用于打开或者创建消息队列：

```
int open_queue( key_t keyval )
{
        int     qid;

        if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(qid);
}
```

注意程序中使用了显式的许可权限 0660。这个小函数或者返回消息队列标识符 (int类型)，或者出错返回－1。必须出错返回－1。必须把关键字的值传给它，这是它唯一的变元。

7. 系统调用：msgsnd ( )

一旦获得了队列标识符，用户就可以开始在该消息队列上执行相关操作了。为了向队列传递消息，用户可以使用 msgsnd系统调用：

```
  SYSTEM CALL: msgsnd();

  PROTOTYPE: int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg
);
    RETURNS: 0 on success
             -1 on error: errno = EAGAIN (queue is full, and IPC_NOWAIT was
asserted)
                                  EACCES (permission denied, no write permis-
sion)
```

```
                                        EFAULT (msgp address isn't accessable -
invalid)

                                        EIDRM  (The message queue has been removed)
                                        EINTR  (Received a signal while waiting to
write)

                                        EINVAL (Invalid message queue identifier,
nonpositive

                                                message type, or invalid message
size)

                                        ENOMEM (Not enough memory to copy message
buffer)
```

msgsnd的第一个变元是队列标识符，它是前面调用 msgget获得的返回值。第二个变元是msgp，它是一个指针，指向我们重新定义和载入的消息缓冲区。 msgsz变元则包含着消息的大小，它是以字节为单位的，其中不包括消息类型的长度 (四个字节长)。

msgflg变元可以设置为0(忽略)，也可以设置为IPC_NOWAIT。如果消息队列已满，则消息将不会被写入到队列中，控制权将被还给调用进程。如果没有指定 IPC_NOWAIT，则调用进程将被中断(阻塞)，直到可以写消息为止。

下面创建另一个发送消息的包装程序：

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

这个小函数接收到一个地址 (qbuf)，并试着把驻留在那个地址中的消息发送到消息队列标志符(qid)所指定的消息队列中， qid也是作为一个参数传送给该函数的。下面这个示例代码片段将用到前面创建的那两个包装函数：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>

main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
                int     request;        /* Work request number */
```

```
        double  salary;         /* Employee's salary */
    } msg;

    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');

    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
            perror("open_queue");
            exit(1);
    }

    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */

    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
            perror("send_message");
            exit(1);
    }
}
```

在创建或者打开了消息队列以后，接下去用测试数据装填消息缓冲区 (注意这里没有使用字符数据，这是为了说明我们有关传送二进制信息的观点)，调用send_message，它会把消息传送到消息队列中。

现在既然消息队列有一个消息，用户可以使用 ipcs命令来查看队列的状态。下面将介绍如何真正从队列中获取消息。为了做到这点，可以使用系统调用 msgrcv ( )：

```
  SYSTEM CALL: msgrcv();
  PROTOTYPE: int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long
mtype, int msgflg );
    RETURNS: Number of bytes copied into message buffer
             -1 on error: errno = E2BIG  (Message length is greater than msgsz,
no MSG_NOERROR)
                                  EACCES (No read permission)
                                  EFAULT (Address pointed to by msgp is in-
valid)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINTR  (Interrupted by arriving signal)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  ENOMSG (IPC_NOWAIT asserted, and no message
exists
                                         in the queue to satisfy the request)
```

显然，第一个变元是用来指定在消息获取过程中所使用的队列的 (该值是由前面调用msgget得到的返回值)。第二个变元(msgp)代表消息缓冲区变量的地址，获取的消息将存放在这里。第三个变元(msgsz)代表消息缓冲区结构的大小，不包括 mtype成员的长度。再说一遍，可以使用下面的公式来计算大小：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个变元(mtype)指定要从队列中获取的消息的类型。内核将查找队列中具有匹配类型的最老的消息，并把它的一个拷贝返回到由 msgp变元所指定的地址中。这里存在一种特殊的情况：如果mtype变元传送一个为零的值，则将返回队列中最老的消息，不管该消息的类型是什么。

如果把IPC_NOWAIT作为一个标志传送给该系统调用，而队列中没有任何消息。则该次调用将会向调用进程返回ENOMSG。否则，调用进程将阻塞，直到满足 msgrcv ( )参数的消息到达队列为止。如果在客户等待消息的时候队列被删除了，则返回 EIDRM。如果在进程阻塞并等待消息的到来时捕获到一个信号，则返回 EINTR。

请看下面这个从队列中获取消息的包装函数：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* The length is essentially the size of the structure minus
sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);

        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }

        return(result);
}
```

在成功地从队列中获取了一个消息之后，队列中的消息项目将被摧毁。

msgflg变元中的MSG_NOERROR位提供了一些其他的功能。如果物理消息数据的大小比msgsz要大，同时又声明了MSG_NOERROR位，则消息将被截断，并只返回 msgsz个字节。一般来说，msgrcv ( )系统调用将返回－1(E2BIG)，并且该消息将保留在队列中，以供以后获取。可以利用这个特点来创建另一个包装程序，该程序使用户能够窥探队列的内部，看看是否有满足请求的消息到达：

```
int peek_message( int qid, long type )
{
        int     result, length;

        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }

        return(FALSE);
}
```

在上面这个程序中，细心的读者可以发现缺少缓冲区地址和长度。在这种特殊的情况下，

我们希望调用出错。然而，我们可以检测 E2BIG的返回，它显示了确实存在匹配所申请的类型的消息。如果成功，则包装程序将返回 TRUE，否则将返回 FALSE。同时请读者注意IPC_NOWAIT标志的使用，它防止了前面介绍过的阻塞行为的发生。

系统调用：msgctl ( )

通过前面介绍的那些包装程序的开发，读者现在应该知道怎样在应用程序中简单地，但同时也是聪明地创建和利用消息队列。下面再回头介绍一下如何直接地对那些与特定的消息队列相联系的内部结构进行操作。

为了在一个消息队列上执行控制操作，用户可以使用 msgctl ( )系统调用。

```
  SYSTEM CALL: msgctl();
  PROTOTYPE: int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
    RETURNS: 0 on success
             -1 on error: errno = EACCES (No read permission and cmd is
IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid
with IPC_SET and
                                          IPC_STAT commands)
                                  EIDRM  (Queue was removed during retrieval)
                                  EINVAL (msgqid invalid, or msgsz less than 0)
                                  EPERM  (IPC_SET or IPC_RMID command was
issued, but
                                          calling process does not have write
(alter)
                                          access to the queue)
  NOTES:
```

现在有一种普遍的观点，认为直接对内部内核数据结构进行操作会给人带来一种满足感和成就感。不幸的是，由此而给程序员方面带来的责任却不是那么轻松的，除非用户喜欢破坏自己的IPC子系统。通过使用具有一个命令可选集合的 msgctl ( )，用户可以操纵那些不太容易造成破坏的项目。请看这些命令：

IPC_STAT

获取队列的msqid_ds结构，并把它存放在buf变元所指定的地址中。

IPC_SET

设置队列的msqid_ds结构的ipc_perm成员的值。它是从buf变元中取得该值的。

IPC_RMID

从内核删除队列。

前面介绍过消息队列的内部数据结构 (msqid_ds)。对于系统中存在的每一个队列，内核为它们都维护该结构的一个实例。通过使用 IPC_STAT命令，用户可以获取该结构的一个拷贝以供检查。请看下面这个包装程序函数，它将获取内部结构，并将其拷贝到作为参数传送给它的一个地址中：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
        if( msgctl( qid, IPC_STAT, qbuf) == -1)
        {
                return(-1);
        }
```

```
        return(0);
}
```

如果不能拷贝这个内部缓冲区，函数将向调用函数返回－ 1。如果一切工作顺利，则将返回一个为零的值，并且缓冲区中将会包含队列标识符 (qid)所代表的消息队列的内部数据结构的一个拷贝。缓冲区和qid都是作为参数传送给该函数的。

既然已经拥有了队列的内部数据结构的一个拷贝，那么，用户可以操作什么属性？用户应该怎样改变它们呢？在那个数据结构中，唯一可以修改的项目就是 ipc_perm成员。它包含该队列的许可权限，以及关于拥有者和创建者的信息。然而， ipc_perm结构中唯一可以修改的成员是mode、uid和gid。用户可以改变拥有者的用户 ID，拥有者的组ID以及队列的访问许可权限。

下面创建一个包装程序函数，用于改变队列的模式。该模式必须以字符数组的形式传送(如“ 660 ”)。

```
int change_queue_mode( int qid, char *mode )
{
        struct msqid_ds tmpbuf;

        /* Retrieve a current copy of the internal data structure */
        get_queue_ds( qid, &tmpbuf);

        /* Change the permissions using an old trick */
        sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);

        /* Update the internal data structure */
        if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
        {
                return(-1);
        }

        return(0);
}
```

通过调用get_queue_ds包装程序函数，可以获取内部数据结构的当前拷贝。然后再调用sscanf ( )，改变相联系的msg_perm结构的方式成员。然而，在用新拷贝更新当前内部版本之前，没有发生任何变化。这是通过使用 IPC_SET命令调用msgctl ( )来实现的。

注意！用户可以修改队列的许可权限。然而在实施过程中稍微的粗心大意会导致难以预料的后果。记住，这些IPC对象不会自动消失，除非它们被正确地删除，或者系统被重启。所以，即使用ipcs命令无法看到队列，也不意味着它不存在了。

为了说明这一点，有一则在某种程度上可以称得上幽默的轶事比较说明问题。在南佛罗里达大学教一门Unix内部结构课程时，我曾经遇到过一个相对尴尬的问题。为了编译和测试一个实验例题以便在我一个星期长的教学中使用，我在上课前的晚上进入实验服务程序。在测试过程中，我发现在用于改变某消息队列的许可权限的代码中，出现了一个打字错误。我创建了一个简单的消息队列，在测试发送和接收能力时并没有什么问题，然而当我试图把队列的访问许可方式从“ 660 ”变为“ 600 ”时，所带来的后果居然是我无法访问自己的队列了。因此，在源目录的同一个目录下，我不能测试消息队列的实验例题。因为我使用 ftok ( )函数

来创建IPC关键字，所以实际上我是企图访问自己没有正确访问权限的队列。最后，在上课前的那个早晨，我联系到了本地系统管理员，花了整整一个小时向他解释这是个什么样的消息队列，而我为什么需要他执行一次ipcrm命令帮我删除该队列。

在成功地从队列获取了一个消息以后，该消息将被删除。然而，正如前面所介绍的，除非用户显式地删除它，或者系统被重启，否则 IPC对象将保留在系统中。因此，消息队列还存在于内核中，在单个消息消失以后很长时间内仍然可用。为了结束一个消息队列的生命周期，用户需要使用IPC_RMID命令来调用msgctl ( )，以便显式地删除它：

```
int remove_queue( int qid )
{
        if( msgctl( qid, IPC_RMID, 0) == -1)
        {
                return(-1);
        }

        return(0);
}
```

如果队列被删除且无错误发生，则上面这个包装程序函数将返回　0，否则就将返回－ 1。队列的删除动作是原子性的，并且以后不论用什么理由访问该队列都将失败。

8. msgtool：交互式消息队列操作程序

如果随时可以得到正确的技术方面的信息，那将会给用户带来很直接的利益，没有人能否认这一点。对于学习和研究新的技术领域，信息和资料会提供一个强有力的帮助。同理可知，如果伴随着这些技术信息还能提供一些实际应用的例子，这无疑会加速学习进程，增强学习效果。

到目前为止，本书所提供的几个有用的例子是操作消息队列的包装程序函数。虽然它们是非常有用的，但它们的表达方式还不足以保证下一步的学习和实验。为了弥补这一点，我们给出msgtool，这是一个操作IPC消息队列的交互式命令行工具。虽然它实际上经常作为一个工具用于教学目的，但是，通过利用标准 shell脚本提供消息队列功能，该工具也可以直接应用在实际的赋值操作中。

9. 背景知识

msgtool依赖于命令行变元来确定自己的行为。当从 shell脚本调用时，这个特征使它显得尤其有用。该工具提供了所有的功能，从创建、发送和接收，到改变许可权限以及最后删除队列。当前，它使用字符数组来作数据，允许用户发送文本消息。当然也可以改变这一点，使它可以发送其他类型数据的工作。我们当作一个练习把它留给读者去完成。

10. 命令行语法

发送消息

```
msgtool s (type) "text"
```

获取消息

```
msgtool r (type)
```

改变许可权限(模式)

```
msgtool m (mode)
```

删除队列

```
msgtool d
```

11. 实例

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

12. 源代码

下面就是msgtool工具的源代码。应该在支持系统 V IPC的最新内核版本上编译这段代码。在作重建时一定要在内核中使能系统 V IPC。

顺便提一句，不论请求执行哪种类型的动作，这个工具在消息队列不存在时都会把它创建出来。

因为这个工具使用 ftok ( )函数来生成IPC关键字的值。用户可能会遇到目录冲突的问题。如果在脚本的任意位置改变目录，它可能不会工作。另一种解决方案是把一个更复杂的路径(如“ /tmp/msgtool ”)硬编码进msgtool中，或者甚至把路径和其他可操作的变元一起通过命令行传送。

```
/*************************************************************************
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *************************************************************************
 MODULE: msgtool.c
 *************************************************************************
 A command line tool for tinkering with SysV style Message Queues
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80

struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];


void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
    int main(int argc, char *argv[])
    {
```

```
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", 'm');

        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                        atol(argv[2]), argv[3]);
                          break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                          break;
                case 'd': remove_queue(msgqueue_id);
                          break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                          break;

                 default: usage();

        }

        return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);

        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
```