第十章 输入/输出子系统

输入/输出子系统（简称I/O子系统）的功能就是使进程能够与外部设备进行通讯，设备驱动程序是输入/输出子系统中的核心模块，并且与设备类型一一对应。

每一种类型的设备都有特定的设备驱动程序；而每一种设备驱动程序控制这种类型的所有设备。

10.1 设备驱动程序接口

UNIX系统中把设备分为两大类：

- 块设备 —— 以块为单位进行数据的输入输出，如硬盘、软盘、磁带、光盘等设备。
- 字符设备（原始设备，raw设备）—— 以字节为单位进行数据的输入输出，如终端、打印机、绘图仪、调制解调器、网卡等。

每个设备都有一个文件名（i节点）相对应，用于标识该设备的属性：

```
br--r--r--   1 root     system       14,  0 Mar 27 2009  cd0
crw-rw-rw-   1 root     system       15,  0 Mar 27 2009  clone
crw--w--w-   1 root     system        4,  0 Mar 27 2009  console
crw-rw-rwT   1 root     system       49,  0 Mar 28 2009  dlc8023
crw-rw-rwT   1 root     system       48,  0 Mar 28 2009  dlcether
crw-rw-rwT   1 root     system       47,  0 Mar 28 2009  dlcfddi
crw-rw-rwT   1 root     system       46,  0 Mar 28 2009  dlcqllc
crw-rw-rwT   1 root     system       45,  0 Mar 28 2009  dlcsdlc
crw-rw-rwT   1 root     system       44,  0 Mar 28 2009  dlctoken
crw-rw-rw-   1 root     system       15, 34 Mar 27 2009  echo
crw--w--w-   1 root     system        6,  0 Aug 28 10:05 error
crw-------   1 root     system        6,  1 Mar 27 2009  errorctl
brw-rw-rw-   1 root     system       19,  0 Mar 27 2009  fd0
```

1、系统配置

系统配置就是告诉核心，当前系统中包含哪些设备，以及这些设备的“地址” —— 建立设备文件、联接设备驱动程序，例如：

mknod /dev/tty15 c 2 15

核心与驱动程序的接口是由块设备开关表和字符设备开关表来描述的。

每一种设备类型在开关表中都有若干表项，这些表项在系统调用时引导核心转向适当的驱动程序接口。

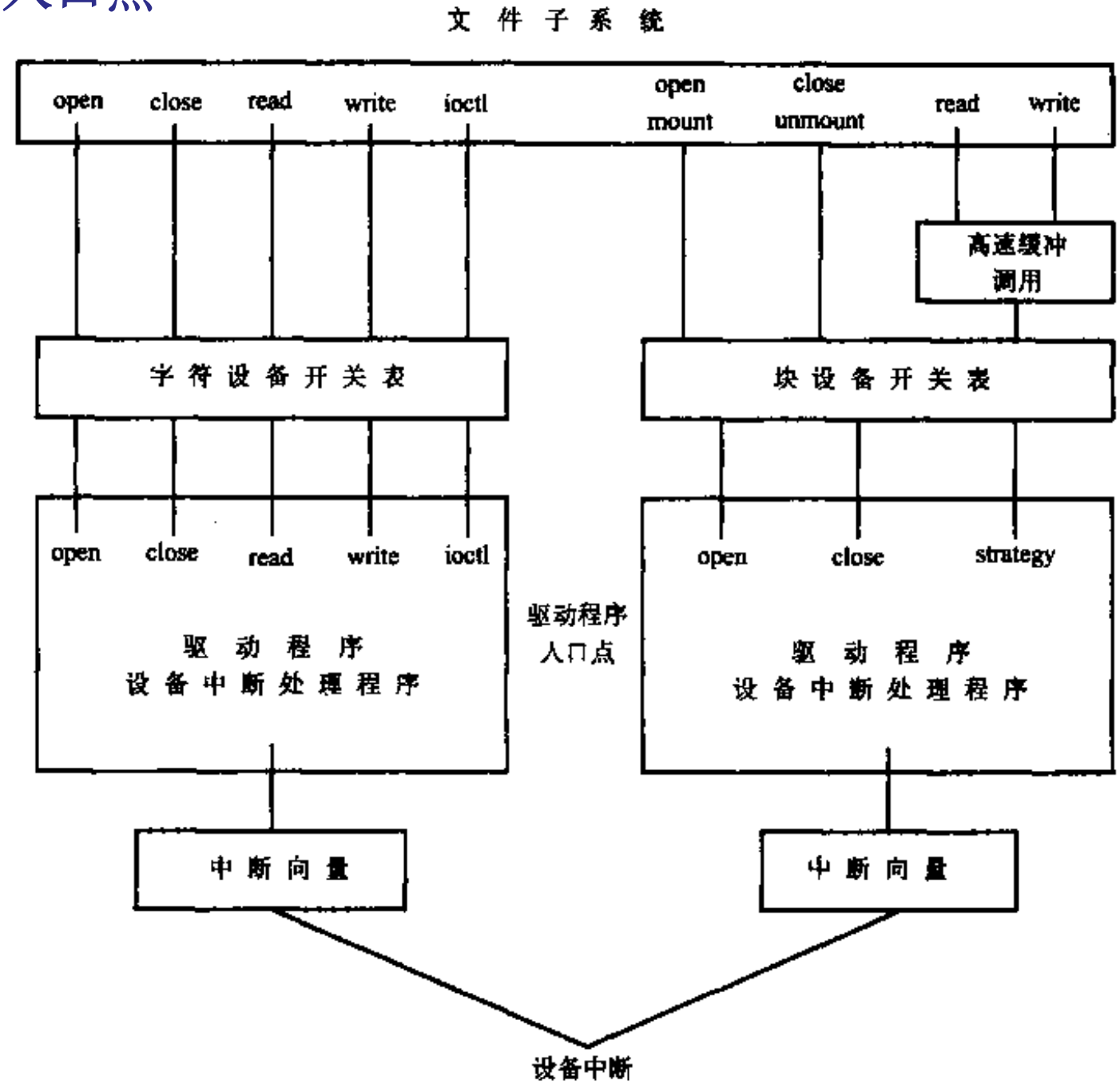
硬件与驱动程序的接口，是由与机器相关的控制寄存器或操作设备的I/O指令，以及中断向量组成：当一个设备发出中断时，系统识别发出中断的设备，并调用适当的中断处理程序。

设备开关表示例

块 设 备 开 关 表			
表项	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

字 符 设 备 开 关 表					
表项	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

驱动程序入口点


文 件 子 系 统
open
close
read
write
ioctl
open
mount
close
unmount
read
write
高速缓冲
调用
字 符 设 备 开 关 表
块 设 备 开 关 表
open
close
read
write
ioctl
驱 动 程 序
设 备 中 断 处 理 程 序
驱动程序
入口点
open
close
strategy
驱 动 程 序
设 备 中 断 处 理 程 序
中 断 向 量
中 断 向 量
设备中断

2、系统调用与驱动程序接口

对于使用文件描述符的系统调用的基本操作流程：

- 核心从用户文件描述符的指针找到系统打开文件表项和文件的索引节点。
- 检查文件的类型，根据需要存取块设备或字符设备开关表。
- 从索引节点中抽取主设备号和次设备号。
- 使用主设备号为索引值进入相应的开关表。
- 根据用户所发的系统调用来调用开关表中的对应函数。

2、系统调用与驱动程序接口（续）

对设备文件的系统调用与对正规文件的系统调用之间的一个重要区别是：当核心执行驱动程序时，对设备文件的索引节点是不上锁的。

因为驱动程序会频繁地睡眠，以等待硬连接或数据的到来，因此核心不能确定一个进程要睡眠多长时间。

如果对设备文件的索引节点上锁，则其它存取此节点的进程（如通过系统调用**stat**）会因为本进程在驱动程序中睡眠，而无限期地睡眠（等待）下去。

① 算法：open　/* 用于设备驱动程序的open */

输入：路径名，打开方式

输出：文件描述符

```
{
    将路径名转换为索引节点，增加索引节点的引用计数；
    与正规文件一样，分配系统打开文件表项和用户文件描述符；
    从索引节点取主设备号和次设备号；
    if (块设备)
    {
        使用主设备号作为查块设备开关表的索引值；
        调用该索引值对应的驱动程序打开过程——传递参数为次设备号和打开方式；
    }
    else   /* 是字符设备 */
    {
        使用主设备号作为查字符设备开关表的索引值；
        调用该索引值对应的驱动程序打开过程——传递参数为次设备号和打开方式；
    }
    if (open在驱动程序中失败)
        减少系统打开文件表项和索引节点的计数值；
}
```

②、算法：close　断开一个进程与设备的连接

a、搜索系统打开文件表（file表）以确认没有其他进程仍然打开着这个设备。

既不能仅仅看file表中的计数值来确认这是该设备的最后一次关闭操作，因为几个进程可能通过不同的file表项(读写指针)来存取该设备；

也不能依靠活动索引节点表(inode表)的计数值来确定这是否为最后一次关闭操作，因为可能有几个不同的文件代表同一设备，例如：

```
crw-rw-rw-  1  root  sys     9, 1    Aug  6  2014    /dev/tty01
crw-rw-rw-  1  root  unix    9, 1    Sep  5  2015    /dev/tty02
```

虽然上述两个设备的名字不同，但它们的主设备号和此设备号相同，因此是同一设备。有可能多个进程独立地打开这两个文件，这些进程存取不同的inode，但却是对同一设备进行操作。

②、算法：close　断开一个进程与设备的连接（续）

b、对于一个字符设备，核心调用该设备对应的close过程并返回用户态。

对于一个块设备，首先判断该设备上是否为一个已安装的文件系统。

如果是已安装的文件系统，则释放索引节点后返回。

如果不是已安装的文件系统，核心要做两件事：

- 搜索数据缓冲区高速缓冲，看是否有先前安装本文件系统时遗留下来的数据块以“延迟写”的形式还在内存中，还没有写回设备。如果有，则先写回这些数据块。
- 在关闭本设备后，核心再扫描一次数据缓冲区高速缓冲，使装有本设备数据块的缓冲区无效（释放到空闲链表的表头），以使装有有效数据的缓冲区能在内存中停留更长时间。

②、算法：close　断开一个进程与设备的连接（续）

c、核心释放设备文件的索引节点。设备关闭程序断开与设备的连接，并重新初始化驱动程序的数据结构和设备硬件，使核心以后又能再次打开该设备。

②、算法：close　断开一个进程与设备的连接

输入：文件描述符
输出：无

```
{
    执行正规的算法close;
    if ( 系统打开文件表项引用计数值非0 )
        goto   finish;
    if (  存在另一个打开文件其主、次设备号与欲关闭文件的相同  )
        goto   finish;
    if ( 字符设备 )
    {
        以主设备号为索引值查找字符设备开关表;
        调用驱动程序关闭子程序：参数为次设备号;
    }
    if ( 块设备 )
    {
        if ( 设备已作为子文件系统安装 )
            goto   finish;
        将数据缓冲区高速缓冲中该设备的数据块写回设备;
        用主设备号为索引值查找块设备开关表;
        调用驱动程序关闭子程序：参数为次设备号;
        使数据缓冲区高速缓冲中该设备的数据块无效;
    }
finish:
    释放设备文件索引节点;
}
```

③ 算法 read 和 write

- 针对设备的读写操作的算法与对正规文件的算法相似；
- 字符设备的驱动程序通常不使用系统中的数据缓冲区，而使用内部的数据缓冲机制；
- 由于外部设备的速度相对较慢，驱动程序的写操作通常包含流量控制；
- 驱动程序与设备间的数据通讯依赖于具体的硬件 —— 通常有“直接存储器存取（DMA）”模式和“可编程I/O”模式。

④、系统调用 ioctl

在较早版本的UNIX系统中提供有专门用于终端的系统调用stty(预置终端参数)和gtty(读取终端预置参数)。

ioctl融合了stty和gtty两者的功能，以提供一个更通用、更规范的设备控制入口。它允许一个进程去设置与一个设备相关联的硬件选项和与一个驱动程序相关联的软件选项。

ioctl可对多种设备进行操作，包括终端（如波特率、数据位宽度、奇偶校验）、磁带（如正绕或反绕的方式）、网络（如ip地址、虚电路数量）等。

由于ioctl规定的专门的动作对每个设备都是不同的，并由设备驱动程序来定义，因此，使用系统调用ioctl的程序必须知道他们正在与什么类型的文件打交道，因为这些文件都是设备专用的。

ioctl (fd, command, arg)

fd: 系统调用返回的文件描述符；

command: 使驱动程序完成指定动作的请求命令；

arg: 对应该命令的一个参数。

stty命令的运行时的屏幕截图

显示的是当前终端的相关参数，包括波特率、中断键、删除键的设置，字节位数、奇偶校验、字符回显的定义等。

```
$
$
$ stty -a
speed 9600 baud; 24 rows; 80 columns;
eucw 1:1:0:0, scrw 1:1:0:0:
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = ^@
eol2 = ^@; start = ^Q; stop = ^S; susp = ^Z; dsusp = ^Y; reprint = ^R
discard = ^O; werase = ^W; lnext = ^V
-parenb -parodd cs8 -cstopb hupcl cread -clocal -parext
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl -iuclc
-ixon -ixany -ixoff imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -flusho -pending iexten
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3
$
$
$
```

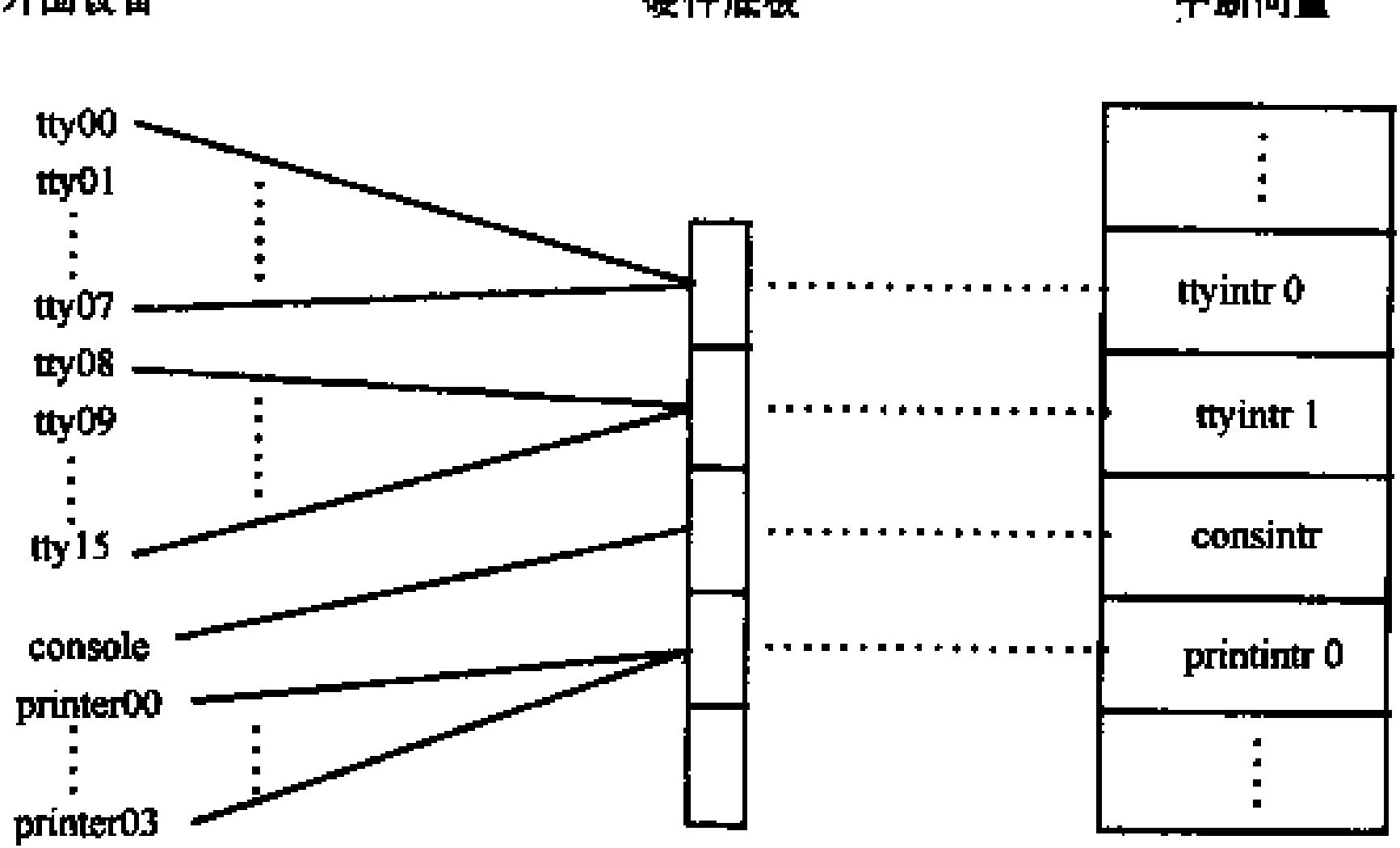
3、设备中断处理程序

一个设备的中断将引起核心执行一个中断处理程序。而执行什么样的中断处理程序是根据发出中断的设备和中断向量表中的偏移量来决定的。

核心调用设备专用的中断处理程序，并将设备号和其它参数传递给它，以便识别引起中断的特定的设备单元。

由于一个中断向量入口点可以与许多物理设备相联系，所以驱动程序必须能够判定是哪一个具体的设备引起的中断。

中断处理程序使用主设备号来识别硬件的类别，次设备号用来识别具体的设备。


外围设备
硬件底板
中断向量
tty00
tty01
tty07
tty08
tty09
tty15
console
printer00
printer03
ttyintr 0
ttyintr 1
consintr
printintr 0

设备中断

10.2 磁盘驱动程序

UNIX通常把一个物理的磁盘划分成若干个磁盘区域——称为逻辑设备，在每个磁盘区域上面都建立一个文件系统。

其主要目的是为了把不同类别的文件分别放在不同的文件系统中互不干扰，便于在进行文件系统的维护、备份和修复等工作时，分别对它们进行操作，减少相互影响。

在使用过程中，（除了根文件系统外）各个子文件系统可以根据需要选择“安装”、“不安装”、“只读”、“只写”和“可读可写”等工作方式，虽然这些子文件系统都在同一个物理设备上。

磁盘驱动程序把一个由逻辑设备号和块号构成的文件系统地址翻译成磁盘上特定的扇区号。

文件系统地址可由以下两种方法之一来获得：

1、使用数据缓冲区高速缓冲池中的一个缓冲区，该缓冲区的首部中就含有逻辑设备号和块号。

2、将逻辑设备号（次设备号）作为参数传递给读写程序，它们再把保留在u区中的字节偏移量转换为适当的块地址。

磁盘驱动程序用设备号来识别物理盘以及所使用的特定的磁盘分区，通过查找磁盘分区表就能找到特定磁盘分区在物理磁盘上的开始扇区，再把文件系统的块号加到起始扇区号上去，即可定位I/O传送所用的扇区号。

预分区的磁盘分区表示例

磁盘分区号	起始块号	块数
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

如果设备文件“/dev/dsk0”、“/dev/dsk1”、“/dev/dsk2”和“/dev/dsk3”对应于磁盘分区0到分区3，则相应于次设备号0到3。

假设文件系统块与磁盘块大小相同，则文件系统“/dev/dsk3”中的第940块，就对应于物理磁盘上的第336940块。

```
#include “fcntl.h”
main()
{
        char  buf1[4096], buf2[4096];
        int  fd1,  fd2,  i;
        if ((( fd1 = open(“/dev/dsk5”, O_RDONLY)) == -1  ||
              fd2 = open(“/dev/rdsk5”, O_RDONLY)) == -1  ))
        {
                printf(“failure  on  open\n”);
                exit();
        }
        if ((read(fd1, buf1, sizeof(buf1)) == -1 || (read(fd2, buf2, sizeof(buf2)) == -1))
        {
                printf(“failure on read\n”);
                exit();
        }
        for ( i=0; i<sizeof(buf1); i++)
                if (buf1[i] != buf2[i])
                {
                        printf(“different at offset %d\n”,  i);
                        exit();
                }
        printf(“reads  match\n”);
}
```

10.3 终端驱动程序

终端是用户与系统的交互界面。终端驱动程序用于控制从终端来和到终端去的数据。

终端驱动程序包含一个与行规则（line discipline）模块的内部接口，行规则程序的功能就是对输入和输出数据进行解释。

终端有两种工作方式：

① 标准方式（canonical）—— 行规则程序把键盘输入的原始数据变换成标准形式；或把进程输出的原始数据变换成用户期望的形式。

② 原始方式（raw）—— 行规则程序只完成进程和终端之间的数据传输，而不做变换。

行规则程序的功能：

① 通过分析将输入字符串变成行；

② 处理擦除键；

③ 处理“抹行”字符，它使当前行上已敲入的所有字符无效；

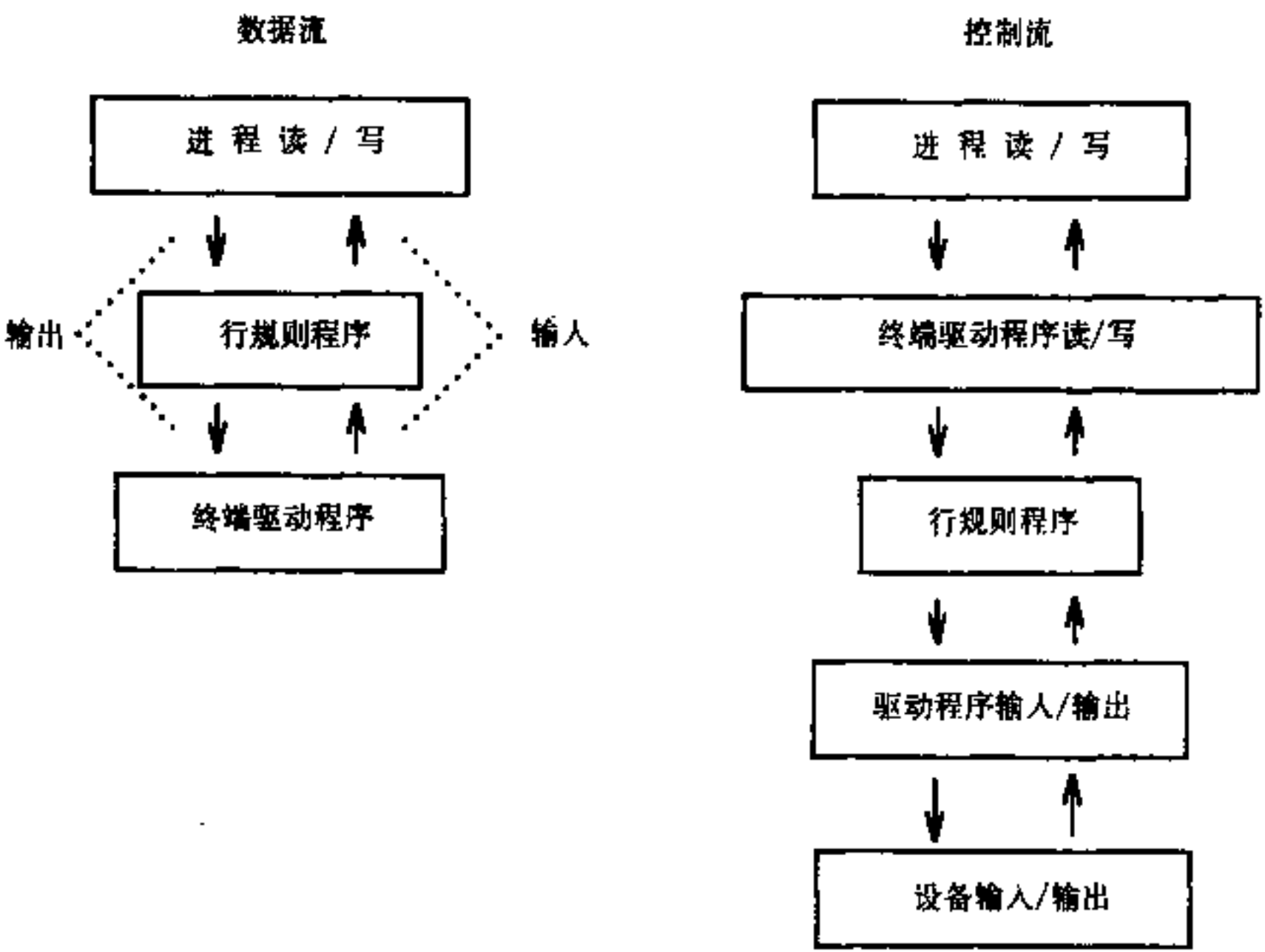
④ 把接受的字符回显（写）到终端上；

⑤ 扩展输出，如把制表符变成一系列空格；

⑥ 为终端挂起（hangup）、断线或响应用户敲入的delete键，向进程产生软中断信号；

⑦ 允许不对特殊字符如擦除、抹行和回车进行解释的原始方式。

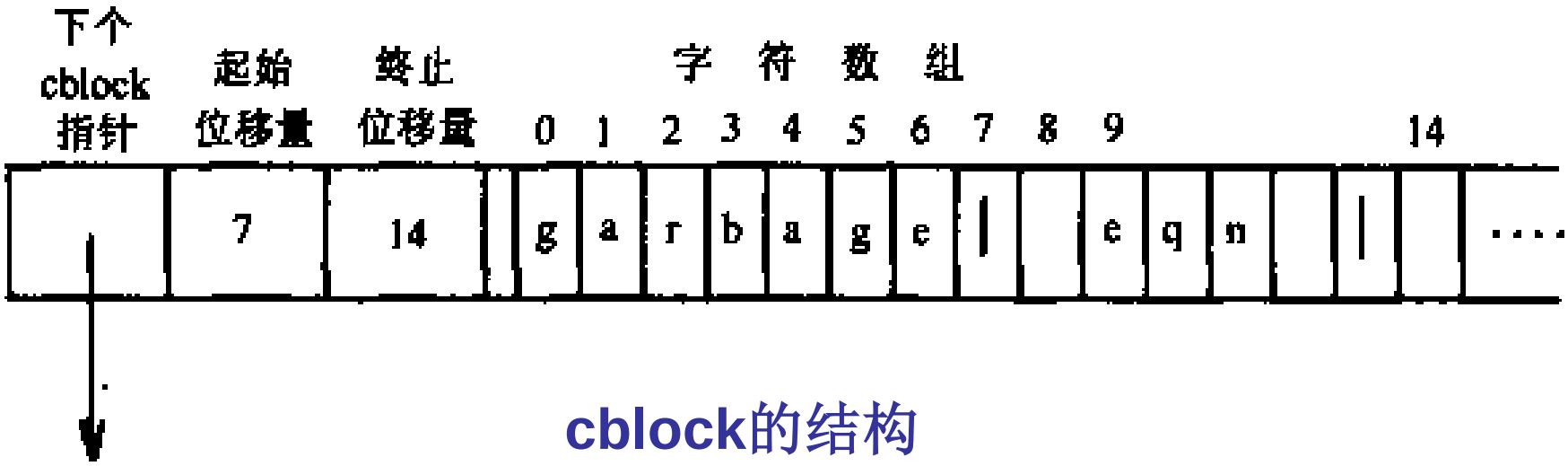
支持原始方式意味着可以使用一个异步的终端，这样进程可以在字符被敲入时就读它们，而不是等待用户敲入一个回车或“enter”键时才读 —— 如cbreak功能。



行规则程序的调用顺序及数据流

1、字符表clist

行规则程序在字符表上操作数据。字符表clist是字符缓冲区cblock的变长链表，并携带有表中的字符计数。



cblock的结构

起始位移量 —— 执向第一个有效数据的位置
终止位移量 —— 指向第一个无效数据的位置

核心维护一个空闲cblock的链表，并规定在clist和cblock上有六种操作：

① 从空闲链表中分配一个cblock给驱动程序；

② 把一个cblock归还给空闲链表；

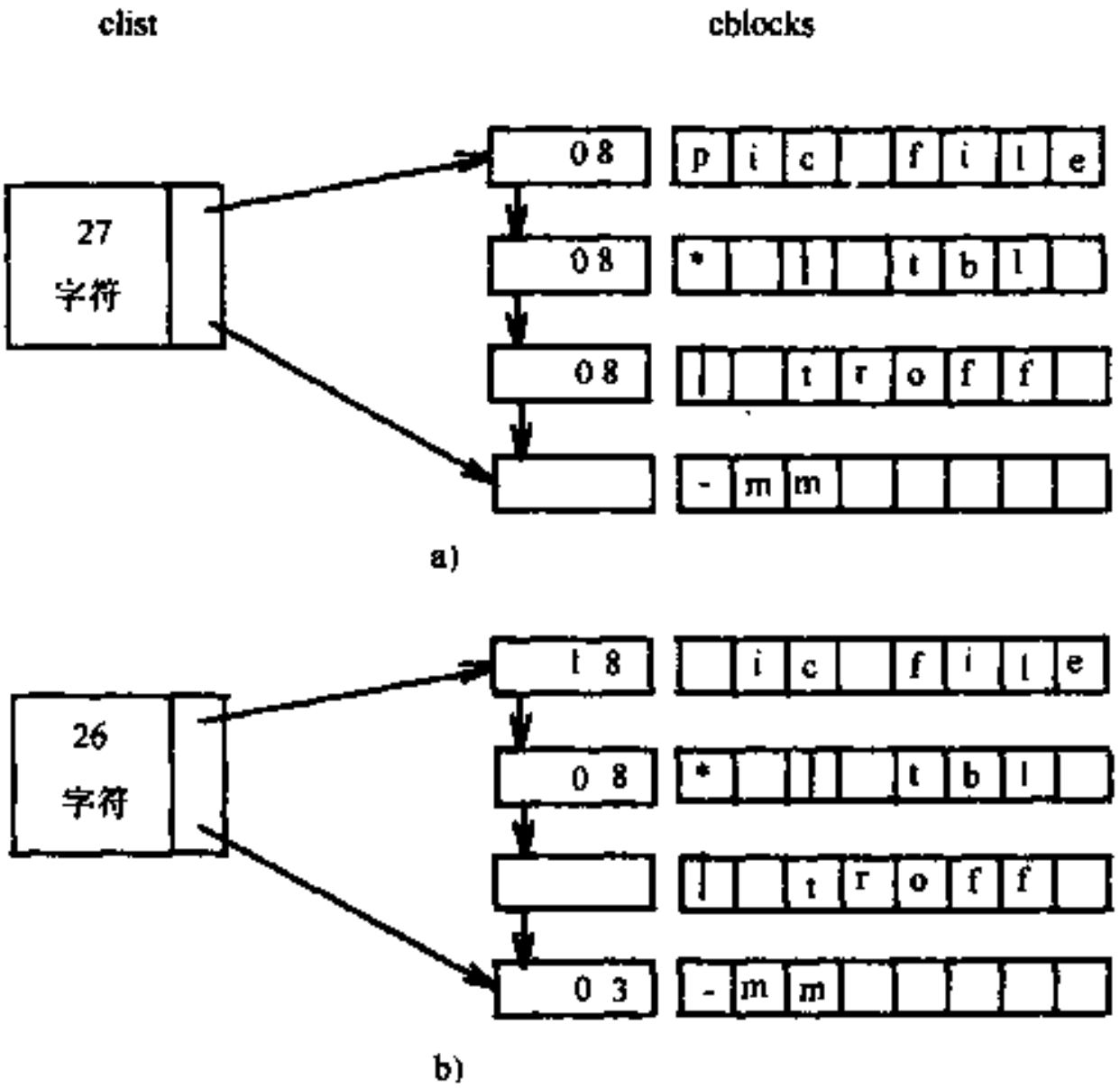
③ 从一个clist中取出一个字符；

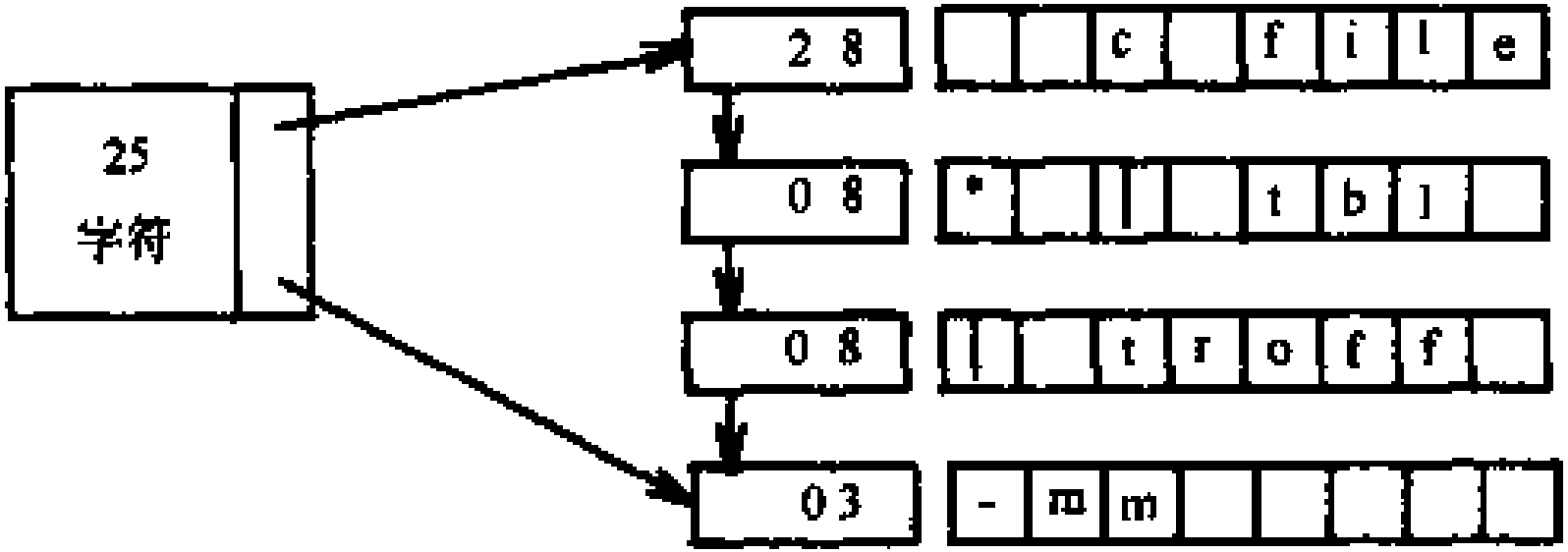
④ 把一个字符放入clist表的末尾；

⑤ 从一个clist表的开头移去一组字符，每次一个cblock；

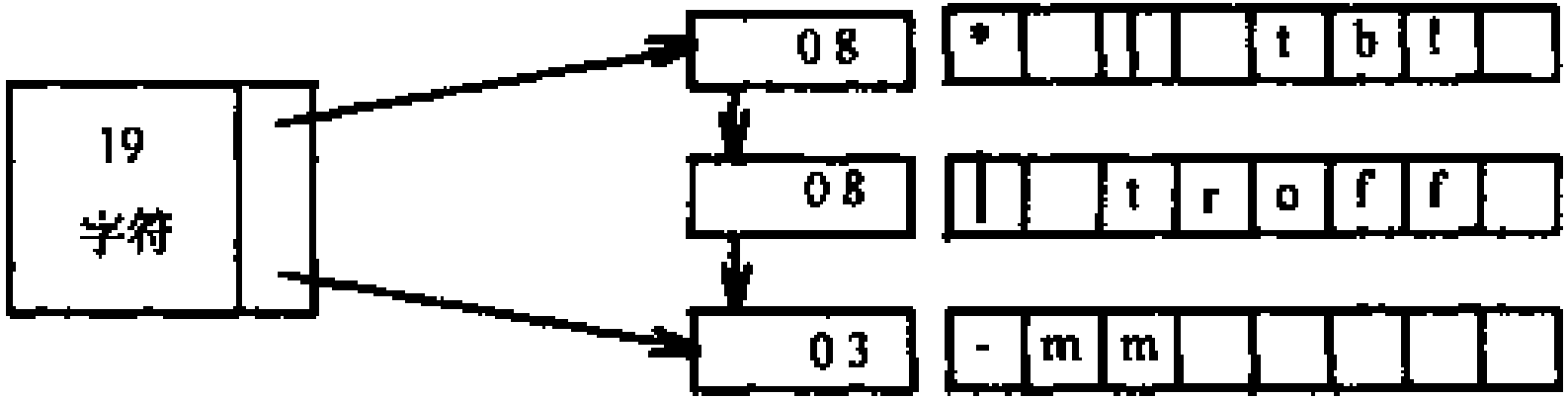
⑥ 把一个cblock的字符放到一个clist表的末尾。

图例1：从clist中移去字符 —— getchar （读数据）



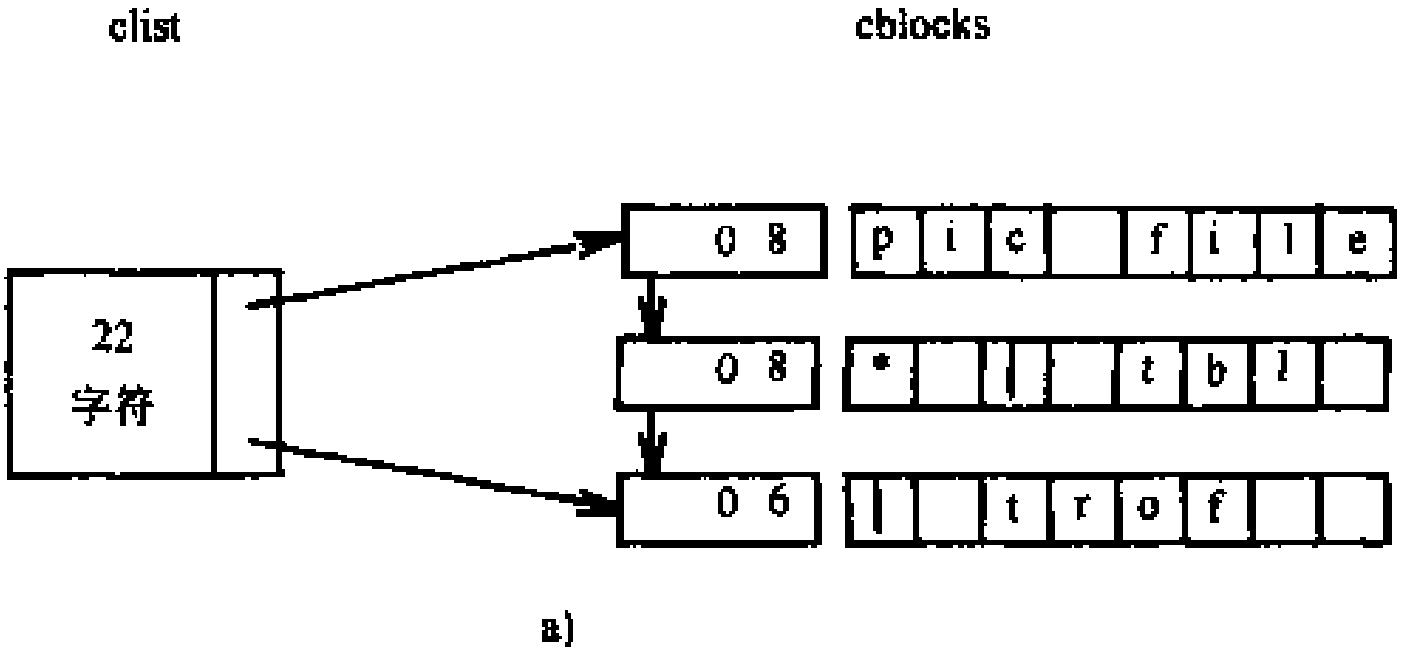


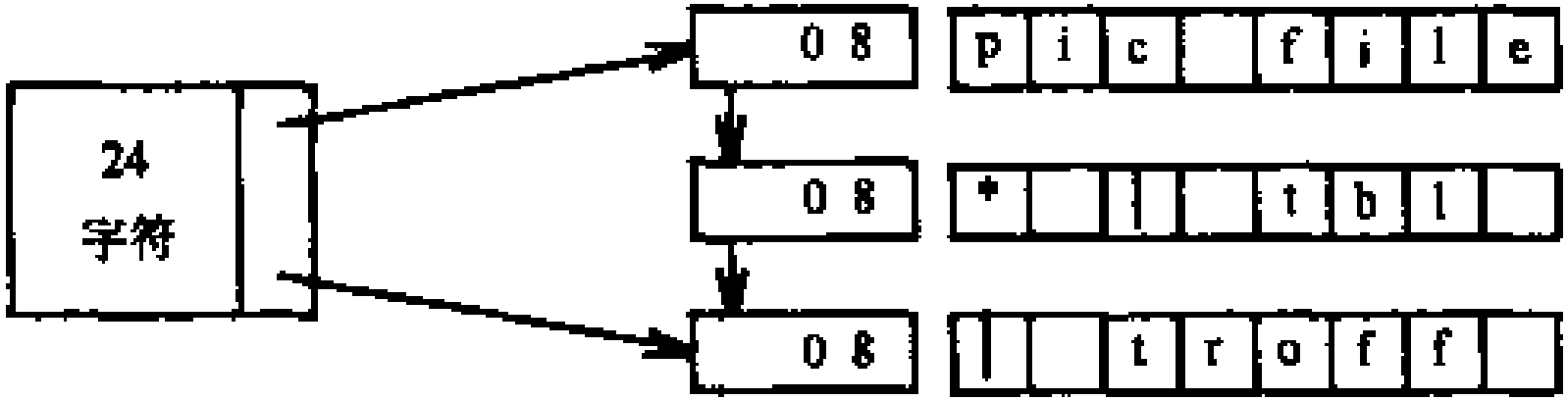
c)



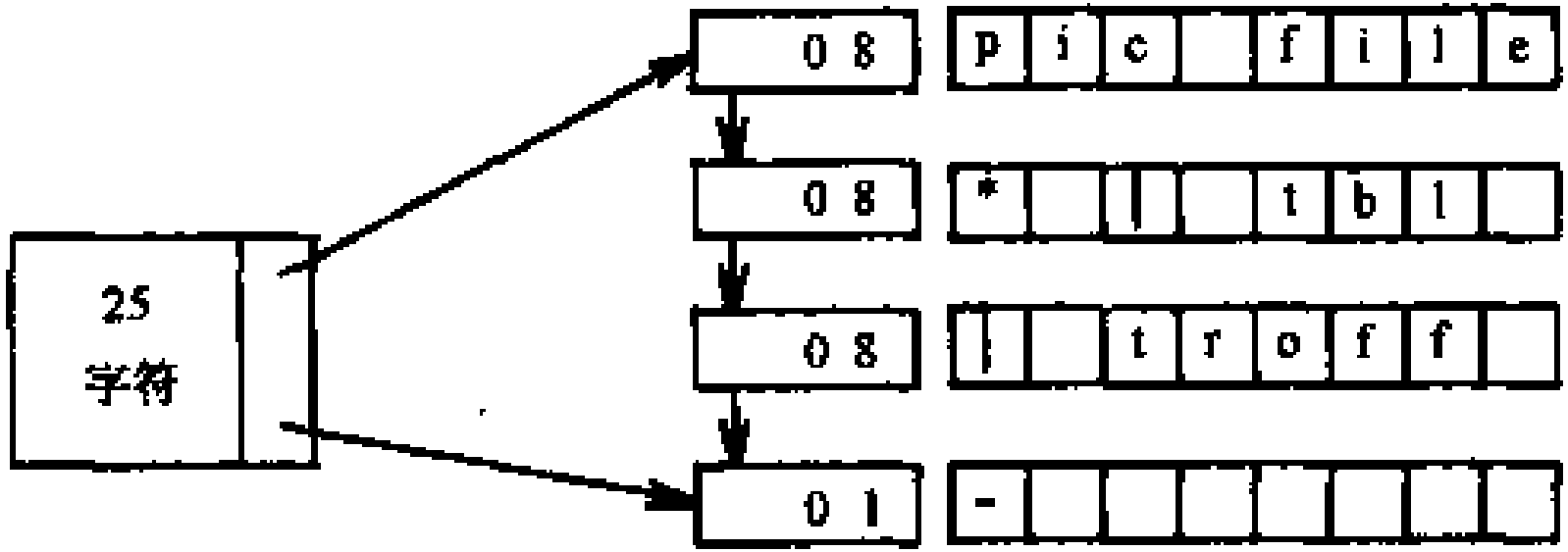
d)

图例2：向clist中放入字符 —— putchar （写操作）




24
字符
0 8
0 8
0 8
p i c f i l e
* | t b l
| t r o f f

c)


25
字符
0 8
0 8
0 8
0 1
p i c f i l e
* | t b l
| t r o f f
-

d)

2、标准方式下的终端驱动程序

在终端驱动程序的数据结构中包含三个与之关联的clist：

一个clist用来存放输出到终端去数据；

一个clist用来存放从终端输入的“原始”数据，这是当用户敲入字符时，终端中断处理程序放入的；

一个clist用来存放“已加工”的输入数据，称之为标准clist。

当一个进程要向一个终端写数据时，终端驱动程序调用行规则程序从用户地址空间读取要输出的字符，将其放入输出clist中。在此过程中，行规则程序对输出字符加以处理，例如将制表符扩展为一组空格。

当输出clist上的字符数达到最大值时，行规则程序调用终端驱动程序把clist中的数据发送到终端，并使写进程进入睡眠。

当输出clist中的字符数小于最小值时，终端驱动程序唤醒所有睡眠等待“该终端能够接受更多数据”事件的进程。

```
算法 terminal_write
{
    while (有数据等待从用户空间输出)
    {
        if (tty结构填满待输出的数据)
        {
            启动硬件写操作以便送出“输出clist”中的数据；
            sleep(tty能接受更多数据的事件)；
            continue;
        }
        从用户空间拷贝cblock大小的数据到“输出clist”中；
        行规则程序变换制表符等；
    }
    启动硬件写操作以便送出“输出clist”中的数据；
}
```

```
char  form[ ] = “this is a sample output string from child”;
main ( )
{
      char output [128];
      int  i;
      for ( i=0; i<18; i++)
      {
            switch(fork( ))
            {
                  case  -1:                        /*  错误——已到达最大进程数  */
                        exit ( );
                  default:                         /*  父进程  */
                        break;
                  case  0:                         /*  子进程  */
                        /*  在变量output中形成输出字符串  */
                        sprint(output,”%s%d\n%s%d\n”, form, i, form, i);
                        for( ; ; )
                              write(1, output, sizeof(output));
            }
      }
}
```

多进程同时在标准输出上输出数据时的竞争

在标准方式下，当一个进程从终端读数据时，由终端中断处理程序把用户敲入的字符放到“原始clist”中，以便输入给读进程，同时还要把字符放到“输出clist”中，以便回显（echo）到终端屏幕上。

如果输入字符中含有一个回车符，则中断处理程序要唤醒全部读进程。

当一个读进程运行时，驱动程序从“原始clist”中移出字符，对擦除和抹行符进行处理，并把字符放入“标准clist”中去，然后再把字符拷贝到用户地址空间中去，直到接收到回车符或满足了系统调用read中的计数值时为止。

一个进程可能会发现，它被唤醒时的那些数据已不存在了，这是因为这个进程被调度之前，其它进程可能也从终端读，并从“原始clist”中移走了数据。

```
算法 terminal_read
{
    if (标准clist中无数据)
    {
        while (原始clist中无数据)
        {
            if (tty是以“不延迟”选项打开的)
                return;
            if (tty是基于定时的原始方式且定时器未激活)
                设置定时器唤醒(callout表);
            sleep(事件：数据从终端到来);
        }
        /* 原始clist上有数据 */
        if (tty是原始方式)
            将全部数据从原始clist拷贝到标准clist;
        else    /* tty是标准方式 */
        {
            while (原始clist中有字符)
            {
                每次从原始clist中拷贝一个字符到标准clist中，并进行擦除、抹行处理;
                if (字符是回车符或文件尾)
                    break;      /* 退出循环 */
            }
        }
    }
    while (标准clist中有字符且欲读的字节数未满足)
        从标准clist的cblock中拷贝字符到用户地址空间;
}
```

```
char input[256];
main( )
{
        register int i;
        for (i = 0; i < 18; i++)
        {
                switch(fork( ))
                {
                        case  -1:       /* 错误 */
                                printf(“error cannot fork\n”);
                                exit();
                        default:        /* 父进程 */
                                break;
                        case  0:        /* 子进程 */
                                for (; ;)
                                {
                                        read(0, input, 256);   /* 读入一行 */
                                        printf(“%d read %s\n”, i, input);
                                }
                }
        }
}
```

示例：多进程对终端输入数据的竞争

3、原始方式下的终端驱动程序

进程可以通过系统调用**ioctl**来设置终端参数，如擦除符、抹行符、波特率、回显符等。行规则程序在收到系统调用**ioctl**的参数时，预置终端数据结构中有关的域。当一个进程预置终端参数时，其它所有使用该终端的进程都将使用这些参数。当预置参数的进程退出时，并不自动地复位对终端的预置。

进程也可以使终端在原始方向下工作，此时行规则程序对用户敲入的字符不作任何处理。由于回车符已被作为普通的输入字符了，所以核心必须要用以下两种方法之一来确定何时应该完成用户的系统调用**read**:

①、终端读入了指定数量的字符后；

②、从终端读入了任意字符后等待了指定的时间。

当上述条件成立时，行规则中断处理程序唤醒所有睡眠的进程，终端驱动程序把原始**clist**中的全部字符移到标准**clist**中，完成用户的读请求。

以原始方式从终端读入5个字符

```
#include   <signal.h>
#include   <termio.h>
struct  termio  savetty;
main()
{
      extern sigcatch();
      struct  termio  newtty;
      int  nrd;
      char  buf[32];
      signal(SIGINT, sigcatch);                    /*  捕俘软中断信号，执行sigcatch  */
      if ( ioctl(0, TCGETA, &savetty) == -1)     /*  保存原来的tty设置  */
      {
            printf("ioctl failed: not a tty\n");
            exit();
      }
      newtty = savetty;
      newtty.c_lflag  &=  ~ICANON;                 /*  停止标准方式  */
      newtty.c_lflag  &=  ~ECHO;                   /*  停止字符回显  */
      newtty.c_cc[VMIN]  =  5;                     /*  最小值为5个字符  */
      newtty.c_cc[VTIME]  =  10;                   /*  10秒间隔  */
```

（接下页）

（接上页）

```
if ( ioctl(0, TCSETAF, &newtty) == -1)     /*  把终端设置为新的工作方式  */
      {
            printf("cannot put tty info raw mode\n");
            exit();
      }
      for ( ;  ; )
      {
            nrd = read(0, buf, sizeof(buf));          /*  从终端读数据  */
            buf[nrd] = 0;
            printf("read %d chars '%s'\n", nrd, buf);    /*  显示实际读入的字符串  */
      }
}
sigcatch()     /*  在等待字符输入时，如果遇到软中断信号，则恢复原来的tty设置  */
{
      ioctl(0, TCSETAF, &savetty);
      exit();
}
```

4、终端探询

应用程序可能需要经常探询（等待）一个终端设备上是否有数据输入，这通常可以用一些简单的方法来实现：有数据出现就读数据；没有数据输入时，就继续正常的处理或者执行空操作等待。

下面的程序中，由于选定了以“不延迟”方式打开终端，无限循环探询终端数据。

```
#include  <fcntl.h>
main()
{
      register  int  i,  n;
      int  fd;
      char  buf[256];
      if ((fd = open("dev/tty", O_RDONLY | O_NDELAY)) == -1)
            exit();
      n = 1;
      for ( ;  ; )
      {
            for ( i=0;  i<n;  i++)
                  ;
            if ( read (fd,  buf,  sizeof(buf)) > 0)
            {
                  printf("read  at  n  %d\n", n);
                  n--;
            }
            else      /*  无数据可读时，因“不延迟”而返回  */
                  n++;
      }
}
```

在UNIX系统中有一个可探询多个设备的系统调用select，其语法格式是：

select(nfds, rfds, wfds, efds, timeout)

其中：
nfds —— 是要选择的文件描述符的数量；
rfds —— 指向“所选择的读文件描述符”的位图指针。如果用户要选择文件描述符fd，则“1”左移文件描述符的值那么多位的位置被置位；
wfds —— 指向“所选择的写文件描述符”的位图指针，置位方法与rfds相同；
efds —— 指向被例外条件监控的文件描述符位图；
timeout —— 指出为等待数据的到来select要睡眠多长时间，当指定时间到，无论是否有设备准备好，都返回。

5、建立控制终端

控制终端通常是用户注册到系统中时所用的终端，它控制用户在该终端上创建的那些进程。

当一个进程打开一个终端时，终端驱动程序就将打开行规则程序。

如果该进程是一个进程组组长，且该进程还没有一个与之相关联的终端，则行规则程序就将打开的这个终端成为该进程的控制终端。

行规则程序会把终端的主设备号和次设备号存放在该进程的u区中，并把该进程的进程组号存放在终端驱动程序的数据结构中。

控制终端在处理软中断信号时起着重要的作用。当用户按下**delete**、**break**、**quit**、**^C**等键时，中断处理程序调用行规则程序，行规则程序向该控制进程所在组中的所有进程发出软中断信号。

当用户挂起，终端中断处理程序会从硬件收到一个挂起指示，而行规则程序向该控制进程所在组的所有进程发出一个**SIGHUP**（**hangup**）软中断信号，用这种方法使一个特定终端上创建的所有进程都收到**SIGHUP**信号。

收到这一信号的大多数进程的缺省动作是退出，这就是当一个用户突然关掉终端电源时，这些杂散的进程被杀掉的方法。

在发出**SIGHUP**信号后，终端中断处理程序断开该终端与该进程组的联系，使该进程组中的其它进程再也不能接收该进程发出的软中断信号了。

6、注册到系统

系统进入到多用户状态后，1号进程——init的一个主要工作就是允许用户通过终端注册进入系统。

init创建若干个getty进程，每个getty进程重置进程组号，打开一个特定的终端线路后，睡眠在系统调用open中，直到检测到与终端的硬件连接建立为止。

当open返回时，getty通过系统调用exec执行注册程序login，等待用户的登录。

如果用户注册成功，login通过系统调用exec执行shell，用户就可以使用系统了。该shell进程就是注册shell，与原来的getty进程具有相同的进程标识号，因此该shell进程就是一个进程组组长。

init进程进入睡眠状态，直到它收到一个子进程死软中断信号。当它被唤醒后，它要查明该僵死子进程是否为一个注册shell进程，如果是，则init又创建出另一个getty进程以便可以再次使用该终端。

```
算法   login   /* 注册的过程 */
{
    getty进程执行;
    设置进程组（系统调用setpgrp）;
    打开tty线路;              /*  睡眠直到被打开 */
    if ( 打开成功)
    {
        执行login程序;
        显示输入用户名要求;
        停止终端回显(echo)字符，显示输入口令要求;
        if ( 成功)        /* 与/etc/passwd中的口令相同 */
        {
            将tty设置为标准方式（用ioctl）;
            执行shell程序;
        }
        else
            计算企图登录的次数，小于设定值时再试;
    }
}
```

课 程 结 束

谢谢！

信息与软件工程学院　　刘 玓