第15章 基本输入/输出介绍

在详细讨论UNIX输入/输出之前，需要先彻底地了解下列三个文件的内容。

15.1 buf.h文件

此文件说明了两个结构，它们是“ buf ”(4520)和“ devtab ”(4551)。“ bfreelist ”说明为“ buf ”结构类型(4567)，具有“ NBUF ”个元素的数组“ buf ”也被说明为“ buf ”类型(4535)。

“ buf ”结构的名字容易引起人们的误解，实际上它起的作用是缓存头 (或缓存控制块)。缓存区则被说明为(4720)：

```
“char buffers[NBUF][514]”;
```

“ buf ”数组中的各指针指向“ buffers ”数组中各缓存，这种关系是由“ binit ”过程建立起来的。

“ buf ”结构的其他实例还有：“ swbuf ”(4721)和“ rrkbuf ”(5387)。对于“ bfreelist ”、“ swbuf ”和“ rrkbuf ”，并无缓存区与它们相关连。

可将“ buf ”结构分成三部分：

1) 标志：这些标志包含状态信息，并且都位于 1个字中。设置这些标志的屏蔽字定义为“ B_READ ”和“ B_WRITE ”等(4572～4586)。

2) 列表指针：两个双向链接列表的前向和后向指针，这两个列表我们将称之为“ b ”列表和“ av ”列表。

3) i/o参数：与实际数据传输相关连的一组值多。

15.2 devtab(4551)

“ devtab ”结构包含5个字，其中最后4个是前向和后向指针。

对于每一个块类型外部设备，在其设备处理程序中都说明一个“ devtab ”实例。在我们的模型系统中，唯一的块设备量RK05磁盘，在5386行将“ rktab ”说明为“ devtab ”结构类型。

“ devtab ”结构包含相应设备的某些状态信息，并用作下列列表的表头：

1) 与相应设备相关连的各缓存的列表，这些缓存可能同时位于“ av ”列表中。

2) 对于该设备的未决i/o请求列表。

15.3 conf.h文件

“ conf.h ”文件说明：

- 将一个整型分解成两个部分 (d_minor和d_major)的另一种方法。注意，“ d_major ”对应于“ 高字节 ”(0180)。
- 两个结构数组。

• 两个整型变量，“ nblkdev ”和“ nchrdev ”。

“ conf.h ”中说明了两个结构数组“ bdevsw ”和“ cdevsw ”，但是没有说明它们的大小和初始化值。对这两个数组的初始化是在“ conf.c ”文件中实现的。

15.4 conf.c文件

“ conf.c ”和“ low.s ”是由程序“ mkconf ”针对每一个系统的具体配置而生成的 (以反映实际安装的外设类型和数量)。在我们的样本中，“ conf.c ”反映了我们的模型系统所提供的设备。

此文件对下列项进行初始化：

```
bdevsw      (4656)      swapdev     (4696)
cdevsw      (4669)      swplo       (4697)
rootdev     (4695)      nswap       (4698)
```

15.5 系统生成

在安装UNIX时系统生成的主要步骤是：

• 以适当的输入运行“ mkconf ”。

• 再编译输出文件(被创建为c.c和l.s)。

• 以修改过的目标文件重新装入系统。

这一过程仅需数分钟 (而在其他操作系统中这往往需要几个小时)。注意，在“ conf.c ”中所定义的“ bdevsw ”和“ cdevsw ”与其他各处有所不同，它们都被定义为一维指针数组，而这些指针指向返回整型值的函数。该定义中忽略了其中某些项，例如“ rktab ”并非函数这一事实，这种处理并未造成问题，其原因是连接程序对此并不进行检查。

15.6 swap(5196)

在详细介绍“ bio.c ”之前，先查看一个先前已引入的名为“ swap ”的例程，这对以后的分析会是有教益和方便的。

缓存头“ swbuf ”的作用是控制交换的输入/输出，这种操作与其他很多对磁盘的存取活动是同时存在、交错进行的。“ swbuf ”不与缓冲池中的缓存相关连，而程序占用 (或将占用)的内存区则起类似于数据缓存的作用。

5200：将“ swbuf ”中各标志的地址传送给寄存器型变量“ fp ”，这主要是为了方便和提高运行效率。

5202：测试“ B_BUSY ”标志，若已设置，则说明已有一交换操作在处理之中，所以设置“ B_WANTED ”标志，然后此进程必须调用“ sleep ”以等待使用“ swbuf ”。

注意，5202至5205行的循环以处理机优先级6级运行，它比磁盘中断优先级高1级。

你能否想到必须这样处理的原因吗?在什么样的条件下，“ B_BUSY ”标志将被设置呢?

5206：设置这三个标志(B_BUSY、B_PHYS和rdflg)以反映：

• “ swbuf ”正在使用之中(B_BUSY)。

• 物理i/o意味着直接针对用户数据段有一个大数据量的读/写操作(B_PHYS)。

• 该操作是读还是写(rdflg是调用swap的一个参数)。

5207：对“ b_dev ”字段赋值(这一操作可以在系统初启时只执行一次(在binit)，而不必在每次使用swbuf时都执行)。

5208：对“ b_wcount ”赋值。注意其中的负值以及乘32的方法。

5210：硬件设备控制器要求完整的物理地址(在PDP/11-40中是18位)。32字块块数必须变换成两部分：低10位向左移6位，然后存入“ b_addr ”，余下的高6位存入“ b_xmem ”(在PDP11/40和PDP11/45中，这6位中只有2位是有意义的)。

5212：初看起来这条语句复杂得令人吃惊!将“ swapdev ”向右移8位以获得主设备号。使用此结果作为“ bdevsw ”数组的下标取得相应bdevsw结构型元素。从该元素中选择、抽取“ d_strategy ”指向的例程，然后以“ swbuf ”地址作为参数调用执行该例程。

5213：解释为什么需要调用“ spl6 ”。

5214：等待直至i/o操作完成。注意，调用“ sleep ”的第1个参数是“ swbuf ”的地址。

5216：唤醒等待“ swbuf ”的所有进程。

5218：将处理机优先级设置为0，于是使任一悬而未决的中断得以发生。

5219：清“ B_BUSY ”和“ B_WANTED ”标志。

15.7 竞态条件

在“ swap ”代码中有一些令人感兴趣的特征。特别是当有很多进程同时运行时，它显现了竞态条件问题。

让我们考虑下列情景：

在进程A启动一个交换操作时，并没有正进行的交换操作。为叙述的方便，将“swbuf.b_flags ”简单地表示为“ flags ”，于是在开始此情景时，具有：

```
flags==null
```

在5204行，进程A没有延迟，立即启动其i/o操作，然后在5215行调用“ sleep ”进入睡眠状态。现在我们具有下列条件表达式：

```
flags==B_BUSY|B_PHYS|rdflg
```

这是在5206行设置的。

现在假定：正在进行i/o操作时，进程B也启动了一个交换操作。这也是从执行“ swap ”开始的，但在执行“ swap ”中发现“ B_BUSY ”已设置，于是它设置“ B_WANTED ”标志(5203)，然后也进入睡眠状态(5204)。现在我们具有：

```
flags==B_BOSY|B_PHYS|rdflg|B_WANTED
```

最后，该i/o操作完成。进程C接获中断并执行“ rkintr ”，它调用(5471)“ iodone ”，而“ iodone ”又调用(5301)“ wakeup ”以唤醒进程A和进程B。“ iodone ”也设置“ B_DONE ”标志，并清“ B_WANTED ”标志， 于是：

```
flags==B_BUSY|B_PHYS|rdflg|B_DONE
```

下一步做什么取决于是进程A先执行还是进程B先执行(这两个进程的优先数相同，都是

PSWP，谁先执行是难以预料的)。

1) 情况(a)：进程A先执行。因为“ B_DONE ”已设置，所以不需要再睡眠。因为“ B_WANTED ”已清0，所以也无需“ 唤醒 ”任何进程。进程A稍作处理(5219)后即从“ swap ”返回，此时：

flags==B_PHYS|rdflg|B_DONE

然后，进程B运行，它无需延迟就可立即启动其i/o操作。

2) 情况(b)：进程B先执行，它发现“ B_BUSY ”标志已设置，所以它重新设置“ B_WANTED ”标志，再次进入睡眠状态，此时：

flags==B_BUSY|B_PHYS|rdflg|B_DONE|B_WANTED

如同情况(a)中一样，进程A执行，但此次它发现“ B_WANTED ”设置，所以除其他操作外，它必须调用“ wakeup ”(5217)。最后，进程B再次苏醒，整个操作链完成。

情况(b)与情况(a)相比，显然其效率较低。看来对5215行稍加更改，将其改成：

“sleep(fp,PSWP-1)”,

就能保证情况(b)再也不会产生，而且这种更改不增加任何开销。

应当仔细思考在各种情况下提高处理机优先级的必要性：例如若省略 5201行，并且假定进程B刚完成5203行，其时进程A起动的i/o操作恰好完成，中断发生，那么“ iodone ”将清“ B_WANTED ”标志，在进程B睡眠之前就对其执行“ wakeup ”，这实际上并不起作用，而进程B真正进入睡眠后却无进程唤醒它，于是进程B永远睡眠!若出现此情况那就太糟了。

15.8 可重入

口进程B可同时执行“ swap ”。所有UNIX过程一般而言是
:可能的)。如果不允许可重入， UNIX会改成什么样式呢?

u_ssav ”进行了讨论但并未结束，现在我们可以完成对此

',它调用(4380)“ swap ”,“ swap ”调用“ sleep ”,“ sleep ”
;av ”(2189)。
”之后，经过四重过程调用再将“ u.u_rsav ”清除。
奇的论文“ The UNIX I/O System ”(UNIX输入/输出系