# 设备管理模块

设备管理模块可以分为块设备管理模块与字符设备管理模块。该模块分别提供了字符设备例如键盘，或者块设备例如硬盘等设备的驱动程序，这些驱动程序在硬件发出中断时被调用执行。

对块设备而言，所有块设备读写操作都通过高速缓存模块向设备驱动发送指令，同时，上层接口也主要通过高速缓存模块访问块设备。同时，字符设备的打开设备表向上层文件系统提供统一的设备管理接口，文件接口通过该管理接口控制相应的字符设备。

高速缓存模块在系统内核中保留了一块物理内存区域，由于块设备的读写性质，每次对块设备的读写操作都将导致一块数据的读入和写入，因此，操作系统为了提高块设备读写操作的利用率，设置了高速缓存。高速缓存模块则是管理这些内存数据的模块。

在Unix V6++中，高速缓存模块的大部分结构都将被保留，只是会使用BufferManager类来统一管理高速缓存的数据块，使函数分工更加清晰。而设备的驱动由于体系结构的巨大差异，将会被完全重新设计。涉及到的驱动程序包括字符驱动，显示驱动以及硬盘设备的驱动。

## 设备管理模块整体结构

由于体系结构的变化，Unix V6++系统将支持ATA磁盘设备以替代原有PDP-11平台上的RK磁盘设备。同时，使用键盘与显示器替代原有的PDP-11平台上的控制台终端。与Unix V6相同，Unix V6++在设计时将设备分类为块设备与字符设备，两种类型的设备使用不同的类进行管理。由于设备差异较大，设备驱动程序将被重写，但仍沿用Unix V6中对设备的抽象方法，仅将部分方法移入设备管理类中以明确职责。



#### 图8.1设备管理模块类结构

图8.1中给出了所有整个设备管理模块中的类结构示意图。其中，设备模块管理类DeviceManager负责对整个设备管理部分的抽象，由他负责管理系统中的块设备和字符设备对应的内核数据结构。

另外，Unix V6++中使用BlockDevice类和CharDevice类分别抽象出块设备和字符设备的通用接口，而并不由这两个类实现针对某种具体块设备或字符设备的操作过程，其中的成员函数基本都采用虚函数方式定义，由其派生类负责具体操作逻辑的实现。对于某种具体设备的操作则是由继承BlockDevice类或者CharDevice类的特定设备派生类实现的，如ATA磁盘设备类ATABlockDevice，键盘与显示器使用ConsoleDevice类进行管理。

Unix V6++中使用派生类ATABlockDevice抽象系统中ATA磁盘设备，该类继承BlockDevice基类所有接口，并且针对特定ATA磁盘设备进行设备操作逻辑的实现。在添加新硬件时，只需要添加相应的派生类即可。

设备管理模块中采用继承的好处在于能够让内核使用同一的接口访问系统中的各类块设备和字符设备，而不需要涉及各类不同设备迥异的物理特性和参数设置细节，方便操作系统对设备进行统一的管理。

在原Unix V6中，使用块设备开关表bdevsw[]这一数据结构为系统中所有块设备定义统一的访问接口如设备的打开Open()、关闭Close()和处理I/O请求块Strategy()等等，每种不同的块设备在bdevsw[]占据一项，针对该种设备具体操作的函数入口地址存放在bdevsw[]对应项中，从而内核上层通过调用bdevsw[]中统一的接口执行对不同设备的特定操作逻辑。由此可见，Unix V6++中对设备管理采用C++虚函数方式，其设计思想是与Unix V6中块(字符)设备开关表一致的，同时更好地发挥出C++面向对象语言的特性。

### 设备管理模块类—DeviceManager

图8.1中的设备管理模块DeviceManager类，它使用两个数组m\_CharDevices[]与m\_BlockDevices[]分别管理系统中的块设备与字符设备(代码8.1)。

class DeviceManager

{

/\* 根据主设备号major获取相应块设备对象实例 \*/

BlockDevice& GetBlockDevice(short major);

/\* 根据主设备号major获取相应块设备对象实例 \*/

CharDevice& GetCharDevice(short major);

private:

/\* 指向块设备基类的指针数组，相当于原Unix V6中块设备开关表bdevsw[] \*/

BlockDevice \*m\_BlockDevices [MAX\_DEVICE\_NUM];

/\* 指向字符设备基类的指针数组，相当于原Unix V6中字符设备开关表cdevsw[] \*/

CharDevice \*m\_CharDevices [MAX\_DEVICE\_NUM];

};

#### 代码8.1 设备管理DeviceManager类

DeviceManager中的这两个数组，分别充当了块设备开关表bdevsw[]与字符设备开关表cdevsw[]的作用。以其中的块设备数组m\_BlockDevices[]来说，它是一个指向块设备基类的指针数组，在Unix V6++系统初始化时需要注册系统中的各种块设备，譬如系统中ATA磁盘设备ATABlockDevice类型的一个实例g\_ATADevice，我们需要将m\_BlockDevices数组中的某一项指向系统中的g\_ATADevice块设备实例，也就是将一个基类BlockDevice类型的指针指向了一个派生类ATABlockDevice类型的对象g\_ATADevice，从而实现多态。当内核调用基类BlockDevice提供的接口时，实际执行的将是派生类ATABlockDevice实现的具体操作。

在完成设备的注册后，内核可以根据主设备号，作为查找块设备开关表m\_BlockDevice[]数组的下标，找到对应的设备。在Unix V6++中，ATA磁盘设备的主设备号为0，对应占据m\_BlockDevice[0]单元。DeviceManager类提供了相应的函数接口用于根据主设备号获取对应块(字符)设备实例的引用(代码8.2)。

BlockDevice& device =

Kernel::Instance().GetDeviceManager().GetBlockDevice( major );

device.Strategy( pBuf );

#### 代码8.2 根据主设备号获取块设备实例

DeviceManager类中负责初始化时注册系统中块设备与字符设备的逻辑由其成员函数Initialize()实现(代码8.3)。

void DeviceManager::Initialize()

{

this->bdevsw[0] = &g\_ATADevice;

this->nblkdev = 1;

this->cdevsw[0] = &g\_ConsoleDevice;

this->nchrdev = 1;

}

#### 代码8.3 DeviceManager::Initialize()的实现

在Initialize()注册了ATA磁盘设备g\_ATADevice和控制台字符设备g\_ConsoleDevice，用于管理显示器和键盘这类字符输入/输出设备，并且登记系统中块设备和字符设备数量都为1。

## 磁盘设备管理类--ATABlockDevice

ATABlockDevice类用于对系统中ATA磁盘设备的管理，其继承自块设备基类BlockDevice，在基类中定义了内核对设备的统一访问接口，如设备的打开Open()、关闭Close()和处理I/O请求块Strategy()。基类BlockDevice中除了这些接口之外，还包含了指向块设备表Devtab的指针(代码8.4)。

class BlockDevice

{

public:

/\* 定义为虚函数，由派生类override实现对特定设备操作。\*/

virtual int Open(short dev, int mode);

virtual int Close(short dev, int mode);

virtual int Strategy(Buf\* bp);

virtual void Start();

public:

Devtab\* d\_tab; /\* 指向块设备表的指针 \*/

};

#### 代码8.4 BlockDevice基类

Unix V6++中块设备基类BlockDevice是将原Unix V6中设备开关与块设备表Devtab两者合并而成的，对于每一类块设备，其实例中总是包含指向对应块设备表Devtab的指针。

### 块设备表Devtab数据结构

块设备表是内核中设立用于管理各类块设备，也就是主设备号不同的各类设备使用情况的一个数据结构，它管理包括对该类设备提出的I/O请求队列和该类设备相关的缓存队列。 Unix V6++中块设备表Devtab类的定义基本与其在Unix V6中一样(代码8.5)。

class Devtab

{

public:

int d\_active; /\* 设备已经启动正在执行一个I/O请求的标志 \*/

int d\_errcnt; /\* 执行一次I/O请求时的出错计数器 \*/

Buf\* b\_forw; /\* 设备队列队首指针 \*/

Buf\* b\_back; /\* 设备队列队尾指针 \*/

Buf\* d\_actf; /\* 设备I/O请求队列的队首指针 \*/

Buf\* d\_actl; /\* 设备I/O请求队列的队尾指针 \*/

};

#### 代码8.5 Devtab类定义

在系统初始化时，还没有发生任何块设备I/O读写请求之前，对块设备表Devtab对象的初始设置应将它设备队列的队首、队尾两个指针都指向其自身，而其设备I/O请求队列的队首、队尾指针设置为空，即当前块设备没有任何正在执行的I/O请求。

### ATABlockDevice类中的函数—Strategy()和Start()

对于ATABlockDevice类而言，它需要针对ATA磁盘实现继承自BlockDevice基类的虚函数接口，对于系统中ATA磁盘，只有启动I/O操作Start()和处理I/O请求块Strategy()有意义，而Open()和Close()没有意义，为空函数。而Strategy()和Start()函数的功能主要是对磁盘管理类ATABlockDevice指向的块设备表进行的相关操作。

#### 图8.2 ATABlockDevice::Strategy() 图8.3 ATABlockDevice::Start()

Strategy()函数包含一个Buf\*类型的参数，其中记录着本次I/O请求参数信息，这些信息包括本次I/O请求的方式是读还是写，要读/写的磁盘扇区号是多少以及磁盘数据要读入到的内存目标地址多少(对于写操作则记录将内存源地址多少的数据写入磁盘)。该函数主要任务是将I/O请求块Buf送入块设备表的I/O请求队列后，随即检查块设备表的d\_active成员，判断硬盘控制器是否处于空闲。如果处于空闲状态，立即调用start()函数，启动磁盘执行刚刚送入的I/O请求。如果硬盘控制器忙碌，则立即返回(图8.2)。

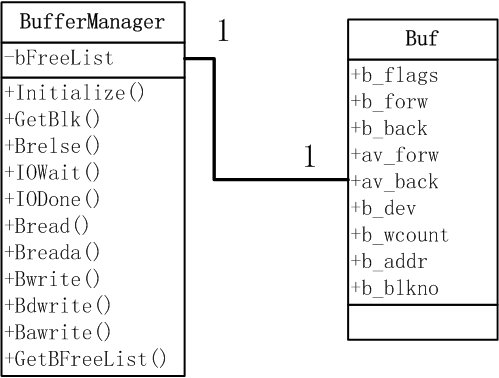
Start()函数在Strategy()函数中被调用，其功能是从块设备表的I/O请求队列中取出位于队首的I/O请求块，并设置块设备表中d\_active成员标记磁盘处于忙碌状态。然后调用设备驱动ATADriver类中的DevStart()函数，由其执行设置磁盘寄存器并驱动磁盘执行I/O请求的具体步骤(图8.3)。

## 高速缓存模块

Unix V6++高速缓存机制主要由BufferManager类进行管理。所有针对文件的操作都不会直接调用ATABlockDevice驱动磁盘读写，而是通过高速缓存BufferManager类提供的接口进行文件读写，间接调用底层设备驱动。BufferManager向上层文件系统提供了统一的访问高速缓存功能的接口。

### 高速缓存模块类结构

图8.4列出了高速缓存模块的主要类结构。Buf类为缓存控制块，在系统运行时，将会通过其b\_forw与b\_back两根指针勾连成一条双向链表，该链表由BufferManager类中的bFreeList作为头结点，Buf类中的b\_addr指向了该缓存对应的物理内存区域。



#### 图8.4 高速缓存模块类结构

在Unix V6++中设置了NBUF个缓冲存储区(该常量定义为15)，每个缓冲区占BUFFER\_SIZE个字节大小(一般为512字节)。所有的这写缓存控制块及其对应的缓冲区、自由缓存队列控制块bFreeList以及一个用于进程图像在内存和交换区之间传送的缓存控制块SwBuf都作为BufferManager的成员变量，由缓存管理模块对它们进行操作(代码8.6)。

class BufferManager

{

private:

Buf bFreeList; /\* 自由缓存队列控制块 \*/

Buf SwBuf; /\* 进程图像传送请求块 \*/

/\* 高速缓存区数据结构 \*/

Buf m\_Buf[NBUF]; /\* 缓存控制块数组 \*/

unsigned char Buffer[NBUF][BUFFER\_SIZE]; /\* 缓冲区数组 \*/

};

#### 代码8.6 BufferManager类的数据成员

### 缓存控制块数据结构—Buf类

系统中为每个缓冲区都设置一块与之对应的缓存控制块，去对它进行管理，记录它的使用信息等等。缓存控制块Buf类的定义如下(代码8.7)：

class Buf

{

public:

unsigned int b\_flags; /\* 缓存控制块标志位 \*/

int padding; /\* 4字节填充 \*/

/\* 缓存控制块队列勾连指针 \*/

Buf\* b\_forw;

Buf\* b\_back;

Buf\* av\_forw;

Buf\* av\_back;

short b\_dev; /\* 设备号，其中高8位是主设备号，低8位是次设备号 \*/

int b\_wcount; /\* 需传送的字节数 \*/

unsigned char\* b\_addr; /\* 指向该缓存控制块所管理的缓冲区的首地址 \*/

int b\_blkno; /\* 磁盘逻辑块号 \*/

int b\_error; /\* I/O出错时信息 \*/

int b\_resid; /\* I/O出错时尚未传送的剩余字节数 \*/

};

#### 代码8.7 Buf类的实现

缓存控制块Buf中的各个成员变量的含义大致如下：

1. **unsigned int b\_flags:** 标志信息，用于指示该缓存的使用情况以及I/O方式，该成员的值是以下定义的标志位中一个或者多个按位或“|”的结果(代码8.8)。

enum BufFlag

{

B\_WRITE = 0x1, /\* 写操作。将缓存中的信息写到硬盘上去 \*/

B\_READ = 0x2, /\* 读操作。从盘读取信息到缓存中 \*/

B\_DONE = 0x4, /\* I/O操作结束 \*/

B\_ERROR = 0x8, /\* I/O因出错而终止 \*/

B\_BUSY = 0x10, /\* 相应缓存正在使用中 \*/

B\_WANTED = 0x20, /\* 有进程正在等待使用该缓存\*/

B\_ASYNC = 0x40, /\* 异步I/O，不需要等待其结束 \*/

B\_DELWRI = 0x80 /\* 延迟写，将缓存要移做它用时，先将缓存写到磁盘上 \*/

};

#### 代码8.8 b\_flags中标志位

1. **int padding:** 4字节填充，在字段是在Unix V6++中新加入的成员，其目的是保证b\_forw和b\_back在Buf类中与Devtab类中的字段顺序(或者说内存布局)能够一致，否则强制转换会出错。
2. **Buf\* b\_forw;**

**Buf\* b\_back;**

**Buf\* av\_forw;**

**Buf\* av\_back;**

缓存控制块队列勾连指针，这是两个队列的勾连指针，任何一个缓存控制块都可以同时存在在两个队列中，每个队列用一对指针实施双向勾连。

1. **short b\_dev:** 表示设备号，该变量指定它所管理的缓冲区属于哪个设备的。它有两个部分组成，高8位指定了主设备号，而低8位则指定了此设备号。
2. **int b\_wcount:** 记录本次I/O请求需要传送的字节数，通常该值等于一个缓冲区的大小，即512字节。
3. **unsigned char\* b\_addr：**指向该缓存控制块所管理的缓冲区的首地址，即令一个缓存控制块与一个缓冲区建立对应关系。
4. **int b\_blkno：**磁盘逻辑块号，该变量指定缓冲区中的数据对应磁盘中哪一个扇区的数据。
5. **int b\_error：**用来记录I/O操作时的错误信息，在Unix V6++中并没有用到该成员。
6. **int b\_resid：**记录 I/O出错时尚未传送的剩余字节数，与上面的b\_error一样，仅仅是取自于原Unix V6中而在Unix V6++中并未实际用到。

以上是对缓存控制块数据结构Buf类中各个成员及其含义的解释，可以看到不仅包含了对缓存的有关管理控制信息，也包括了I/O请求信息。因此，缓存控制块Buf类不仅是缓存控制块，还是针对该缓存进行的I/O请求块。

此外，内核中设置一个自由缓存队列控制块bFreeList，与一般缓存控制块不同的是它没有与之对应的缓冲区。该缓存控制块的作用是作为系统中自由缓存队列的队首节点，自由队列用于勾连系统中所有处于空闲状态，可供重新分配使用的缓存，bFreeList中的一对队列指针av\_forw、av\_back分别指向自由缓存队列的队首和队尾结点。

### 高速缓存管理

###### 缓存控制块队列管理

系统中设置了自由队列和设备队列进行对缓存控制块的管理。因为缓存控制块及其对应的缓冲区是一一绑定关系，所以后面直接使用缓存这一术语代表这两者。

* 自由队列

一般而言，系统中每一个可被分配作它用的缓存都应该位于自由队列中。在此队列中，所有缓存的b\_flags标志皆不包含B\_BUSY标志。自由缓存队列的控制块结点是bFreeList。bFreeList和队列中各缓存控制块相互用指针av\_forw、av\_back双向勾连(图8.5)。

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

bfreelist

buf[….]

buf[….]

buf[….]

#### 图8.5 自由缓存队列

* 设备队列

每类块设备都有一个设备缓存队列，其队首、队尾分别由相应设备表中的b\_forw和b\_back指示。队列中各缓存控制块用指针b\_forw、b\_back双向勾连(图8.6。一个缓存被分配用于读、写某类块设备上的某一个字符块时，其相应缓存就进入该块设备缓存队列中，除非将其再分配移作他用，否则它一直停留在该队列中。

在设备缓存队列中，每个缓存总与该类一个设备上的某个字符块相关连，该设备的设备号是b\_dev，字符块块号为b\_blkno。所谓相关连是指两者的内容或者现在或者在将来某一时刻是相同的。

NODEV队列是个特殊的设备缓存队列。当系统需使用缓存，但它不与特定的设备字符块相关连时，将分配到的缓存控制块缓存送入NODEV队列中。其队列控制块也是bFreeList，但它与自由队列不同，用对首、队尾指针b\_forw、b\_back。在Unix V6++中有两种情况将缓存送入NODEV队列。一种是在进城执行一个目标程序的开始阶段，它用缓存存放传向该目标程序的参数；另一种情况是用缓存存放文件系统的资源管理块(SuperBlock)。另外，一个自由缓存控制块，不但在自由缓存队列中，同时一定在某个设备缓存队列中，为此，在系统初启阶段将它们全部送入NODEV设备队列。

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

devtab

buf[….]

buf[….]

buf[….]

#### 图8.6 设备缓存队列

###### 缓存管理算法

系统内有限的缓存要被所有进程共享使用，为了提高其使用效率，选择适当的缓存管理算法是非常重要的。Unix V6++采用的缓存淘汰算法是极为精确的LRU算法。

一个缓存刚被分配用于读、写某块设备上某字符块时，相应缓存控制块的b\_flags中的B\_BUSY标志，它一定位于相应的设备缓存队列中，而不在自由队列中。

一旦读、写操作结束就释放缓存。在这里“释放”指的是清相应缓存控制块的b\_flags中的B\_BUSY标志，并将它送入自由缓存队列结尾，表示它可被考虑移作他用。但是该缓存仍留在原设备缓存队列中。所谓操作读操作结束，就是将所需的字符块已经由缓存读入了用户指定的内存区。如果所需字符块原先已经在某一缓存中，则不需要进行读块设备操作；如果不在一个缓存中，则先要将相应块设备上的字符块传送到某个缓存中，再从缓存中传送到用户目标区。所谓“写”结束是指经由缓存将用户指定的内存区的信息写到了指定设备的某个字符块上。但是如果缓存没有写满，那么考虑到以后可能还会继续写下去，所以不急于立即进行写块设备操作，而是设置b\_flags中的B\_DELWRI标志，表示具体的写块设备操作推迟到某个恰当时刻进行。考虑到缓存是系统中竞争非常剧烈的资源，要使其被尽可能地共享，所以也请其B\_BUSY标志，将它释放入自由缓存队列。

一个可以移作它用的缓存控制块，既留在原设备缓存队列中，又进入自由缓存队列，这种安排的好处是：

(1) 在自由队列中的缓存，只要还没有重新分配就保持其原有内容不变，因此如果需要，只要简单地将相应缓存从自由队列中抽出，就可按原状继续使用它。这样，对读、写操作而言，都避免了重复而又十分耗费时间的设备I/O操作过程，这就大大提高了文件系统工作的效率。

(2) 如果要将一个缓存重新分配移作他用，则只需将它从自由缓存队列和原设备缓存队列中同时抽出，送入新的设备缓存队列。这样就实现了进程对有限缓存的共享。

为了使得一个已被释放的缓存尽可能长地保持原来的使用状态，将它送入自由缓存队列的尾部；而分配缓存时又从自由缓存队列首部取出。当一个缓存在自由队列内移动时，只要有按原状使用它的需要，就立即将他从自由队列中抽出。当它再次被释放时又进入自由缓存队列末尾。这样就保证了在所有自由缓存中，淘汰最后一次使用时间离现在时刻最远的一个缓存内容。这既是LRU算法。

已经设置了B\_DELWRI延迟写标志的缓存，虽然已经在自由队列中，实际上其内容尚未写到相应的块设备上去。当它已经移到自由缓存对首准备移作他用时，不应将其按一般自由缓存处理，即不能立即对它进行重新分配，而是要提出I/O请求，以便将其内容写到相应设备的字符块上去。为此，将它从自由队列中抽出，而只留在原设备队列中。写操作结束后，这种缓存又被释放进入自由缓存队列末尾，同时仍保留在原设备队列中。

###### 缓存队列的初始化

系统初启阶段对高速缓存初始化，所有的缓存控制块按序号从0到(NBUF - 1)排在自由缓存队列中；按序号(NBUF - 1)到0排在NODEV队列中，其中NBUF是缓冲区的数目。

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

**.....**

b\_back

b\_forw

**.....**

bfreelist

buf[0]

buf[NBUF-2]

buf[NBUF-1]

#### 图8,7 NODEV缓存队列初始状态

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

**.....**

av\_back

av\_forw

**.....**

bfreelist

buf[0]

buf[1]

buf[NBUF-1]

#### 图8,8 自由缓存队列初始状态

###### 缓存申请

当进程需要对某设备dev上的一个字符块blkno信息进行处理前，先要为其分配一缓存。根据dev和blkno获得缓存的途径有两种：

1. 在主设备号为Utility::GetMajor(dev)的设备队列中搜索，如果找到一个缓存其b\_dev和dev相同且b\_blkno与blkno相同，则说明所需的缓存已经存在，因此不必再另外分配。但是该缓存的使用情况可能有两种：
   1. 它处于空闲状态(B\_BUSY标志没有设置)，于是只要将它从自由缓存队列中抽出，并增设B\_BUSY标志，然后就可以按其原状继续使用。
   2. 它正被某个进程使用(B\_BUSY标志已设置)，则在其b\_flags中增设B\_WANTED标志，表示有进程正在等待使用它。然后调用GetBlk(dev,blkno)的进程进入睡眠状态，待该缓存使用完毕被释放时再被唤醒。
2. 如果在上述设备缓存队列中没有找到相关缓存，那么就需要在自由缓存队列中进行分配。自由缓存队列的情况可能是：
   1. 自由缓存队列空。此时在bFreeList的b\_flags中增设B\_WANTED标志，表示有进程正在等待使用自由缓存。然后，调用GetBlk(dev,blkno)程序的进程进入睡眠状态，待某个缓存被释放进入自由缓存队列时再被唤醒。
   2. 自由缓存队列非空。此时从其对首取出一缓存，并增设其B\_BUSY标志。但是若该缓存的延迟写标志DELWRI已经设置，则将他所管理的缓存内容以异步方式写到相应设备上去，要求分配缓存的进程立即重复分配缓存操作。若此缓存的延迟写标志没有设置，则表示这确实是一个可以重新分配的缓存。于是，将他从原设备缓存队列中抽出，并从首部插入主设备号为Utility::GetMajor(dev)的设备缓存队列。最后将缓存的b\_dev和b\_blkno分别设置为dev和blkno，建立起了它与相应设备上一个指定字符块的连接关系。

内核中实施这一功能的程序BufferManager::GetBlk(dev,blkno)，其流程图如下(图8.9)。



#### 图8.9 GetBlk(dev, blkno)

###### 缓存释放

缓存使用完毕后，立即被释放。此时，它仍留在原设备buf队列中，但同时进入自由buf队列末尾。释放缓存的程序是BufferManager::Brelse(pBuf)，参数为指向一欲释放的缓存控制块的指针。其流程图如下(图8.10)。



#### 图8.10 Brelse(pBuf)

### 高速缓存模块与设备驱动

图8.11从所处内核层次以及相互之间调用关系角度描述了高速缓存模块和块设备驱动两者之间的关联。可以看到，设备驱动位于高速缓存管理模块下层，上层的高速缓存模块通过调用驱动层提供操作块设备的操作接口，来完成字符块设备的读、写操作。同时告诉高速缓存模块还需管理自己模块内部的数据结构，维护系统中缓存资源的分配和释放等操作。

GetBlk

Brelse

Bread

Bwrite

IOWait

IODone

Breada

Bdwrite

Bawrite

高速缓存管理模块

ATA磁盘设备驱动

Open()

Close()

Strategy()

Start()

#### 图8.11 高速缓存模块与ATA磁盘驱动

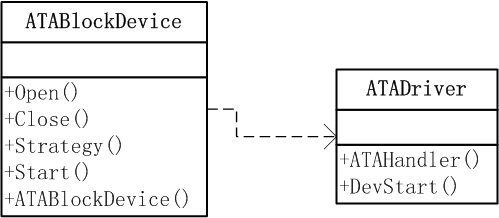
因为高速缓存管理模块BufferManager与底层体系结构关系不大，所以基本保留原Unix V6中的数据结构和算法思想。高速缓存的大部分抽象只是函数名细微变化与从属关系的细化，并不牵涉到算法与程序逻辑的修改，因此这里并不对其中的各个块设备读、写函数一一展开介绍。

## ATA磁盘设备驱动

如前所述，ATABlockDevice类实现了对ATA磁盘设备的操作函数，其主要作用是对块设备表Devtab及其相关的设备队列和I/O请求对列进行的相关操作。应当说，ATABlockDevice完成了对系统中ATA磁盘设备的管理任务，它和ATADriver类共同组成ATA设备驱动模块，两者的同属于设备驱动的范畴，只是侧重有所不同。

ATABlockDevice及其所管理的块设备表Devtab侧重于ATA磁盘设备在内核中的数据结构表示，譬如块设备表Devtab就是在内核中设立的进行对ATA磁盘管理的软件层面抽象；而ATADriver类在驱动模块中侧重于对磁盘控制器硬件的抽象和操作。

具体来说，ATABlockDevice在完成了对硬件设备内存数据结构的一系列操作之后，将具体启动设备进行I/O操作的工作交由ATADriver类来完成，由该类负责具体对磁盘控制器的设置和响应工作，两者协作共同完成对ATA磁盘设备的驱动(图8.12)。



#### 图8.12 ATA设备驱动类结构

在ATADriver类中包含的两个重要函数分别是DevStart()和ATAHandler()。其中，DevStart()根据本次I/O请求的参数，设置ATA磁盘控制器中相应控制寄存器，然后启动磁盘设备执行本次I/O请求任务。ATAHandler()则为磁盘中断处理函数，磁盘在执行完一次I/O请求之后会向CPU发出中断，告知本次I/O执行完成，磁盘中断处理函数就是用于在完成一次I/O操作后进行相应的善后处理工作。

DevStart()中使用到了较多关于磁盘控制器以及DMA编程的内容，所以先对基础知识进行介绍(关于Bootloader文档中对此也有涉及)。

## 磁盘控制器&DMA编程

### Polling、中断方式I/O、DMA

一般来说，对磁盘控制器编程来实现数据在磁盘与内存之间传送，主要有以下三种方式：

1. **Polling(轮询)：**CPU向磁盘控制器发送读/写命令，启动一次I/O操作。由于磁盘这类物理设备读写操作需要一定时间，而CPU并不能知道本次操作需要多少时间，于是在本次I/O操作完成之前唯一可以做的就是周期性地询问磁盘控制器“操作是否已经完成”；对于读操作，也就是是否已经完成读取扇区操作，数据已在缓冲区中。
2. **Interrupt-driven I/O(中断方式I/O)：**CPU向磁盘控制器发送读/写命令启动一次I/O操作，然后CPU可以继续执行后续指令，而并不需要轮询磁盘控制器检查操作是否完成。等到I/O操作完成之后，磁盘控制器会向CPU发出中断告知本次I/O操作完成，数据已经在缓冲区中。
3. **DMA(Direct Memory Access, 直接内存访问)：与中断方式I/O类似，由CPU向磁盘控制器和DMA控制器发送命令启动一次I/O操作，较中断方式I/O更先进的地方在于，整个数据传送不需要CPU介入，基本不占用CPU周期；而是由DMA控制器操纵直接在磁盘与内存之间进行**成块**数据传送。CPU在此过程中可以执行后续指令，直至本次I/O操作完成后会向CPU发送中断信号，此时磁盘数据已经在内存中。**

以上三种方式中，除了DMA方式中CPU收到中断表示数据已经读入到内存中(对读操作而言)，Polling以及Interrupt-driven I/O中读取完成并不表示数据已经在内存中，仅仅意味数据已经从扇区中读入到磁盘控制器缓冲区中，我们还需要将它从缓冲区读入内存。

在Unix V6++中，磁盘驱动实现了DMA方式的数据传输，相比Polling以及Interrupt-driven I/O方式中数据经过磁盘控制器缓冲区中转更有效率，代码逻辑更清晰，但额外的代价是我们除了需要对ATA控制器编程之外，还要对DMA控制器进行编程。

### ATA磁盘控制器寄存器接口

首先，磁盘控制器是什么？从硬件角度来说，当然是一块芯片；而从软件角度来说，磁盘控制器可以被认为是由一系列寄存器组成的，包括：状态寄存器、命令寄存器、数据寄存器、扇区号寄存器等，还加上用于临时存储数据的控制器缓冲区。我们通过对寄存器写入恰当的数值来发送I/O操作命令，告诉控制器要读写的扇区号是多少，操作方式为读或写，并通过读取状态寄存器的值了解操作是否成功完成等，所以说将磁盘控制器抽象为一组寄存器是比较恰当的。

其次，如何访问这些不同的寄存器？每个寄存器都有其特定的端口地址，我们通过IN/OUT指令访问指定的端口地址就可以访问到这些寄存器，对其写入或读取数据。至于每个寄存器具体的端口地址则是系统体系结构规范确定的。

下面表8.2中给出了ATA磁盘控制寄存器的端口地址和用途。表中有两个寄存器共用一个端口地址的情况：在读操作时，访问一个寄存器；而在写操作时，访问另一个寄存器。这样安排是为了节省I／O地址资源。

**表8.2 ATA磁盘控制寄存器端口及用途**

|  |  |  |
| --- | --- | --- |
| 端口 | 读操作 | 写操作 |
| 0x1f0 | 数据寄存器 | |
| 0x1f1 | 错误寄存器(错误状态) | 写前预补偿寄存器 |
| 0x1f2 | 扇区数寄存器 | |
| 0x1f3 | 扇区号寄存器 | |
| 0x1f4 | 柱面号寄存器(低位) | |
| 0x1f5 | 柱面号寄存器(高位) | |
| 0x1f6 | 驱动器/磁头寄存器——驱动器号/磁头号 | |
| 0x1f7 | 主状态寄存器 | 命令寄存器 |
| 0x3f6 | 辅助状态寄存器 | 磁盘控制寄存器 |
| 0x3f7 | 驱动器地址寄存器 | ---- |

这里对表8.2中给出的各个寄存器进行说明：

1. **数据寄存器(0x1f0，读/写)**

这是一对16位数据传输器，用于扇区读、写和磁道格式化操作。该寄存器用于在内存和控制器缓冲区之间进行32位数据交换，即每次4字节。在使用polling和中断方式I/O传输时要使用该寄存器。当要进行一个扇区512直接的数据读取时，应当使用rep insd指令将控制器缓冲区中的数据经由数据寄存器(0x1f0)读入内存。

1. **错误寄存器(0x1f1，读)/写前预补偿寄存器(0x1f1，写)**

这是一个8位的寄存器。读操作时，它反映控制器在诊断方式(在加 电、复位或执行驱动器诊断命令时所处的方式)或操作方式(执行除诊断命令以外的所有命令时所处的方式)下的错误原因。例如，在操作方式下，该寄存器的最高位为1表示访问的扇区是坏扇区，最低位为1表示地址标志没有找到。只有当主状态寄存器(0x1f7)的位0等于1时该寄存器的数据才有效指示错误状态。

在写操作时，作为写前预补偿寄存器，该寄存器的内容等于柱面号除以4，表示开始写预补偿的起始柱面号。现在的磁盘基本已不使用该寄存器。

#### 表8.3 错误寄存器状态含义

|  |  |  |
| --- | --- | --- |
| 值 | 诊断命令时 | 其它命令时 |
| 0x01 | 无错误 | 数据标志丢失 |
| 0x02 | 控制器出错 | 磁道0错 |
| 0x03 | 扇区缓冲区错 | ---- |
| 0x04 | ECC部件错 | 命令放弃 |
| 0x05 | 控制处理器错 |  |
| 0x10 | ---- | ID未找到 |
| 0x40 | ---- | ECC错误 |
| 0x80- | ---- | 坏扇区 |

1. **扇区数寄存器(0x1f2，读/写)**

该寄存器记录要访问的扇区的数目。用于多扇区操作时，每访问一个扇区，其值自动减1。如果初值为0，则表示扇区数是256。

LBA(Logical Block Addressing,逻辑块寻址)采用从0开始的数值对磁盘扇区进行编号，允许程序使用绝对的扇区号访问磁盘，而不再是原先依据磁盘物理结构的“磁柱-磁头-扇区(CHS)”寻址模式。在访问磁盘时，由 IDE 控制器把逻辑扇区号转换为实际磁盘对应的柱面、磁头、扇区等参数。

1. **扇区号寄存器(0x1f3，读/写)**

该寄存器记录要访问的起始扇区号。用于多扇区操作时，每完成一个扇区读写操作寄存器的值自动加1。如果使用LBA28寻址(逻辑块寻址)方式，它记录起始逻辑扇区号的第7～0位。

1. **柱面号寄存器(0x1f4，0x1f5，读/写)**

这是由两个8位寄存器组成的16位寄存器，0x1f4为低字节，0x1f5为高字节。它用来记录要操作的柱面的编号。如果使用LBA28寻址方式，这两个寄存器分别存放起始逻辑扇区号的第15～8位和第23～16位。

1. **驱动器／磁头寄存器(0x1f6，读/写)**

该寄存器记录要访问的驱动器号、磁头号以及寻址方式。对于IDE接口的磁盘而言，其中第7位和第5位必须置1，而第6位置1时开启LBA模式，否则就用“磁柱-磁头-扇区(CHS)”寻址方式。第4位为驱动器选择，指定此次访问的对象是主盘还是从盘，置0时访问的主盘，置1时访问的从盘。当采用LBA模式时第3位~第0位表示LBA28位中的第24~27位。

1. **主状态寄存器(0x1f7，读)/命令寄存器(0x1f7，写)**

在读时，该端口地址对应主状态寄存器，反映磁盘控制器在执行命令后的状态。各状态位含义如表8.4所示。

#### 表8.4 主状态寄存器各位含义

|  |  |
| --- | --- |
| 值 | 说明 |
| 0x01 | 命令执行错误 |
| 0x02 | 收到索引 |
| 0x04 | ECC校验错误 |
| 0x08 | 请求服务，表示可传输数据 |
| 0x10 | 寻道结束 |
| 0x20 | 驱动器故障 |
| 0x40 | 驱动器准备好 |
| 0x80- | 控制器忙碌 |

在写时，该端口地址对应命令寄存器，该寄存器接收并保存CPU发送到控制器的命令。命令一旦发送到命令寄存器中，该命令就会开始执行。

#### 表8.5 命令寄存器命令值

|  |  |  |  |
| --- | --- | --- | --- |
| 命令码 | | 命令默认值 | 命令说明 |
| 高4位 | D3 D2 D1 D0 |
| 0x1 | R R R R | 0x10 | 驱动器重新校正 |
| 0x2 | 0 0 L T | 0x20 | 读操作 |
| 0x3 | 0 0 L T | 0x30 | 写操作 |
| 0x4 | 0 0 0 T | 0x40 | 扇区检验 |
| 0x5 | 0 0 0 0 | 0x50 | 格式化磁道 |
| 0x6 | 0 0 0 0 | 0x60 | 控制器初始化 |
| 0x7 | R R R R | 0x70 | 寻道 |
| 0x8 | 0 0 0 0 | 0x80 | 控制器诊断 |
| 0x9 | 0 0 0 1 | 0x90 | 建立驱动器参数 |

在表8.5中命令码的低4位是附加参数：

R是步进速率。R=0，则步进速率35us；R=1为0.5ms，以此量递增。L是数据模式。L=0表示读/写扇区为512B；L=1表示读/写为512B加4B的ECC码。T是重试模式。T=0表示允许重试；T=1则禁止重试。

然而实际上大多数情况下我们只需要使用默认的命令值即可。即对于读操作命令值为0x20，写操作对应命令值0x30。

1. **辅助状态寄存器(0x3f6，读)/ 磁盘控制寄存器(0x3f6，写) (未使用)**

在读时，它与状态寄存器的内容相同，但读该寄存器时不会清除中断请求信号。在写时，用于存放磁盘控制字节并控制复位操作。

1. **驱动器地址寄存器(0x3f7，读) (未使用)**

在ATA-1标准中该寄存器用来保存命令执行后的某些信息。在ATA-2标准中没有定义该寄存器。

### DMA控制器编程

###### DMA编程的数据结构—PRD和PRD Table

对DMA控制器的编程需要用到一个叫做PRD(Physical Region Descriptor，物理内存区描述符)的数据结构。该数据结构用于描述进行DMA数据传输时，源(或目标)物理内存区域起始地址、 长度(图8.13)。

0

物理内存区起始地址[31 : 1]

0

Byte 2

Byte 3

Byte 1

物理内存区

Byte 0

DWord 0

DWord 1

传输字节数[15 : 1]

保留

EOT

#### 图8.13 PRD(Physical Region Descriptor，物理内存区描述符)

每个PRD描述符大小为8字节，低四字节为物理内存区的起始地址，随后的2字节记录DMA要传输的字节数，保留区域占据最高2字节中的低15位，而EOT位为End of Table位，表示当前PRD描述符是本次DMA请求所涉及一系列PRD描述符表的最后一项，即要进行传输的最后一个物理内存区域。

此外，PRD描述符定义中物理内存区起始地址和传输字节数最低位默认为0，说明每次进行DMA传输的内存区域必须是2字节对齐的。

Unix V6++中使用PhysicalRegionDescriptor类对PRD描述符进行封装，其成员变量的顺序按照PRD描述符规范中内存字节从低到高依次定义，并且使用\_\_attribute\_\_((packed))修饰符确保每个PRD描述符对象大小严格为8字节(代码8.9)。

class PhysicalRegionDescriptor

{

private:

unsigned long m\_BaseAddressZeroBit : 1;

unsigned long m\_MemoryRegionPhysicalBaseAddress : 31;

unsigned short m\_ByteCountZeroBit : 1;

unsigned short m\_ByteCount : 15;

unsigned short m\_Reserved : 15;

unsigned short m\_EOT : 1;

}\_\_attribute\_\_((packed));

#### 代码8.9 PhysicalRegionDescriptor类

一次DMA传输可以跨越多个物理内存区，每个内存区对应一个PRD描述符，一个或多个PRD可以构成描述符表，称为物理内存区描述符表(PRD Table)。DMA控制器进行一次DMA传输时以物理内存区描述符表为单位，每启动一次DMA传输时，DMA控制器从PRD Table的第0项开始，依次读/写表中每一个PRD描述的内存区域，直至DMA控制器检测到表中第n个PRD的Bit(31)EOT位为1，则认为PRD Table结束，才算完成DMA本次传输。

在Unix V6++中，使用了PRDTable类对物理内存区描述符表进行封装，其中的主要成员变量是一个PRD描述符数组m\_Descriptors[]，我们使用\_\_attribute\_\_((aligned (4)))修饰符确保PRD Table数据结构在内存中的起始地址是4字节对齐的 (代码8.10)。

class PRDTable

{

public:

static const int NSIZE = 10; /\* PRD Table中描述符的最大允许数目 \*/

private:

PhysicalRegionDescriptor m\_Descriptors[NSIZE] \_\_attribute\_\_((aligned (4)));

};

#### 代码8.10 物理内存区描述符表PRDTable

PRD Table的起始地址必须4字节对齐是由于DMA控制芯片内部的物理区域描述符表寄存器(PRDTR)，该寄存器存放PRD Table的Base Address中的[31 : 2]位,而忽略掉[1 : 0]两位，因此需要4字节对齐。

PRD Table中相邻两个描述符，它们所描述的物理内存区可以是不连续的，但是这两个描述符自身必须在内存上是连续的，因此PRDTable类中使用数组的形式来实现描述符表。

###### DMA控制器的寄存器接口

DMA控制器的寄存器接口包含命令寄存器、状态寄存器、PRD Table基地址寄存器这三个。

* 命令寄存器

用于向DMA控制器发送命令启动一次DMA操作，其格式如图8.14所示。命令寄存器的Bit(3)是读/写位，告知DMA控制器进行DMA传输的方向。Bit(3) = 0时表示写硬盘，读内存；Bit(3) = 1；表示读硬盘，写入内存。命令寄存器的Bit(0)为DMA操作启动开关，设置Bit(0) = 1则立即启动一次DMA；Bit(0) = 0；停止正在执行的DMA。通常，在执行完前一次DMA之后也需由软件指令将Bit(0)置为0。

保留

保留

保留

保留

R/W

保留

保留

Start/Stop

5

7

4

6

3

2

1

0

#### 图8.14 命令寄存器

* 状态寄存器

状态寄存器是用来指示DMA控制器当前状态，让程序根据其状态决定下一步要执行的操作，其格式如图8.15所示。

该寄存器的Acitve位Bit(0)在DMA操作执行期间硬件置1，DMA完成后由硬件自动置0。Error位Bit(1)：DMA执行过程中如果出错，硬件将该位置1，软件指令通过写入一个“1”将该比特位复位到0。Interrupt位Bit(2)：当DMA传输完成，并且外设已向CPU发出中断请求(不论CPU是否关中断)之后，由硬件置为1，表示已经发出中断请求。软件指令通过写入一个“1”将该比特位复位到0。

D0DC

Simplex

保留

D1DC

保留

Interrupt

Error

Active

5

7

4

6

3

2

1

0

#### 图8.15 状态寄存器

* PRD Table基地址寄存器

PRD Table基地址寄存器（Physical Region Descriptor Table Register，PRDTR）的格式如图8.16所示。它指向物理内存中PRD Table的起始地址，DMA控制器通过该起始地址找到并依次传输PRD Table中的各个PRD项，直至表中最后EOT位为1的末尾项。该寄存器长度为32位，由于DMA控制器忽略其最低2位，因此要求内存中的PRD Table满足4字节对齐。

PRDTable物理起始地址[31 : 2]

保留

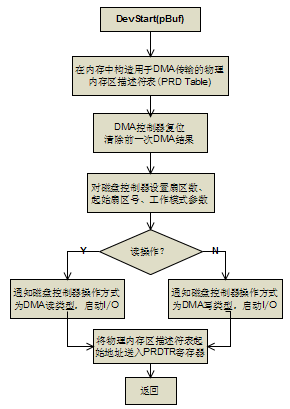
[31 : 2]

[1 : 0]

#### 图8.16 PRD Table基地址寄存器

### 设备启动函数—DevStart()

DevStart()根据本次I/O请求的参数，设置ATA磁盘控制器中相应控制寄存器，然后启动磁盘设备执行本次I/O请求任务。该函数的大致流程如下(图8.17)。



#### 图8.13 DevStart()设备启动函数

在向磁盘控制器发送命令之前，先要检查其是否就绪，如果控制器长时间未能进入就绪状态，表示出错，则没有必要进行后续操作。

在控制器响应的前提下，就要进入到对磁盘控制器和DMA控制器两者编程的阶段了。首先，在构造用于DMA传输的数据结构—物理内存区描述符表(PRD Table)，并根据本次I/O请求的参数初始化该表中的PRD项。对于Unix V6++中而言，没有一次DMA传输需要跨越多个物理内存区域的情况，所以通常每次PRD Table中只有一项。

在PRD Table准备就绪之后，复位DMA控制器。然后，对磁盘控制器发送一系列命令字，包括一次要读/写的扇区数，读/写的起始逻辑扇区号和磁盘控制器的工作模式之类。

然后将本次I/O请求的操作方式：读或者写同时告知磁盘控制器和DMA控制器，它们两者都需要清楚知道本次I/O请求的数据传输方向：从磁盘到内存进行数据传输或相反。最后将之前建立用于DMA传输的数据结构—PRD Table的物理起始地址装入DMA的物理内存区描述符表基地址寄存器(PRDTR)中，启动DMA控制器上的开关就开始数据传输了。至此DevStart()函数完成了启动一次I/O请求所需的全部工作。