操作系统　第五章　外设管理

5.2 磁盘高速缓存、磁盘驱动 和 磁盘中断处理程序

同济大学计算机系

磁盘很慢

Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

磁盘很慢，所以操作系统内核将最近访问过的磁盘数据块缓存在内存中。

磁盘高速缓存池，Unix V6++为例

- 磁盘高速缓存池 由 若干缓存块组成。每一个缓存块存放一个磁盘数据块。

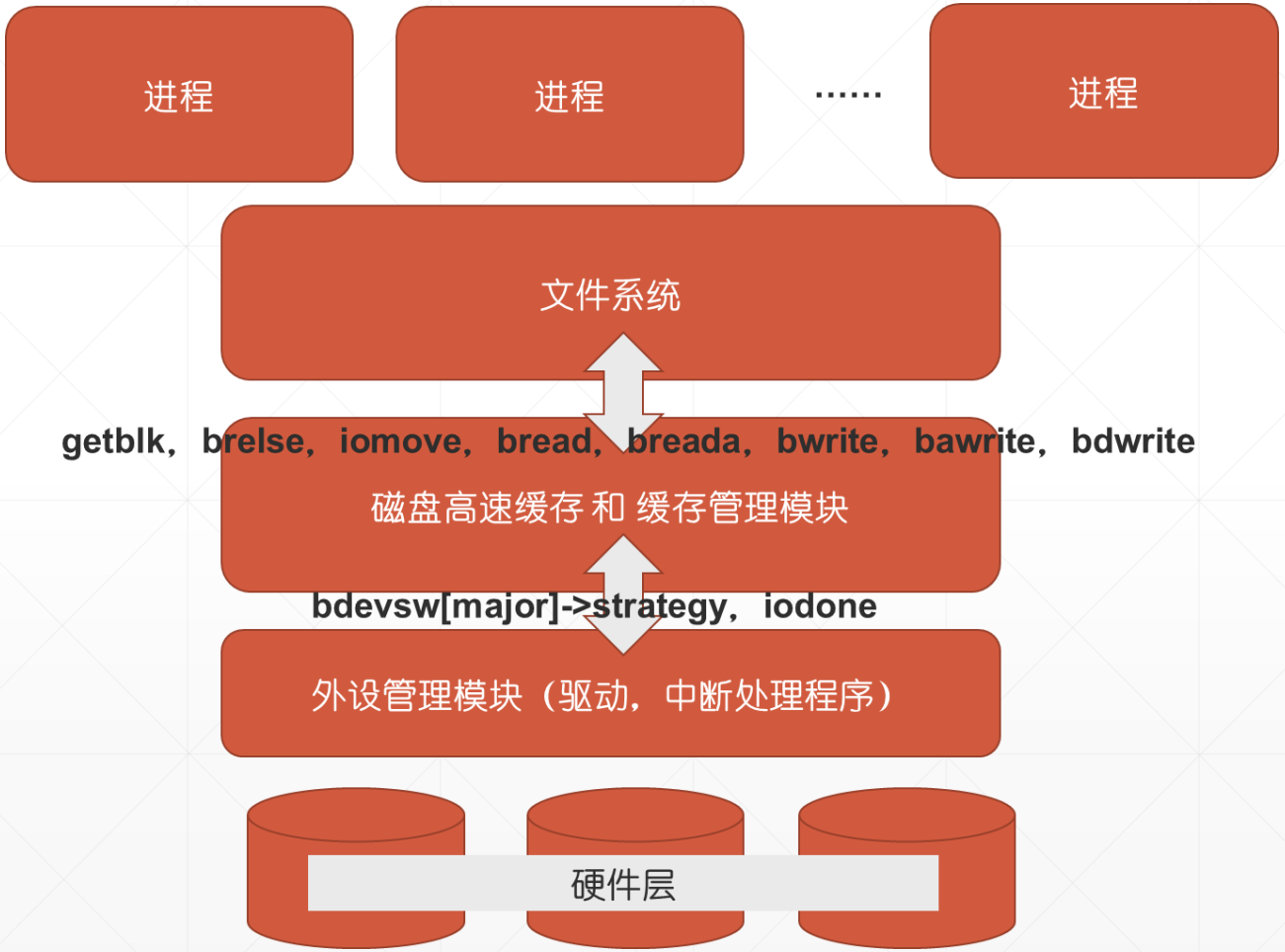
```
class BufferManager   // 缓存池
{
public:
    static const int NBUF = 15;
    static const int BUFFER_SIZE = 512;
      ……
    Buf m_Buf[NBUF];
    unsigned char Buffer[NBUF][BUFFER_SIZE];
}
```

Buffer[i], i#缓存块
m_Buf[i], i#缓存块的控制块
m_Buf[i].b_addr == Buffer[i]

- 磁盘高速缓存池，所有磁盘共享。每个缓存块有一个缓存控制块，记录缓存块中存放的是哪张磁盘的哪块数据。

```
class Buf  // 缓存控制块
{
    short  b_dev;  // 磁盘设备号
    int      b_blkno;  // 数据块号，存放这个数据块的扇区号
    unsigned char* b_addr;  // 缓存块的首地址
      ……
}
```

磁盘高速缓存在内核中的位置



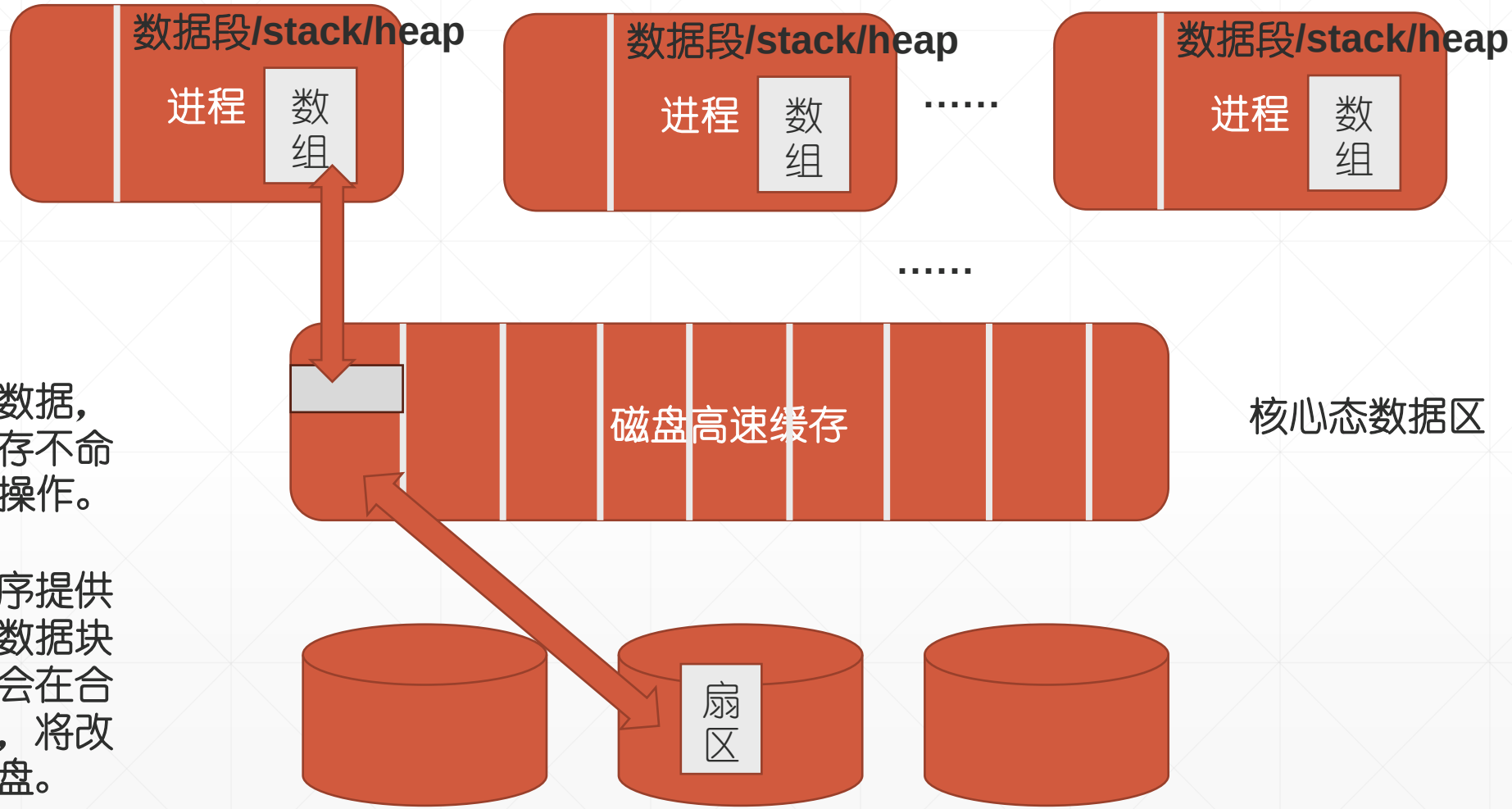
文件系统访问磁盘高速缓存中保存的磁盘数据，不直接向磁盘发IO命令。

缓存管理模块，为数据块分配缓存块，统筹IO操作，提升系统性能。

Unix V6++文件管理系统 和 外设管理系统架构

基于缓存的IO —— 应用程序读写磁盘文件

用户态数据区（应用程序的全局/局部变量）

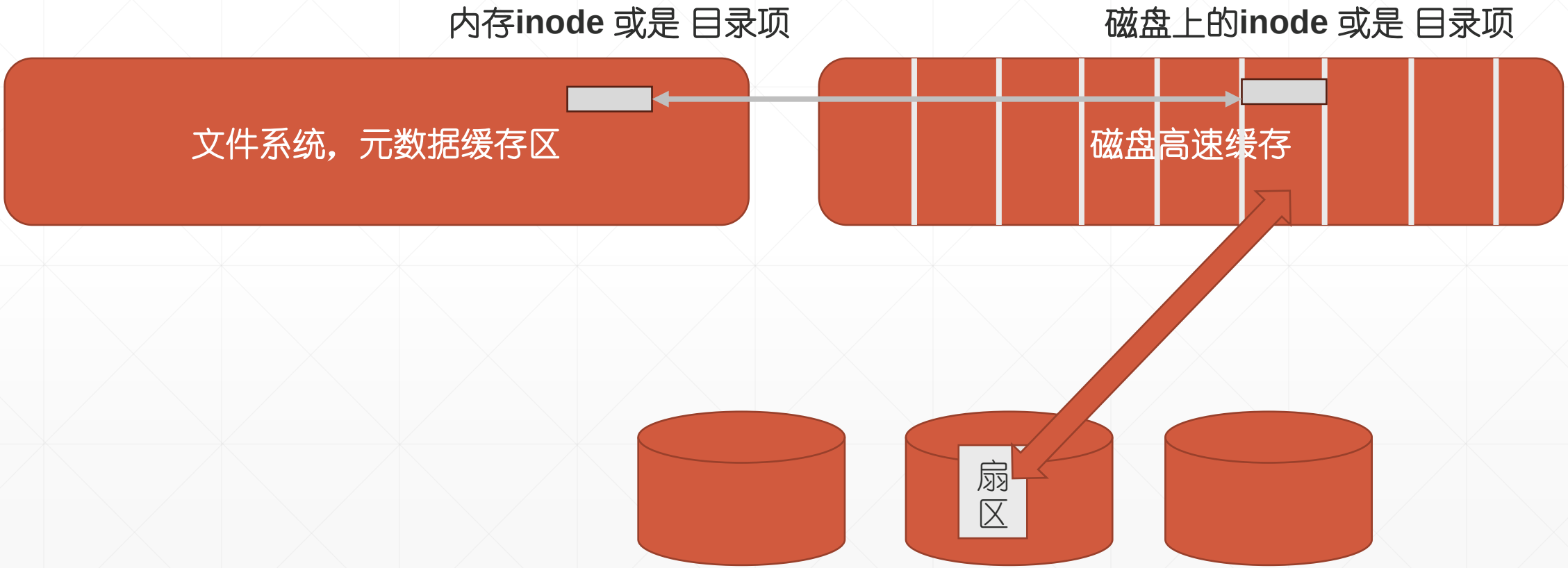


核心态数据区

读操作，从缓存块中取数据，送给应用程序。只有缓存不命中时，系统才会执行IO操作。

写操作，只需将应用程序提供的数据写入分配给目标数据块缓存块。缓存管理模块会在合适的时机执行后台操作，将改写过的数据块写回磁盘。

基于缓存的IO —— 文件系统读写文件系统元数据



Unix V6（V6++）高速缓存管理 —— 缓存的使用状态

```
class Buf
{
    short       b_dev;
    int         b_blkno;
    unsigned char*    b_addr;
    unsigned int      b_flags;
    int               b_error;
    int               b_resid;
    Buf*        b_forw;
    Buf*        b_back;
    Buf*        av_forw;
    Buf*        av_back;
};
```

```
enum BufFlag    /* b_flags中标志位 */
{
    B_WRITE     = 0x1,      /* 写操作。缓存中的信息写硬盘 */
    B_READ      = 0x2,      /* 读操作。硬盘信息读入缓存 */
    B_DONE      = 0x4,      /* I/O操作结束，缓存数据可用 */
    B_ERROR     = 0x8,      /* I/O出错 */
    B_BUSY      = 0x10,     /* 忙，有进程正在访问缓存中的数据 */
    B_WANTED    = 0x20,     /* 有进程等待使用缓存中的数据，清B_BUSY标志时，唤醒这些进程 */
    B_ASYNC     = 0x40,     /* 异步I/O，  */
    B_DELWRI    = 0x80      /* 延迟写，相应缓存移做他用时将其内容写回磁盘 */
};
```

B_DELWRI，延迟写，又叫脏标识。缓存块被写过，但尚未写回磁盘。

B_ASYNC，异步IO标识。没有进程等待，IO完成后中断处理程序解锁缓存，不执行WakeUpAll操作。

Unix V6（V6++）高速缓存管理

```
class BufferManager
{
    Buf bFreeList;
    Buf SwBuf;
    Buf m_Buf[NBUF];
    unsigned char Buffer[NBUF][BUFFER_SIZE];
    DeviceManager* m_DeviceManager;
};              // 设备管理模块的入口
```

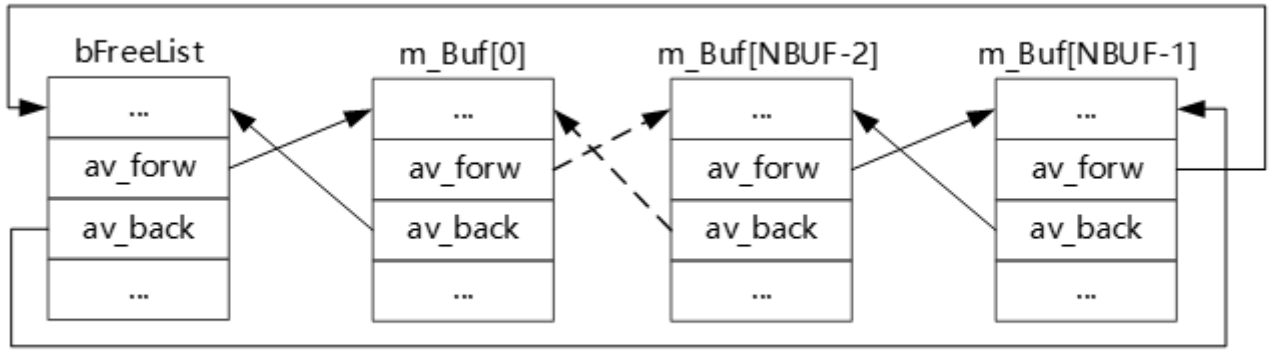
bFreeList自由缓存队列的队首，它是一个普通的缓存控制块。进程图像换入换出时，Swbuf登记IO操作细节，挂IO请求队列尾部；IO完成后，Swbuf置空闲。

SwBuf的细节，使用方法参考Swap( )函数

```
class Buf
{
    short       b_dev;    // 主硬盘
    int         b_blkno;  // 可交换部分，代码段在盘交换区上的起始扇区号
    unsigned char*  b_addr;  // ……内存首地址（物理地址）
    unsigned int    b_flags;  // 换入，B_READ；换出，B_WRITE
    int             b_error;
    int             b_resid;
    Buf*        b_forw;
    Buf*        b_back;
    Buf*        av_forw;   }
    Buf*        av_back;   }   挂 IO 请求队列
};
```

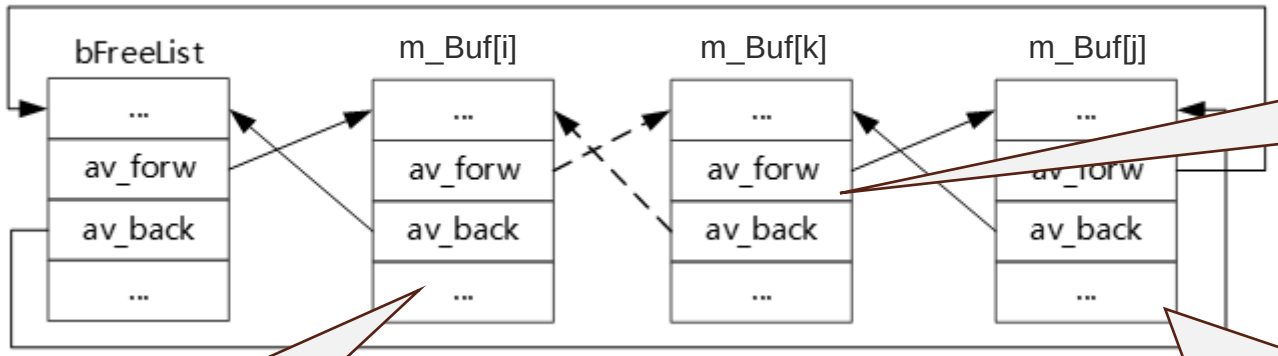
Unix V6（V6++）高速缓存管理 —— 自由缓存队列

管理 B_BUSY 为0的缓存块。



(b) 自由缓存队列初始状态

15个空白缓存块



命中的缓存，摘除。
（若其它进程正在使用，等待其使用完毕将缓存块送入队尾）

LRU缓存，分配给新数据块

使用完毕的缓存，插队尾

Unix V6（V6++）设备管理 —— 设备号 和 设备开关表

设备号　　short　b_dev;


```
class DeviceManager
{
        ……
        int nblkdev;        // 块设备的类型数
        BlockDevice         *bdevsw[MAX_DEVICE_NUM];        // 块设备开关表，每类块设备一项

        int nchrdev;        // 字符设备的类型数
        CharDevice          *cdevsw[MAX_DEVICE_NUM];    // 字符设备开关表，每类字符设备一项
}
```

```
class DeviceManager
{
    ……
      int nblkdev;          // 块设备的类型数
      BlockDevice           *bdevsw[MAX_DEVICE_NUM];    // 块设备开关表，每类块设备一项

      int nchrdev;          // 字符设备的类型数
      CharDevice            *cdevsw[MAX_DEVICE_NUM];    // 字符设备开关表，每类字符设备一项
}
```

设备号 short b_dev;

major	minor

```
class BlockDevice
{
public:  // 该型号磁盘的驱动程序
      virtual int Open(short dev, int mode);
      virtual int Close(short dev, int mode);
      virtual int Strategy(Buf* bp);  // 硬盘IO函数
      virtual void Start();

public:
      Devtab*  d_tab[n];  // 配置该型号磁盘 n 个。每张
                          磁盘一个块设备表Devtab，
                          登记磁盘的工作状态
};
```

```
class CharDevice
{
public:  // 该型号终端的驱动程序
      virtual void Open(short dev, int mode);
      virtual void Close(short dev, int mode);
      virtual void Read(short dev);   // 终端读函数
      virtual void Write(short dev);   // 终端写函数
      virtual void SgTTy(short dev, TTy* pTTy);

public:
      TTy* m_TTy[m];  // 配置该型号终端m台。每台终端一个tty
                      结构，登记该终端的工作状态，主要是
                      终端的输入缓存和输出缓存
};
```

Unix V6（V6++）配置的设备

```
class DeviceManager
{
   static const short ROOTDEV = (0 << 8) | 0; // 1张主硬盘（c.img，装有引导扇区和内核），主、从设备号都为0
   static const short TTYDEV = (0 << 8) | 0;     // 1个控制台（0#终端），主、从设备号都为0
            ……
}
```

```
class BlockDevice
{
public:  // 该型号磁盘的驱动程序
      virtual int Open(short dev, int mode);
      virtual int Close(short dev, int mode);
      virtual int Strategy(Buf* bp);  // 硬盘IO函数
      virtual void Start();

public:
      Devtab*  d_tab;  // 主硬盘的设备开关表
};
```

```
class CharDevice
{
public:  // 该型号终端的驱动程序
      virtual void Open(short dev, int mode);
      virtual void Close(short dev, int mode);
      virtual void Read(short dev);   // 终端读函数
      virtual void Write(short dev);   // 终端写函数
      virtual void SgTTy(short dev, TTy* pTTy);

public:
      TTy* m_TTy;   // 控制台的tty
};
```

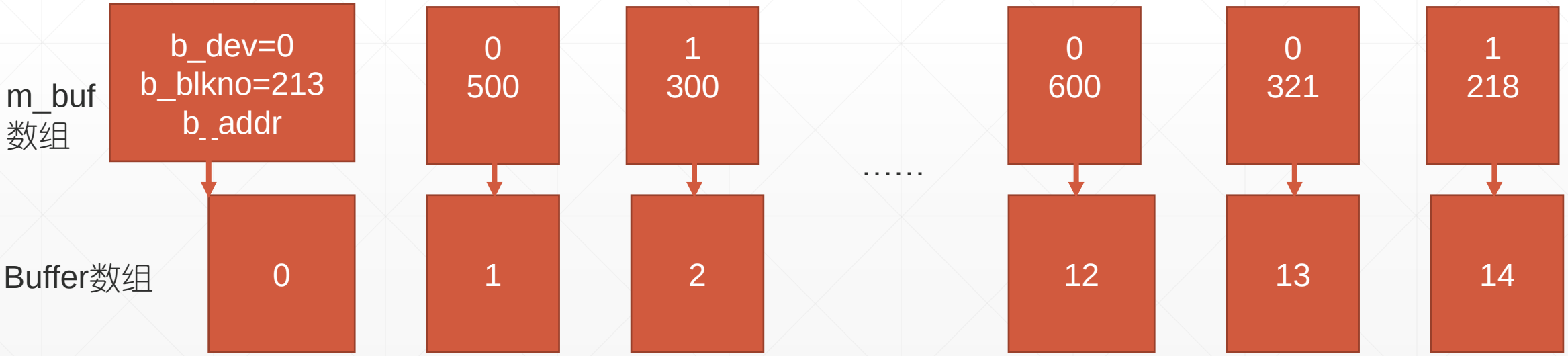
Unix V6（V6++）设备管理 —— 块设备表

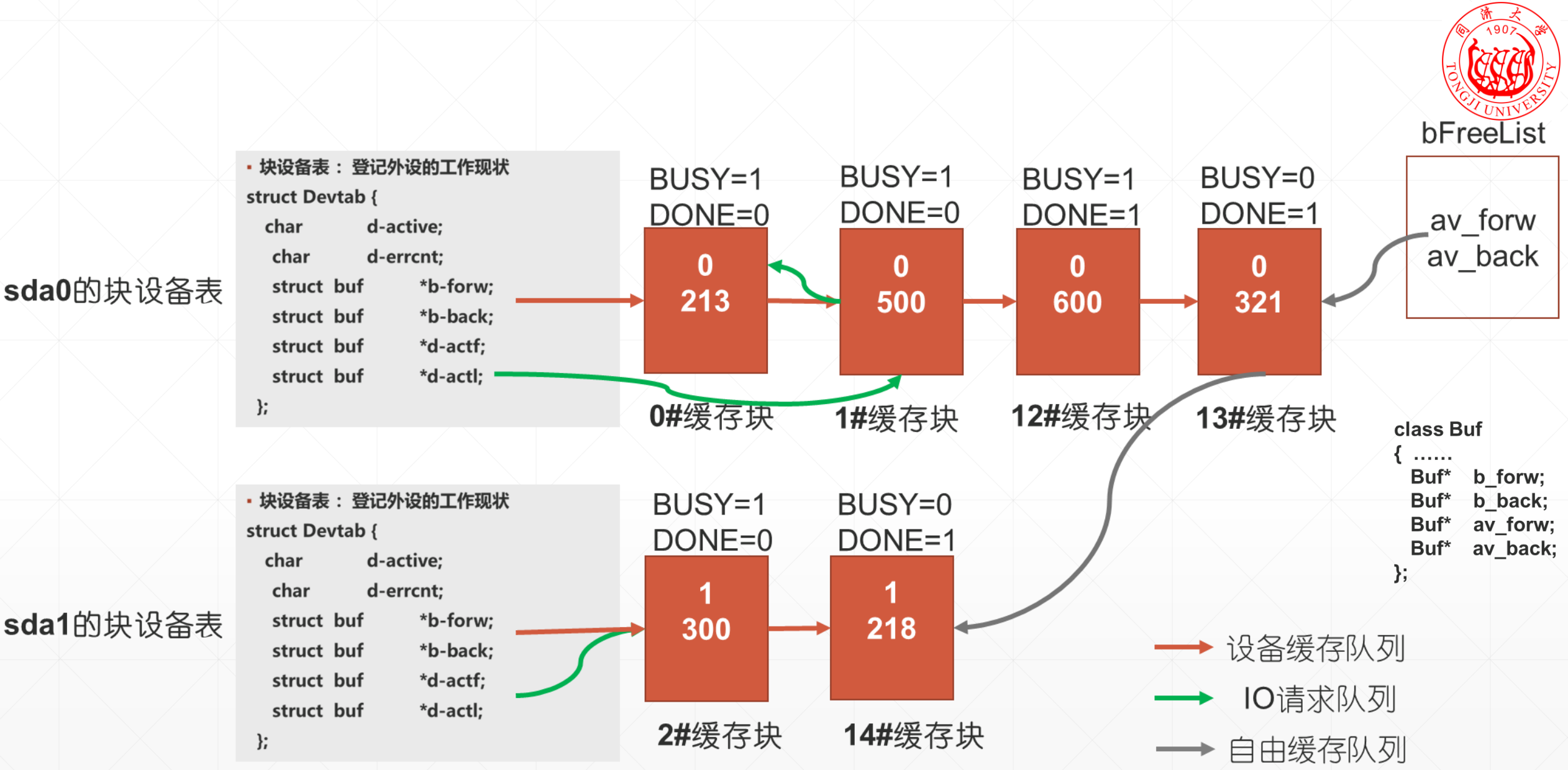
```
class Devtab
{
        int     d_active;   // 非0，硬盘忙
        int     d_errcnt;   //  当前IO操作出错次数
        Buf*    b_forw;     // 设备缓存队列，用来管理分配给该磁盘的所有缓存
        Buf*    b_back;
        Buf*    d_actf;     // IO请求队列，队首缓存IO执行中，其余缓存IO操作等待执行。
        Buf*    d_actl;
};
```

例题：

例：配有2个硬盘的UNIX V6++系统　sda0设备号0,0；　sda1设备号0,1

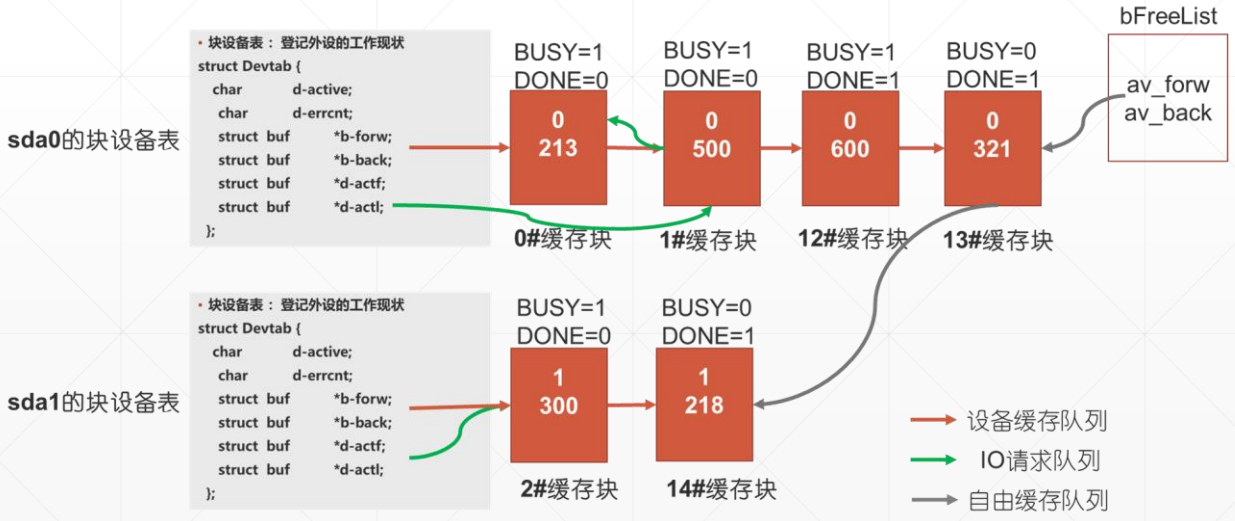
块设备开关表bdevsw，只有一个元素。d_tab数组，2个元素。

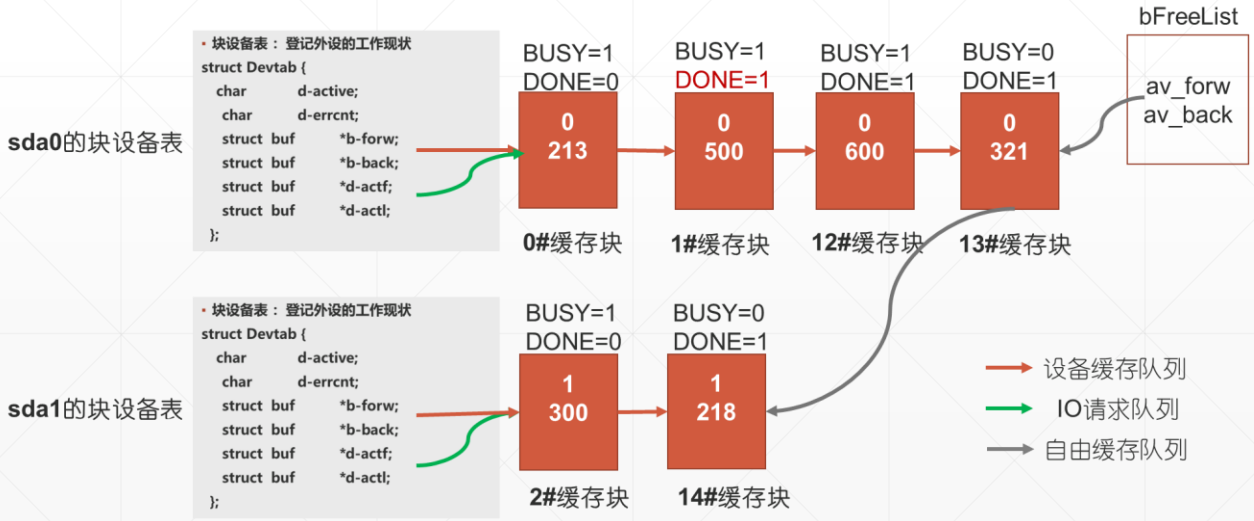



bFreeList
av_forw
av_back
sda0的块设备表
块设备表： 登记外设的工作现状
struct Devtab {
char d-active;
char d-errcnt;
struct buf *b-forw;
struct buf *b-back;
struct buf *d-actf;
struct buf *d-actl;
};
BUSY=1 DONE=0
0 213
BUSY=1 DONE=0
0 500
BUSY=1 DONE=1
0 600
BUSY=0 DONE=1
0 321
0#缓存块
1#缓存块
12#缓存块
13#缓存块
sda1的块设备表
块设备表： 登记外设的工作现状
struct Devtab {
char d-active;
char d-errcnt;
struct buf *b-forw;
struct buf *b-back;
struct buf *d-actf;
struct buf *d-actl;
};
BUSY=1 DONE=0
1 300
BUSY=0 DONE=1
1 218
2#缓存块
14#缓存块
class Buf
{ ……
Buf* b_forw;
Buf* b_back;
Buf* av_forw;
Buf* av_back;
};
设备缓存队列
IO请求队列
自由缓存队列

数据块 <0,500> ， IO完成

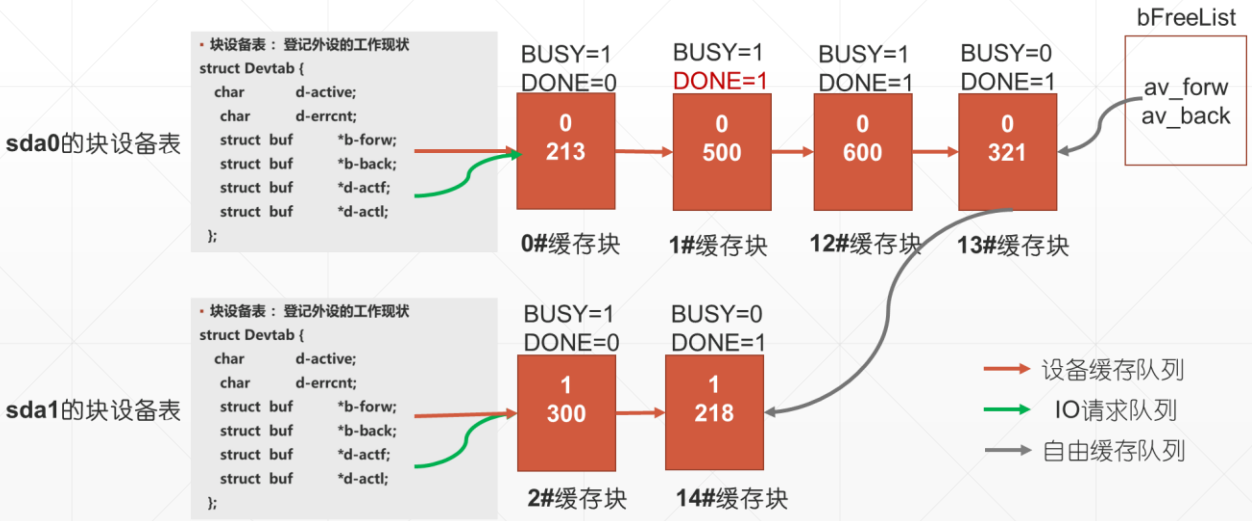
- 磁盘发中断请求
- 处理器响应中断，执行中断处理程序
- 中断处理程序
  - B_DONE = 1，WakeUpAll(&Buf[1])
  - 将Buf[1]从IO请求队列移除
  - 启动数据块<0,213>的IO操作

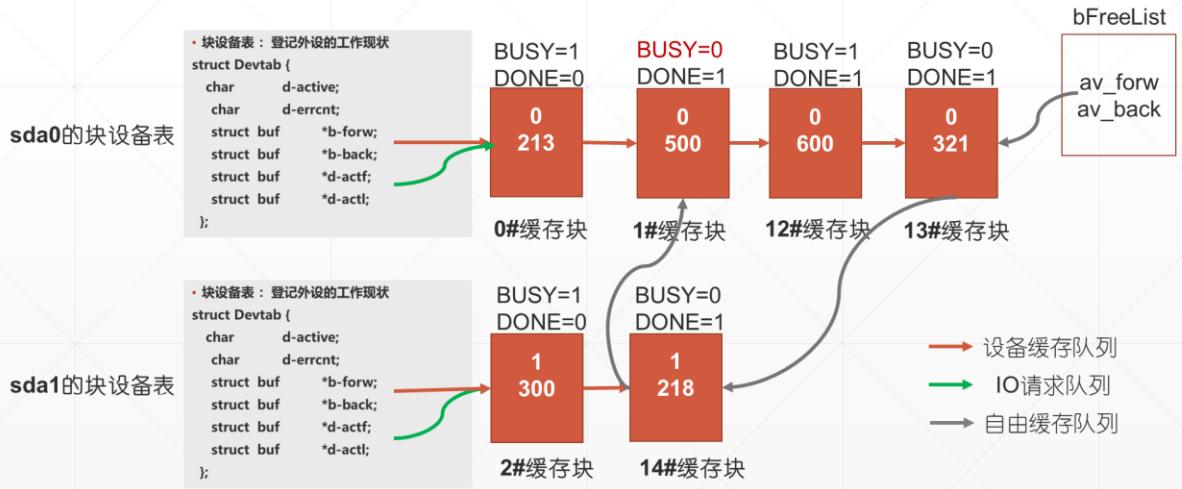




数据块 <0,500> 使用完毕

- B_BUSY=0 解锁
- 插自由缓存队列尾部





一、同步读

- 使用场合：进程需要阻塞等待读操作结束，之后处理同步读入的磁盘数据。
- 文件系统默认的读操作。

1、将磁盘数据块<dev，blkno>读入缓存池，锁住分配给它的缓存控制块，返回其首地址。

```
bp = Bread(dev,blkno);
```

2、将缓存块中磁盘数据，复制到现运行进程的用户空间 或 文件系统使用的某个内核变量。

```
IOMove(src, dst, nbytes)
```

// src是磁盘数据在缓存块bp中的首地址，dst是用来装磁盘数据的数据结构首地址，nbytes是磁盘数据的尺寸。

3、释放缓存块。

```
Brelse(bp);
```

Bread(short dev, int blkno)

```
{
        Buf* bp;

        bp = this->GetBlk(dev, blkno);    // 锁住分配给blkno的缓存块

        if(bp->b_flags & Buf::B_DONE)   // 如果缓存命中，B_DONE会是1
        {
                return bp;    // 返回，无需读磁盘
        }

        bp->b_flags |= Buf::B_READ;    // 构造IO请求块 ： 读操作
        bp->b_wcount = BufferManager::BUFFER_SIZE;    // 构造IO请求块 ： 读512字节

        this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp);  // 送IO请求队列尾部

        this->IOWait(bp);  // 睡眠等待读操作完成
        return bp;
}
```

GetBlk(dev, blkno)，锁住分配给<dev,blkno>的缓存块，获得使用权

- loop:

```
1、major = Utility::GetMajor(dev);
   dp = this->m_DeviceManager->GetBlockDevice(major).d_tab;  // dp是磁盘dev设备缓存队列的队首
2、for(bp = dp->b_forw; bp != (Buf *)dp; bp = bp->b_forw)     // 遍历设备缓存队列
   {
      if(bp->b_blkno != blkno || bp->b_dev != dev)  // 找数据块<dev,blkno>
            continue;
      // 命中。。。。。。数据可重用，不必IO磁盘
      if(bp->b_flags & Buf::B_BUSY)  // 数据块，其它进程正在使用
      {
          bp->b_flags |= Buf::B_WANTED;    // 置等待标识
          u.u_procp->Sleep((unsigned long)bp, -50);    // 入睡，等待缓存块解锁B_BUSY变0
          goto loop;
      }
      this->NotAvail(bp);          return bp;
   }
```

```
Brelse(Buf* bp)  //持锁进程用完数据块，会解锁缓存
{
    ......
    if(bp->b_flags & Buf::B_WANTED)
        procMgr.WakeUpAll((unsigned long)bp);
    ......
    bp->b_flags &= ~(Buf::B_WANTED | Buf::B_BUSY | ......);
    ......
}
```

```
void BufferManager::NotAvail(Buf *bp)
{
        X86Assembly::CLI();
        /* 从自由队列中取出 */
        bp->av_back->av_forw = bp->av_forw;
        bp->av_forw->av_back = bp->av_back;
        /* 设置B_BUSY标志 */
        bp->b_flags |= Buf::B_BUSY;
        X86Assembly::STI();
        return;
}
```

B_BUSY是1，
B_DONE是1

// 缓存不命中，为<dev，blkno>分配缓存块：自由缓存队列中第一个不脏的缓存块

3、if(this->bFreeList.av_forw == &this->bFreeList) // 自由缓存队列为空，入睡等待自由缓存

{

this->bFreeList.b_flags |= Buf::B_WANTED;

u.u_procp->Sleep((unsigned long)&this->bFreeList, -50);

X86Assembly::STI();

goto loop;

}

4、bp = this->bFreeList.av_forw;

this->NotAvail(bp); //自由缓存队列队首缓存出列

5、if(bp->b_flags & Buf::B_DELWRI) // 脏缓存

{ // 异步写回磁盘

bp->b_flags |= Buf::B_ASYNC;

this->Bwrite(bp);

goto loop; // 不必等待脏缓存写操作结束，为<dev,blkno>分配第一个不脏的自由缓存

}

```
void BufferManager::Brelse(Buf* bp)
{
        ……
   if(this->bFreeList.b_flags & Buf::B_WANTED)
   {
      this->bFreeList.b_flags &= (~Buf::B_WANTED);
      procMgr.WakeUpAll((unsigned long)&this->bFreeList);
   }
        ……
}
```

6、bp->b_flags = Buf::B_BUSY; //上锁，清空其它标识

从原设备队列中抽出该缓存块

bp->b_dev = dev;

bp->b_blkno = blkno;

} 登记新的<dev，blkno>

插入新的设备缓存队列，队首

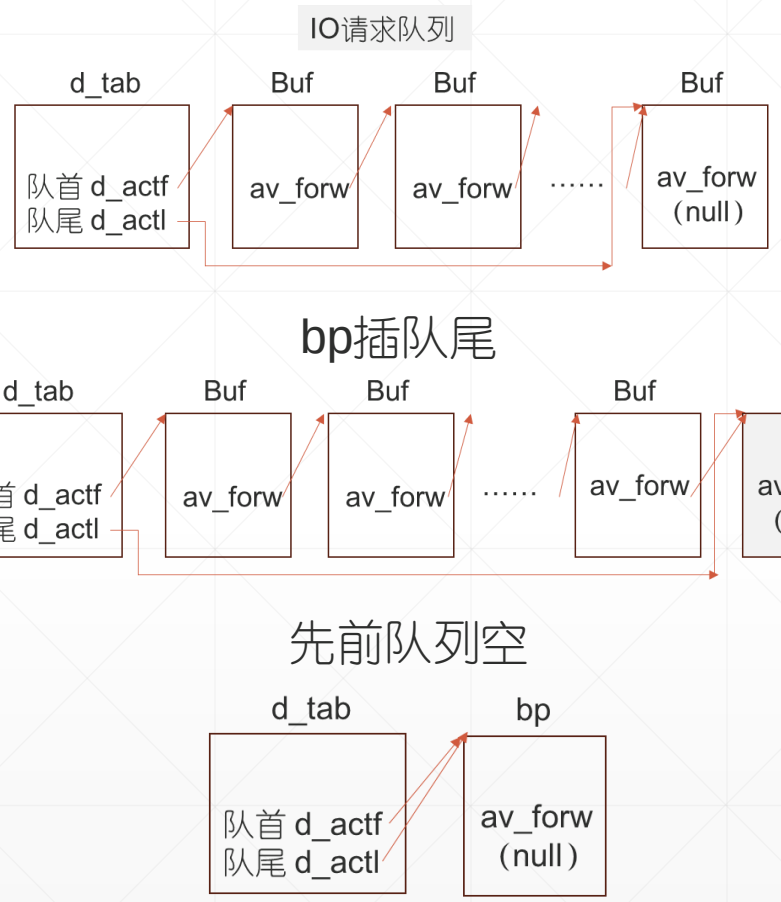
7、return bp; //返回一个上锁的空缓存块

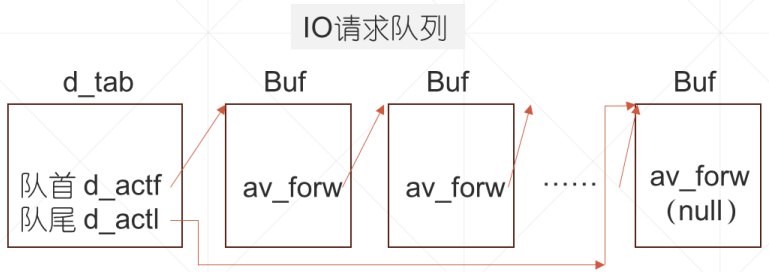
B_BUSY是
B_DONE是0

```
void BufferManager::IOWait(Buf* bp)
{
        User& u = Kernel::Instance().GetUser();

        X86Assembly::CLI();
        while( (bp->b_flags & Buf::B_DONE) == 0 )   // 睡眠等待IO完成B_DONE变1
        {
                u.u_procp->Sleep((unsigned long)bp, ProcessManager::PRIBIO);
        }
        X86Assembly::STI();

        this->GetError(bp);
        return;
}
```

二、磁盘驱动

```
int ATABlockDevice::Strategy(Buf* bp)    // bp，IO请求块
{
        ……
        // 1、bp送I/O请求队列队尾
        bp->av_forw = NULL;
        if(this->d_tab->d_actf == NULL)
                this->d_tab->d_actf = bp;
        else
                this->d_tab->d_actl->av_forw = bp;
        this->d_tab->d_actl = bp;

        // 2、如果磁盘空闲
        if(this->d_tab->d_active == 0)
                this->Start();   // 启动IO操作
        return 0;
}
```

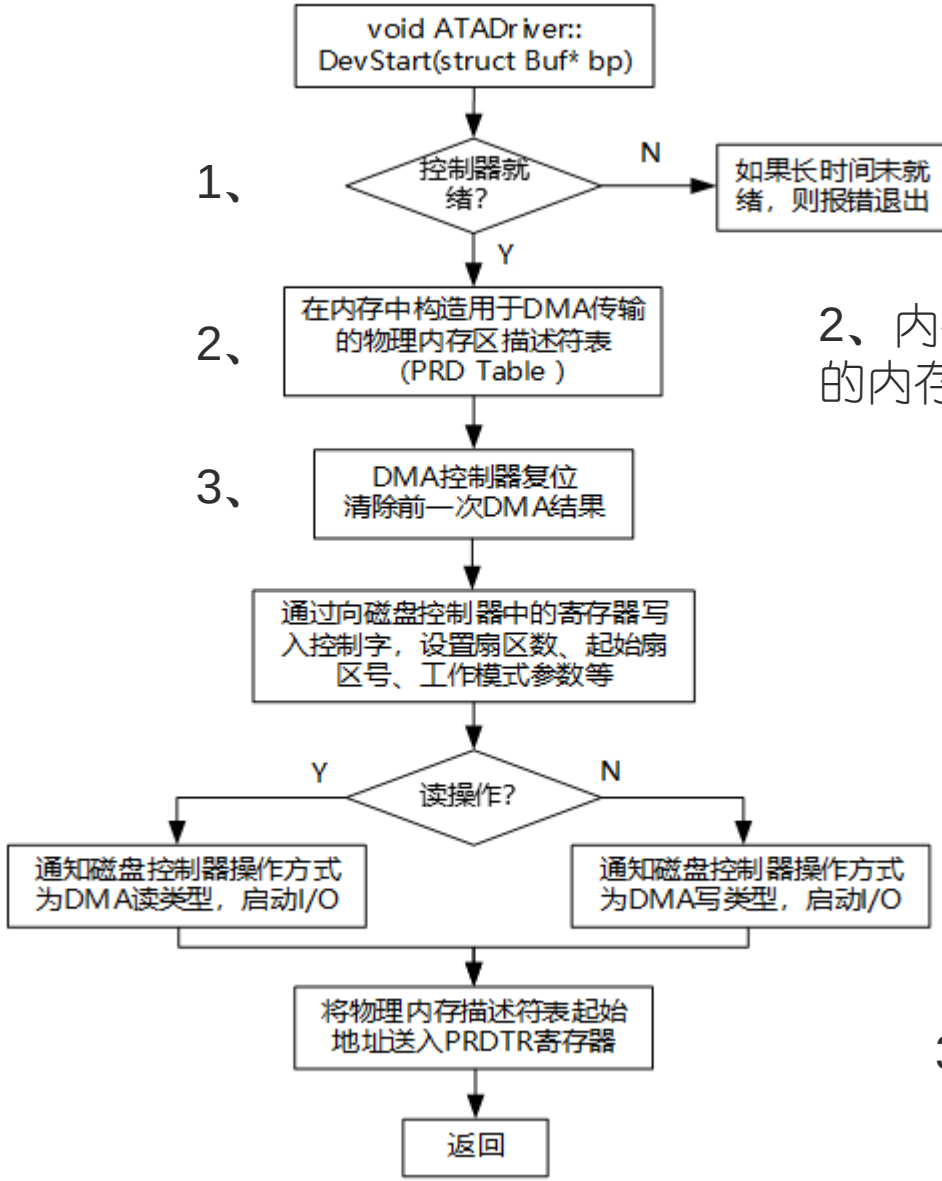





```
void ATABlockDevice::Start()
{
        Buf* bp;

        if( (bp = this->d_tab->d_actf) == 0 )      // I/O请求队列空
                return;      // 完成最后一个IO请求后，中断处理程序无需启动下个IO操作，走这个分支返回

        this->d_tab->d_active++;    // 置磁盘忙

        // 启动队首数据块的I/O操作
        ATADriver::DevStart(bp);
}
```

磁盘IO，使用DMA控制方式。

所以DevStart要编程2块芯片：磁盘控制器 和 DMA控制器



1、检查磁盘控制器工作状态

2、内存中为DMA数据传输准备 PRD表。DMA数据传输可以覆盖多个不连续的内存区域，PRD表登记每块内存区域的起始地址和数据传输量。

0		第1块内存区域的数据传输量	0
第1块内存区域的起始物理地址			0
1		第2块内存区域的数据传输量	0
第2块内存区域的起始物理地址			0

Unix V6++使用的PRD表只有1项

1		512 / p_size / x_size	0
Buffer[i] / p_addr / x_cadddr			0

bp->b_wcount

bp->b_addr & ~0xC0000000

3、复位DMA控制器中的所有寄存器

```
class PRDTable     // 内存区域描述符表
{
        static const int NSIZE = 10;      /* 一次DMA操作，最多读写10个不连续的内存区域 */
        PhysicalRegionDescriptor m_Descriptors[NSIZE] ……;
};
```

```
class PhysicalRegionDescriptor
{    // 内存区域描述符
     unsigned long m_BaseAddressZeroBit : 1;              // 0，内存区域“物理起始地址”的第0位
     unsigned long m_MemoryRegionPhysicalBaseAddress : 31;     // 内存区域 "物理起始地址"的第[31 : 1]位
     unsigned short m_ByteCountZeroBit : 1;               // 0，DMA传输字节数的第0位
     unsigned short m_ByteCount : 15;                                  //  DMA传输字节数的第[15 : 1]位
     unsigned short m_Reserved : 15;                      // 保留区域
     unsigned short m_EOT : 1;  // End of Table位。该位为1表示当前物理内存区描述符(PRD)是PRD表中的最后一项。
}__attribute__((packed));
```

4、编程磁盘控制器，设置起始扇区号 bp->b_blkno 和 需要传输的数据量 bp->b_wcount

```
85      /* 设置扇区数 */
86      IOPort::OutByte(ATADriver::NSECTOR_PORT, bp->b_wcount / BufferManager::BUFFER_SIZE);
87      /* 设置LBA28寻址模式中磁盘块号的0-7位 */
88      IOPort::OutByte(ATADriver::BLKNO_PORT_1, bp->b_blkno & 0xFF);
89      /* 设置LBA28寻址模式中磁盘块号的8-15位 */
90      IOPort::OutByte(ATADriver::BLKNO_PORT_2, (bp->b_blkno >> 8) & 0xFF);
91      /* 设置LBA28寻址模式中磁盘块号的16-23位 */
92      IOPort::OutByte(ATADriver::BLKNO_PORT_3, (bp->b_blkno >> 16) & 0xFF);
93      /* 设置ATA磁盘工作模式寄存器，以及LBA28中的24-27位 */
94      IOPort::OutByte(ATADriver::MODE_PORT, ATADriver::MODE_IDE | ATADriver::MODE_LBA28 | (minor << 4) | ((bp->b_blkno >> 24) & 0x0F) );
95
96      /* 如果是读操作 */
97      if( (bp->b_flags & Buf::B_READ) == Buf::B_READ )
98      {
99          /* 告诉磁盘控制器的读、写类型，启动I/O */
100         IOPort::OutByte(ATADriver::CMD_PORT, ATADriver::HD_DMA_READ);
101
102         /* 告诉DMA控制器的读、写类型，传入PRD Table的物理起始地址，启动一次DMA */
103         DMA::Start(DMA::READ, table.GetPRDTableBaseAddress());
104     }
105     else    /* 如果是写操作 */
106     {
107         IOPort::OutByte(ATADriver::CMD_PORT, ATADriver::HD_DMA_WRITE);
108
109         DMA::Start(DMA::WRITE, table.GetPRDTableBaseAddress());
110     }
111     return;
112 }
```

5、写磁盘控制器的命令端口，发读命令，启动磁盘

6、编程DMA控制器，设置PRDR寄存器引用内存中的PRT表，写命令端口发读命令，启动DMA操作

三、中断处理程序

```
void ATADriver::ATAHandler(struct pt_regs *reg, struct pt_context *context)
{       ……
        short major = Utility::GetMajor(DeviceManager::ROOTDEV);
        BlockDevice& bdev = Kernel::Instance().GetDeviceManager().GetBlockDevice(major);
        atab = bdev.d_tab;   // atab块设备表

        if( atab->d_active == 0 )    //  怎么会？没明白
                return;

        bp = atab->d_actf;    // bp，IO请求队列队首。IO操作完成。
        atab->d_active = 0;   // 置磁盘空闲状态
```

```
    if( ATADriver::IsError() || DMA::IsError() )   // IO故障
    {
            if(++atab->d_errcnt <= 10) // 磁盘操作错误计数器
            {
                    bdev.Start();   // 重启队首IO操作
                    return;
            }
            bp->b_flags |= Buf::B_ERROR;    // 10次还不成功，扇区坏了放弃这个IO操作，Buf置出错标记
    }

    atab->d_errcnt = 0;                     /* 磁盘操作错误计数器归零 */
    atab->d_actf = bp->av_forw;             /* bp，I/O请求队列出列 */
    Kernel::Instance().GetBufferManager().IODone(bp);    /* 唤醒等待IO操作结束的进程 */
    bdev.Start();       /* 启动I/O请求队列中下一个I/O请求 */
    /* 对主、从8259A中断控制芯片分别发送EOI命令。 */
    IOPort::OutByte(Chip8259A::MASTER_IO_PORT_1, Chip8259A::EOI);
    IOPort::OutByte(Chip8259A::SLAVE_IO_PORT_1, Chip8259A::EOI);
    return;
}
```

磁盘IO出错可能有多种原因（1）扇区坏或者其它硬件故障，这种错误无法恢复（2）盘面静电或其它原因导致的故障，重新执行IO命令后，故障可以排除

```
void BufferManager::IODone(Buf* bp)
{
	/* 置上I/O完成标志 */
	bp->b_flags |= Buf::B_DONE;
	if(bp->b_flags & Buf::B_ASYNC)
	{
		/* 如果是异步操作,立即释放缓存块，解锁、放自由缓存队列尾部 */
		this->Brelse(bp);
	}
	else
	{
		/* 清除B_WANTED标志位 */
		bp->b_flags &= (~Buf::B_WANTED);
		Kernel::Instance().GetProcessManager().WakeUpAll((unsigned long)bp);
	}
	return;
}
```

四、被磁盘中断处理程序唤醒的进程 1

一、同步读

- 使用场合：进程需要阻塞等待读操作结束，之后需要处理同步读入的磁盘数据。
- 文件系统默认的读操作。

1、将磁盘数据块<dev，blkno>读入缓存池，锁住分配给它的缓存控制块，返回其首地址。

```
bp = Bread(dev,blkno);
```

2、将缓存块中磁盘数据，复制到现运某个内核变量。

```
IOMove(src, dst, nbytes)
// src是磁盘数据在缓存块中的首地址，dst是用来
```

3、释放缓存块。

```
Brelse(pBuf);
```

```
Bread(short dev, int b
{
    Buf* bp;

    bp = this->GetBlk(dev, blkno);    // 

    if(bp->b_flags & Buf::B_DONE)    //
    {
        return bp;    // 返回，无需
    }

    bp->b_flags |= Buf::B_READ;    // 构造IO请求块 ： 读操作
    bp->b_wcount = BufferManager::BUFFER_SIZE;    // 构造IO请求块 ： 读512字节

    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp);  // 送IO请求队列尾部

    this->IOWait(bp);  // 睡眠等待读操作完成
    return bp;
}
```

```
void BufferManager::IOWait(Buf* bp)
{
    User& u = Kernel::Instance().GetUser();

    X86Assembly::CLI();
    while( (bp->b_flags & Buf::B_DONE) == 0 )  // 睡眠等待，IO完成，B_DONE变1
    {
        u.u_procp->Sleep((unsigned long)bp, ProcessManager::PRIBIO);
    }
    X86Assembly::STI();

    this->GetError(bp);
    return;
}
```

启动IO操作的进程，完成磁盘数据读取操作，返回

被磁盘中断处理程序唤醒的进程 2

一、同步读

- 使用场合：进程需要阻塞等待读操作结束，之后需要处理同步读入的磁盘数据。
- 文件系统默认的读操作。

1、将磁盘数据块<dev，blkno>读入缓存池，锁住分配给它的缓存控制块，返回其首地址。

bp = Bread(dev,blkno);

2、将缓存块中磁盘数据，复制到现运某个内核变量。

IOMove(src, dst, nbytes)

// src是磁盘数据在缓存块中的首地址，dst是用来

3、释放缓存块。

Brelse(pBuf);

等待复用数据块的进程，缓存块未解锁（B_BUSY是0），置B_WANTED，再次入睡

操作系统 电信学院计算机系 邓蓉

```
Bread(short dev, int bl
{
    Buf* bp;

    bp = this->GetBlk(dev, blkno);    // 锁

    if(bp->b_flags & Buf::B_DONE)    // 
    {
        return bp;    // 返回，无需
    }

    bp->b_flags |= Buf::B_READ;    // 构造IO请求块 ： 读操作
    bp->b_wcount = BufferManager::BUFFER_SIZE;    // 构造IO请求块 ： 读512字节

    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp);  // 送IO请求队列尾部

    this->IOWait(bp);  // 睡眠等待读操作完成
    return bp;
}
```

操作系统 电信学院计算机系 邓蓉 19

GetBlk(dev, blkno)，锁住分配给<dev,blkno>的缓存块，获得使用权

- loop:

```
1、major = Utility::GetMajor(dev);
   dp = this->m_DeviceManager->GetBlockDevice(major).d_tab;  // dp是磁盘dev设备缓存队列的队首
2、for(bp = dp->b_forw; bp != (Buf *)dp; bp = bp->b_forw)    // 遍历设备缓存队列
   {
      if(bp->b_blkno != blkno || bp->b_dev != dev)  // 找数据块<dev,blkno>
          continue;
      // 命中。。。。。。数据可重用，不必IO磁盘
      if(bp->b_flags & Buf::B_BUSY)  // 数据块，其它进程正在使用
      {
          bp->b_flags |= Buf::B_WANTED;    // 置等待标识
          u.u_procp->Sleep((unsigned long)bp, -50);    // 入睡，等待缓存块解锁
          goto loop;
      }
      this->NotAvail(bp);        return bp;
   }
```

```
Brelse(Buf* bp)  //持锁进程用完数据块，会解锁缓存
{
    ......
    if(bp->b_flags & Buf::B_WANTED)
        procMgr.WakeUpAll((unsigned long)bp);
    ......
    bp->b_flags &= ~(Buf::B_WANTED | Buf::B_BUSY | ......)
    ......
}
```

```
void BufferManager::NotAvail(Buf *bp)
{
    X86Assembly::CLI();
    /* 从自由队列中取出*/
    bp->av_back->av_forw = bp->av_forw;
    bp->av_forw->av_back = bp->av_back;
    /* 设置B_BUSY标志 */
    bp->b_flags |= Buf::B_BUSY;
    X86Assembly::STI();
    return;
}
```

// 获得缓存块的使用权，上锁，将其从自由缓存队列移除。返回的缓存块，B_BUSY是1，B_DONE是1

操作系统 电信学院计算机系 邓蓉 21

启动IO操作的进程，释放缓存块时，唤醒等待复用数据块的进程。
如果有多个进程等待复用同一个数据块，依次执行系统调用下半部，读写缓存块中的数据。
读写操作执行顺序随机。

四、被磁盘中断处理程序唤醒的进程 1

一、同步读

- 使用场合：进程需要阻塞等待读操作结束，之后需要处理同步读入的磁盘数据。
- 文件系统默认的读操作。

1、将磁盘数据块<dev，blkno>读入缓存池，锁住分配给它的缓存控制块，返回其首地址。

```
bp = Bread(dev,blkno);
```

2、将缓存块中磁盘数据，复制到现运某个内核变量。

```
IOMove(src, dst, nbytes)
// src是磁盘数据在缓存块中的首地址，dst是用来
```

3、释放缓存块。

```
Brelse(pBuf);
```

```
Bread(short dev, int b
{
    Buf* bp;

    bp = this->GetBlk(dev, blkno);   // 
    {
        return bp;   // 返回，无需
    }

    bp->b_flags |= Buf::B_READ;   // 构造IO请求块：读操作
    bp->b_wcount = BufferManager::BUFFER_SIZE;   // 构造IO请求块：读512字节

    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp); // 送IO请求队列尾部

    this->IOWait(bp); // 睡眠等待读操作完成
    return bp;
}
```

```
void BufferManager::IOWait(Buf* bp)
{
    User& u = Kernel::Instance().GetUser();

    X86Assembly::CLI();
    while( (bp->b_flags & Buf::B_DONE) == 0 )   // 睡眠等待，IO完成，B_DONE变1
    {
        u.u_procp->Sleep((unsigned long)bp, ProcessManager::PRIBIO);   ★
    }
    X86Assembly::STI();

    this->GetError(bp);
    return;
}
```

启动IO操作的进程，完成磁盘数据读取操作，返回

被磁盘中断处理程序唤醒的进程 2

一、同步读

- 使用场合：进程需要阻塞等待读操作结束，之后需要处理同步读入的磁盘数据。
- 文件系统默认的读操作。

1、将磁盘数据块<dev，blkno>读入缓存池，锁住分配给它的缓存控制块，返回其首地址。

```
bp = Bread(dev,blkno);
```

2、将缓存块中磁盘数据，复制到现运某个内核变量。

```
IOMove(src, dst, nbytes)
// src是磁盘数据在缓存块中的首地址，dst是用来
```

3、释放缓存块。

```
Brelse(pBuf);
```

```
Bread(short dev, int bl
{
    Buf* bp;

    bp = this->GetBlk(dev, blkno);   // 锁
    if(bp->b_flags & Buf::B_DONE)   // 
    {
        return bp;   // 返回，无需
    }

    bp->b_flags |= Buf::B_READ;   // 构造IO请求块：读操作
    bp->b_wcount = BufferManager::BUFFER_SIZE;   // 构造IO请求块：读512字节

    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(dev)).Strategy(bp); // 送IO请求队列尾部

    this->IOWait(bp); // 睡眠等待读操作完成
    return bp;
}
```

```
GetBlk(dev, blkno)，锁住分配给<dev,blkno>的缓存块，获得使用权
loop:
1、major = Utility::GetMajor(dev);
   dp = this->m_DeviceManager->GetBlockDevice(major).d_tab; // dp是磁盘dev设备缓存队列的队首
2、for(bp = dp->b_forw; bp != (Buf *)dp; bp = bp->b_forw)   // 遍历设备缓存队列
   {
       if(bp->b_blkno != blkno || bp->b_dev != dev) // 找数据块<dev,blkno>
           continue;
       // 命中。。。。。数据可重用，不必IO磁盘
       if(bp->b_flags & Buf::B_BUSY) // 数据块，其它进程正在使用
       {
           bp->b_flags |= Buf::B_WANTED;   // 置等待标识
           u.u_procp->Sleep((unsigned long)bp, -50);   // 入睡，等待缓存块解锁
           goto loop;
       }
       this->NotAvail(bp);       return bp;
   }
```

```
void BufferManager::NotAvail(Buf *bp)
{
    X86Assembly::CLI();
    /* 从自由队列中取出*/
    bp->av_back->av_forw = bp->av_forw;
    bp->av_forw->av_back = bp->av_back;
    /* 设置B_BUSY标志*/
    bp->b_flags |= Buf::B_BUSY;
    X86Assembly::STI();
    return;
}
```

// 获得缓存块的使用权，上锁，将其从自由缓存队列移除。返回的缓存块，B_BUSY是1，B_DONE是1

等待复用数据块的进程，缓存块未解锁（B_BUSY是0），置B_WANTED，再次入睡

五、习题

- 已知，磁盘访问一次耗时T，访问缓存块中的数据每次耗时t。某时段，数据块X命中n次。问，访问X，平均耗时？

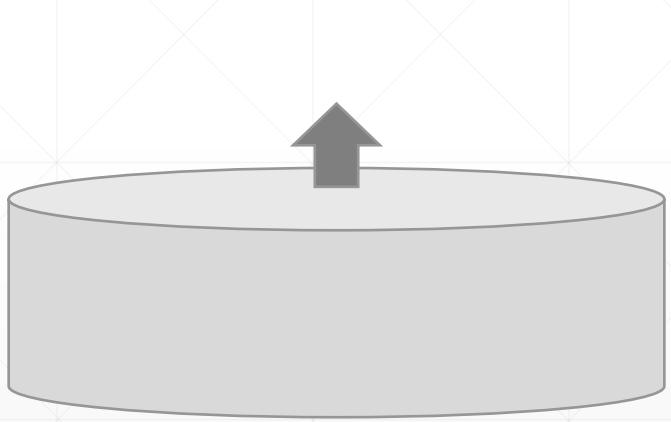
答：IO一次，将数据块装入缓存块，耗时T

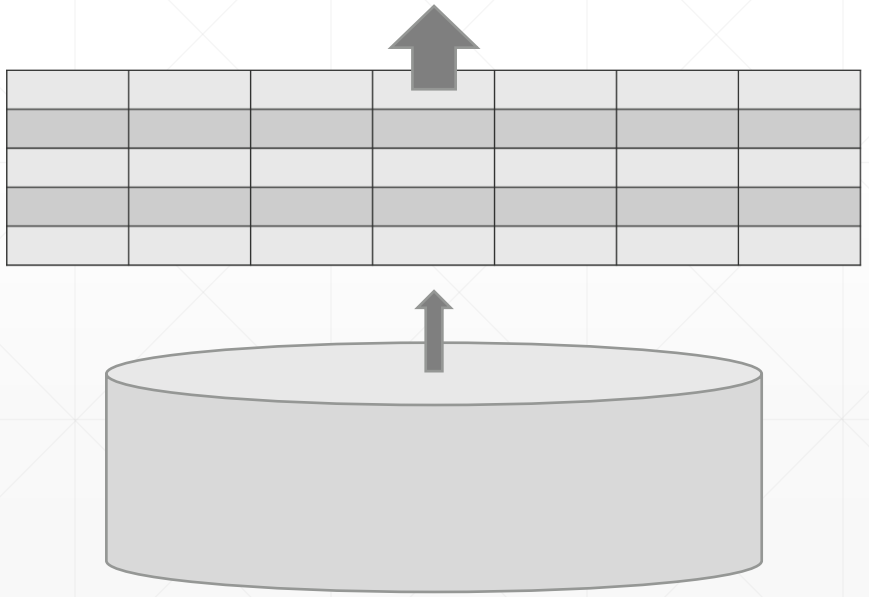
访问数据块 n+1 次，取用缓存块中的数据，耗时(n+1)*t

平均耗时 t + T/(n+1)。减少 n 次磁盘IO操作。

对于读操作，缓存的价值：
数据重用，减小读操作的平均耗时，提高磁盘吞吐率

- 若数据块有重用的可能，磁盘高速缓存池可以
  - 平摊耗时的磁盘IO操作，有效缩短读操作的平均耗时
  - 减少磁盘IO操作的总数，缩短IO请求队列长度，减小IO操作的平均耗时 T，进一步缩短读操作的平均耗时





引入缓存的代价

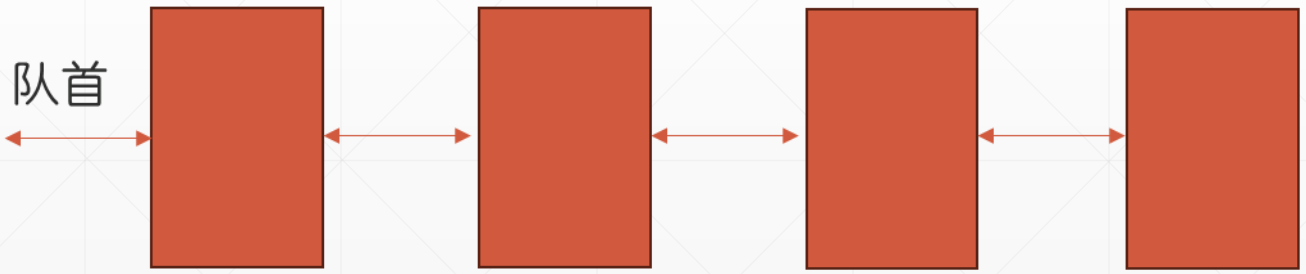
更一般地，借助缓存，访问一个磁盘数据块的平均耗时 t + T/(n+1)，n是该数据块缓存命中次数。

原本，数据直接进用户空间，耗时 T。
现在数据先进内核缓存块，再进用户空间。若数据块没有重用可能，耗时是增加的。

如果我们需要遍历千兆以上的大数据集，每块数据仅使用一次。
关掉磁盘高速缓存管理模块，有利于提升系统性能。例：GFS。

思考题：用什么操作可以废掉LRU队列？
请想办法 保护 LRU 队列。

LRU队列，管理自由缓存块（整个系统只有一个）



释放的缓存块，解锁、放队尾
命中的缓存块，上锁，从队列中删除

队首，**LRU**缓存块。上锁，摘除，分配给新数据块