操作系统　第五章　外设管理

5.3　块设备读写技术

同济大学计算机系

- **基本的块设备读写技术**
  - **同步读（已完成）**
  - **先读后写**
  - **异步写**
- **优化的块设备读写技术**
  - **预读**
  - **延迟写**

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。

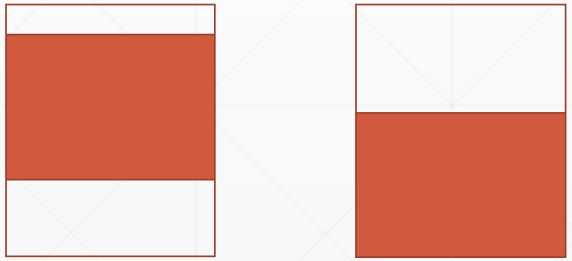
3、释放缓存块。

```
Brelse(bp);
```

数据写入缓存

写满（全部新数据）
不必先读

缓存块未写满，先读后写
（读入磁盘数据，保护不被改写的部分）

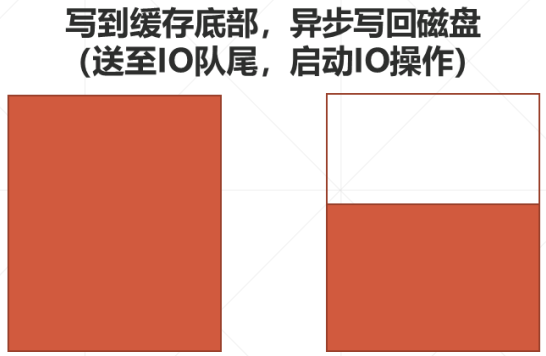


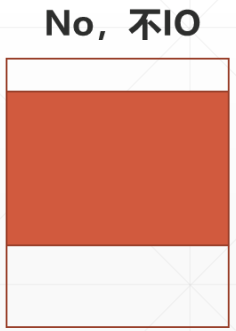
注意：先读后写

```
// 写数据块< dev, bn >。nbytes是需要写入的数据量。
if( Inode::BLOCK_SIZE == nbytes ) // 缓存块的尺寸，512字节
{
        pBuf = bufMgr.GetBlk(dev, bn); // 为数据块< dev, bn >分配缓存块
}
else
{
        pBuf = bufMgr.Bread(dev, bn); // 先读，保护数据块不被改写的部分
}
```

unsigned char* src = 写盘数据在用户区的首地址 或 内存元数据首地址
unsigned char* dst = 缓存块写入位置，首地址
Utility::IOMove(src, dst, nbytes); // 新数据写入缓存

缓存数据写回磁盘





缓存块中的数据与磁盘数据块不一致，是**脏缓存**（延迟写缓存）

```
// offset = dst+nbytes，写操作结束的位置

if( (offset % Inode::BLOCK_SIZE) == 0 ) // 写满啦
{
        bufMgr.Bawrite(pBuf); // 异步写回磁盘
}
else // 没有
{
        bufMgr.Bdwrite(pBuf); // 不启动IO，写操作延迟
}
```

```
void BufferManager::Bawrite(Buf *bp)
{
        /* 标记为异步写 */
        bp->b_flags |= Buf::B_ASYNC;
        this->Bwrite(bp);  // 执行写IO操作
        return;
}

void BufferManager::Bdwrite(Buf *bp)
{
    // 置脏标识 B_DELWRI
    // 置 B_DONE，允许其它进程复用数据块
    bp->b_flags |= (Buf::B_DELWRI | Buf::B_DONE);
    // 缓存块解锁
    this->Brelse(bp);
    return;
}
```

脏数据块，延迟写盘

- 写盘时机：（1）LRU的脏缓存块分配给其它数据块（2）写至缓存底部（3）磁盘卸载 unmount（4）关机。
- 写盘操作：（1，2，3）Bawrite( ) 异步写回磁盘（4） Bwrite( ) 同步写回磁盘。

延迟写技术

```
void BufferManager::Bwrite(Buf *bp)  // 写 IO 操作
{
    unsigned int flags = bp->b_flags;

    // 构造写IO请求块
    bp->b_flags &= ~(Buf::B_READ | Buf::B_DONE | Buf::B_ERROR | Buf::B_DELWRI);
    bp->b_wcount = BufferManager::BUFFER_SIZE;                    /* 512字节 */

    // 送IO队列尾
    this->m_DeviceManager->GetBlockDevice(Utility::GetMajor(bp->b_dev)).Strategy(bp);

    if( (flags & Buf::B_ASYNC) == 0 )   // 同步写
    {
            this->IOWait(bp);     // 等待。I/O结束后进程才能返回
            this->Brelse(bp);     // 解锁缓存
    }
    else if( (flags & Buf::B_DELWRI) == 0)  // 如果异步写 & 写操作是现运行进程执行的
    {
            this->GetError(bp);   // 看下IO有出错吗？出错，置IO 出错标记
    }
    return;    // 异步写。。。中断处理程序解锁缓存
}
```

延迟写技术的优缺点

- 优点

（1）写操作不用等IO完成，耗时降低很多（2）合并写操作，减少写IO操作的数量

- 缺点

（1）牺牲文件系统一致性，会丢失文件系统更新

- 克服缺点的方法

（1）定期将脏缓存刷回磁盘，Unix V6，30s。

操作系统　第五章　外设管理

5.4　磁盘性能优化技术

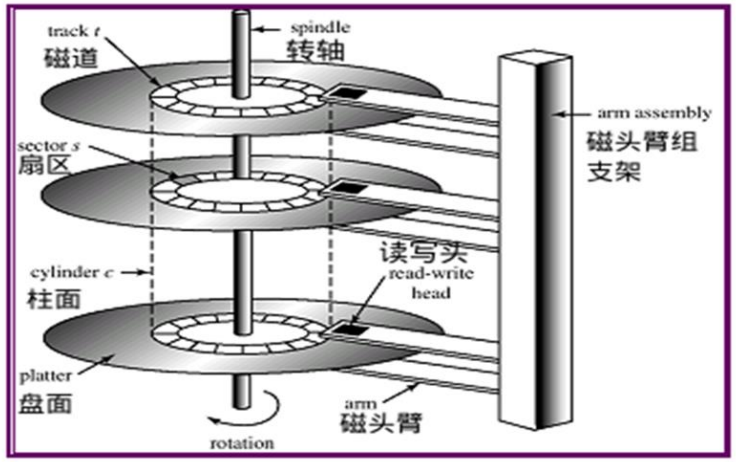
缓存技术在内存中缓存系统最近访问过的磁盘数据块，从体系结构角度降低磁盘访问平均耗时，并且可以大幅减少IO操作数量。

Part 1 磁盘调度算法重新排列、合并等待执行的IO请求，降低IO请求块的平均等待时间。

同济大学计算机系

机械硬盘

- 机械硬盘，同心圆盘。
  - 每个面，布有用来存储信息的许多同心磁道。从外至内编号，0，1，2，3。。。磁道号。所有盘面，相同磁道号的磁道组成柱面。**柱面号**就是**磁道号**。
  - 每个磁道，许多等圆心角圆弧。每个圆弧1个扇区。0，1，2，3 。。。**扇区号**。所有磁道，扇区数量相等。
  - 每个圆盘，上下2个盘面，每个盘面配有一个用来读写的磁头。磁头编号，0，1，2，3.。。。**磁头号**。
- 磁盘，信息存储、访问单位为扇区，目前的技术标准是每个扇区存放512字节。扇区的物理地址：柱面号、磁头号、扇区号。序列化，得扇区的逻辑地址。

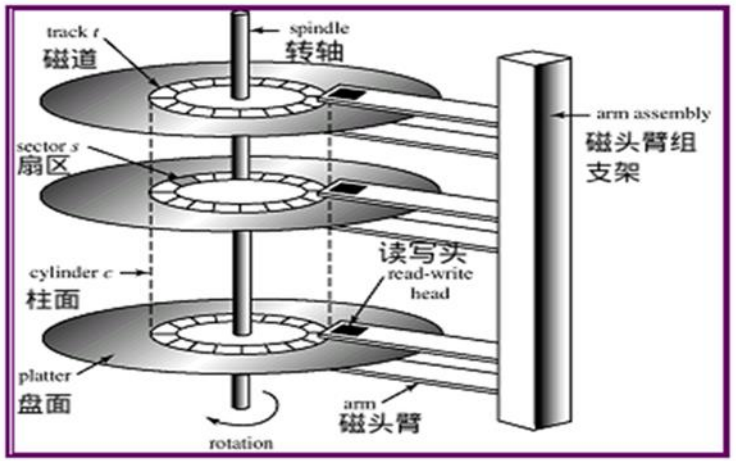


一张只有1个盘片，2个盘面的机械硬盘，每根磁道包含 n个扇区：

0#柱面，0#磁头：逻辑扇区号 0 ~ n-1
0#柱面，1#磁头：逻辑扇区号 n ~ 2n-1
1#柱面，0#磁头：逻辑扇区号 2n ~ 3n-1
1#柱面，1#磁头：……

机械硬盘（移动头磁盘）工作的物理特性

- 每个盘面配有一个读写头，所有读写头
  - 位于同一个柱面，同一个扇区
  - 同步移动，移动方向：沿半径方向里外移动
- 机器加电后，所有盘片同步、等速圆周运转。
- 传统的机械硬盘，每次只能读写一个扇区。磁盘控制器给出扇区号，硬盘
  - 将目标扇区号 分割成 柱面号、磁头号 和 扇区号
  - 移动磁臂，将所有读写头从当前位置同步移动至目标磁道（柱面）
  - 等待目标扇区首部旋转至读写头所在的位置
  - 目标盘片读写头下降
  - 随着盘片的旋转，目标扇区经过读写头，磁盘读出（写入）512字节



读写头又叫做磁头

硬盘访问时间（一个扇区）

硬盘访问时间 = 寻道时间 + 旋转延时 + 数据传输时间

- 磁头定位时间
  - 寻道时间：磁头从当前柱面移动到目标柱面的时间。平均寻道时间小于10ms。
  - 旋转延时：磁头到达目标柱面（磁道）后，等待欲访问扇区转到磁头之下。平均旋转延时为磁盘旋转一周所需时间T的1/2。
- 数据传输时间
  - 磁头经过目标扇区的耗时。若每个磁道包含N个扇区。数据传输时间为：T/N。

优化硬盘访问时间

- 硬盘是可以供多个进程共享的外部设备。
- 当有多个进程需要执行IO操作时，合理调度硬盘访问请求，可以缩短平均访问时间。提高硬盘吞吐率。

- 优化硬盘访问时间的思路：
  - 缩短寻道时间
  - 缩短旋转延迟
  - 。。。

优化寻道时间（磁臂调度算法）

例：以下是IO请求序列，数字是需要访问的磁道。 当前磁道号：100。

55，58，39，18，90，160，150，38，184；

排列这些磁道，使得总寻道时间最短（磁臂移动距离最短）

已知：寻道时间 = m*n + s。m 和 s 是磁盘的硬件参数，分别表示磁头移动一根磁道的耗时 和 磁臂启动的耗时。n 是磁头当前所在磁道 和 目标磁道之间的距离。

所以，磁臂移动磁道数越小，总寻道时间就越短。

FCFS 和 SSTF

IO请求序列：**55，58，39，18，90，160，150，38，184；**　　磁头当前位置：**100**

1、FIFS（FIFO）：按 IO请求提交次序服务。
响应次序：55，58，39，18，90，160，150，38，184。
磁头移动轨迹：100，55，58，39，18，90，160，150，38，184。
磁臂移动平均距离：55.3 个磁道

性能差，不存在饿死现象

2、SSTF，最短寻道时间优先。优先响应离磁头当前位置近的请求。
响应次序：90，58，55，39，38，18，150，160，184；
磁头移动轨迹：100，90，58，55，39，38，18，150，160，184。
磁臂移动平均距离：27.5 个磁道

性能好，会饿死远端请求

电梯调度算法

IO请求序列：**55，58，39，18，90，160，150，38，184；**　　磁头当前位置：**100，向大号磁道移动**

3、SCAN（电梯调度算法）：考虑磁头的当前位置和运动方向，依次服务途径的所有磁道上的服务请求。当磁头到达最高或最低磁道时（或前方不再有请求时），改变运动方向。

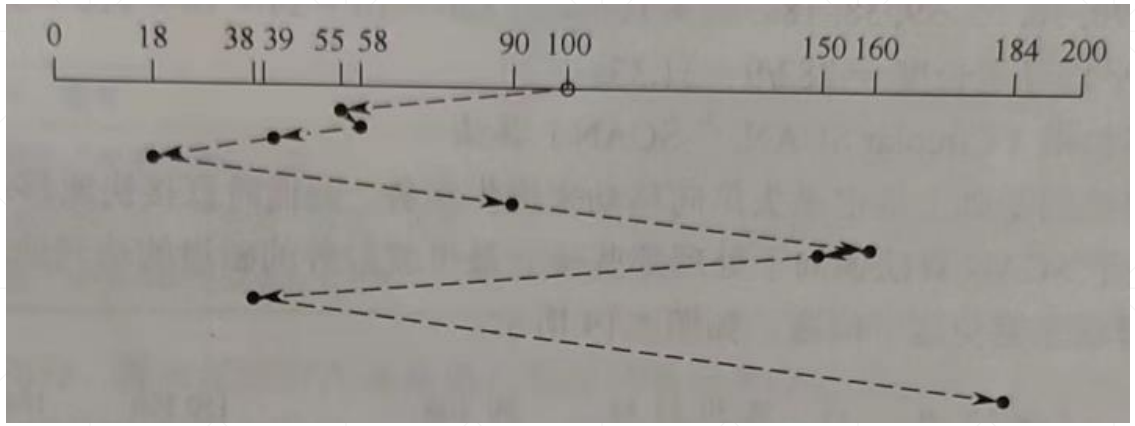
响应次序： 150，160，184，90，58，55，39，38，18。
磁臂移动平均距离：27.8 个磁道

兼顾了性能和公平

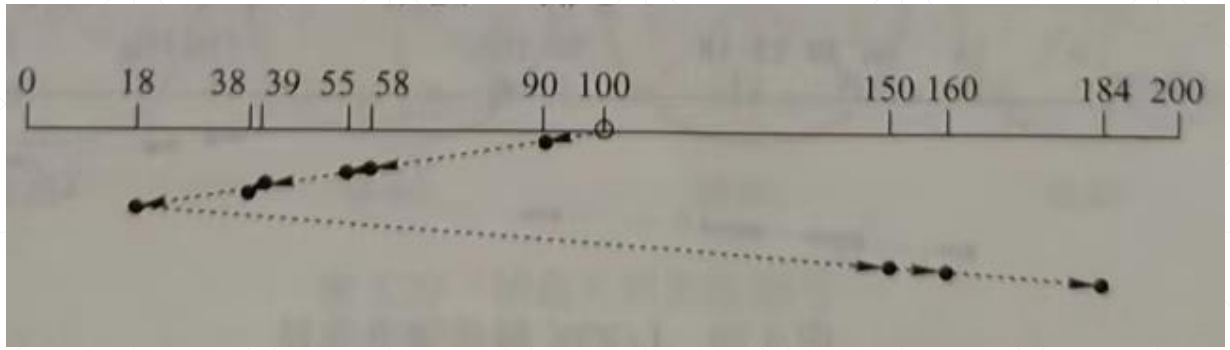
4、C-SCAN（Circular Elevator Algorithm）：与SCAN类似，但磁头折返时不服务。

磁头移动轨迹： 150，160，184，18，38，39，55，58，90；
磁臂移动平均距离： 35.8个磁道

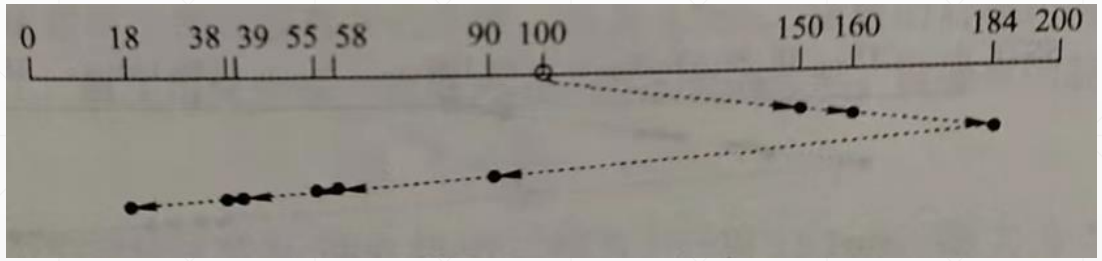
IO请求序列：**55，58，39，18，90，160，150，38，184；**　　磁头当前位置：**100**



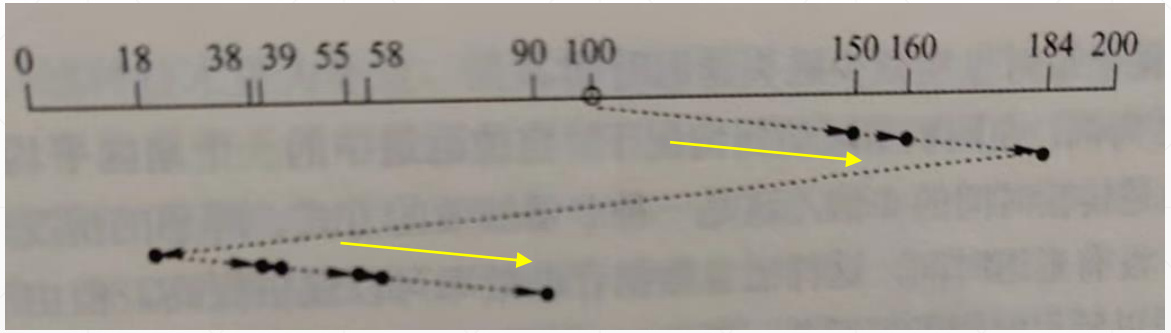
1、FCFS



2、SSTF



3、SCAN



4、CSCAN

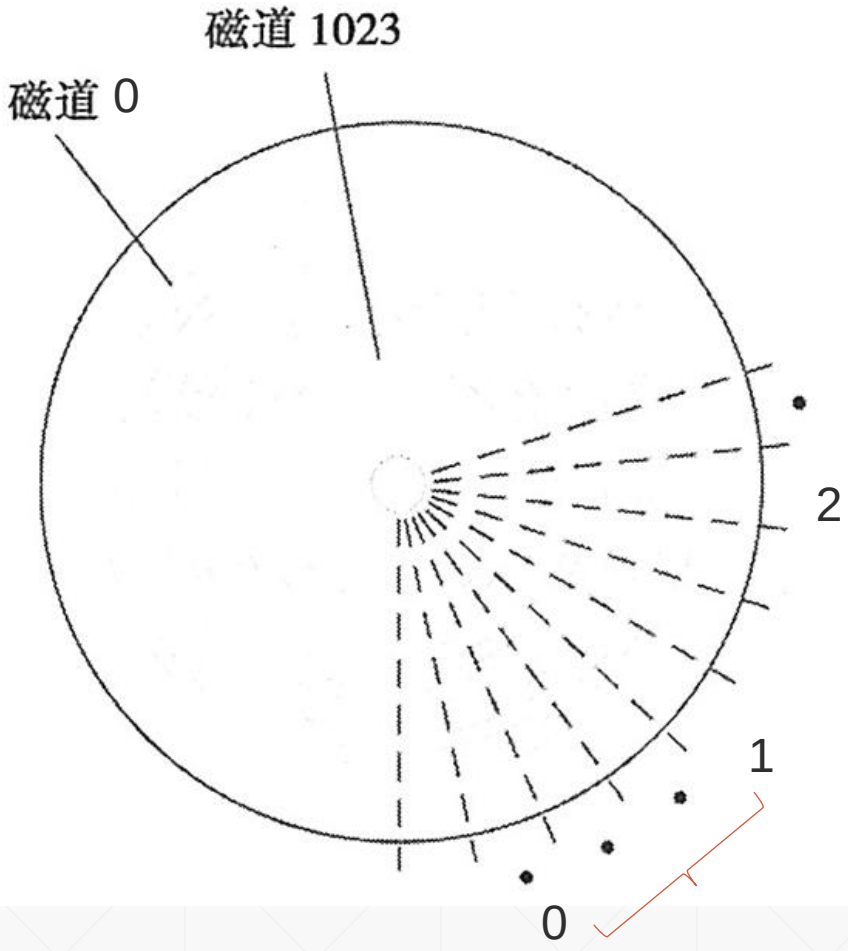
算法性能比较 和 进一步的改进

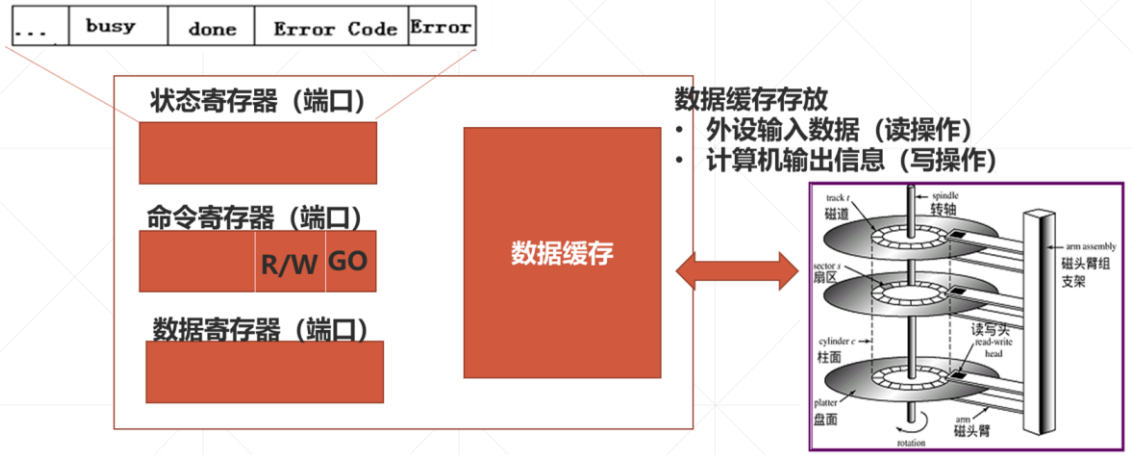
- SSTF，SCAN和C-SCAN会出现磁臂粘着，IO请求被饿死的情况。
- 磁臂粘着（磁头粘滞）：程序对某些磁道频繁访问，导致io队列中存在很多针对少数磁道的IO请求，其它磁道的请求被搁置、长期得不到服务的现象。
- 可以防止磁臂粘着的算法

(1) FSCAN（fair）：维护2个IO请求集合，active 和 pending。算法调度active集合中的请求，在该集合变空之前，新请求放入pending集合。active集合变空的时候，交换active 和 pending指针。 被调度的是固定集合，所以算法不会产生饿死现象。

(2) N-Step-SCAN算法：算法维护一条最大长度为N的io请求队列 和 一条不限长度的io等待队列。算法调度io请求队列中的请求。新进的请求（1）io请求队列 队长小于N时，进io请求队列（2）否则，进io等待队列。io请求队列变空时，算法从等待队列中取最多N个元素，加入io请求队列。 FIFS，无饿死现象；其余，会改善磁盘的io吞吐率。

优化寻转延迟（硬件）

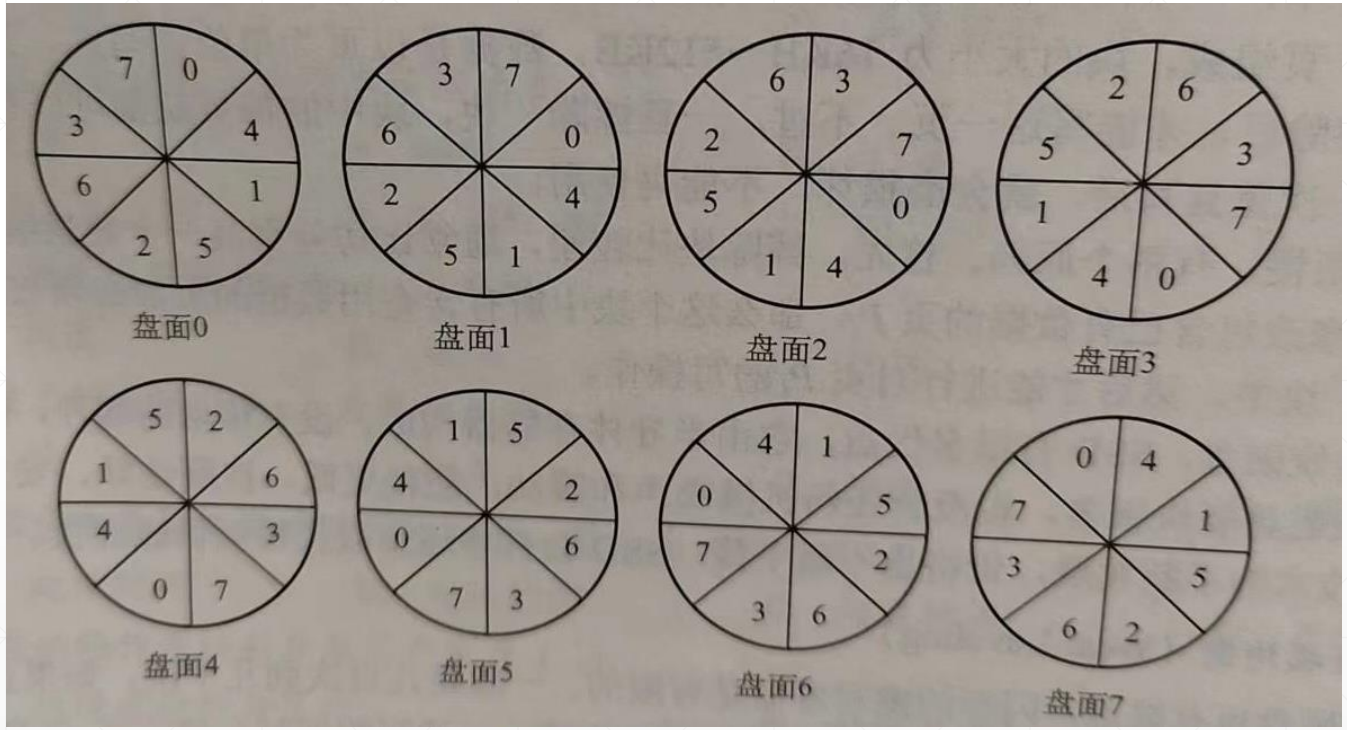




相邻扇区号对应的扇区，物理上一定不能相邻（DMA数据传输期间，磁头一定会刚好错过下个物理相邻的扇区首部）。
∴在同一磁道上，n扇区和n+1扇区之间一定会相隔若干个扇区。
下面的问题是：应该相隔多少个物理扇区为好呢？
答案是：磁头越过这几个扇区需要的时间必须大于DMA芯片将512字节从磁盘数据缓存读入内存所需的时间。
精确的值需要根据硬件参数仔细计算，并留一定余量。

2T/N > DMA传输的耗时

不同盘面，扇区错位编号



相邻盘面，扇区编号错一位

盘面0 7#扇区访问完毕后，收起0#盘面读写磁头，放下1#盘面读写磁头后，通过磁头的第1个扇区是 0# 扇区

例： 关于磁盘文件布局

移动硬盘，技术参数如下：转速7500rpm，**平均寻道时间为4ms**，每磁道500扇区，同磁道内相邻逻辑扇区错开2个物理扇区。问：读取一个1.28M（十进制）字节的文件需要多长时间?

- **每分钟7500转，旋转1周时间为 8ms。**
- **平均的旋转延迟 4ms**
- **一个扇区的读取时间为 8ms/500**
- 1.28M/500=2500 sector（扇区）

若2500扇区完全随机存放。

访问每个扇区都要经历寻道和旋转延时：

4ms+4ms+（8ms/500）

读取该文件需要：

2500*（4ms+4ms+（8ms/500））= 20.04s

若2500扇区顺序排在相邻5个磁道内。

访问第一个500个扇区：

$$4ms+4ms+8ms*(2\frac{498}{500}) \approx 32ms$$

后续4个磁道不需要寻道，每个磁道的访问时间：

$$4ms+8ms*(2\frac{498}{500}) \text{ 约等于 } 28ms$$

读取该文件耗时 ≈ 32+28*4 = 144ms

考虑硬盘物理特性优化文件传输速度

- 文件在磁盘上尽量连续存放，遍历文件耗时短。swap分区上的进程图像必需连续存放。

Part 2、预读技术

- 为什么要使用预读技术

- 力争重复使用原先读、写过，现在尚留在缓存中的字符块是UNIX缓存技术的一个重要目的，但是对块设备的读操作仍然是不可避免的。原因在于：（1）对某个字符块的第一次读操作一般总是对块设备进行的（除非在此之前对该块进行过写操作）。（2）原先对某个字符块虽然进行过读、写，但是由于对缓存的竞争使用，它占用的缓存区已被重新分配改作它用，因此当再次使用时，就必须从块设备上读入。

- 以同步方式读块设备时，进程不得不进入睡眠状态以等待数据传输结束，因此速度是相当低的。为了加快进程前进速度，提高中央处理机和块设备工作并行程度，最好在实际使用某字符块前，用异步方式提前将它读入缓存，在实际使用时，可立即从缓存中取用而无需等待。这种读字符块的方式称为**预读**。

一、Unix V6++系统中使用的预读技术

- read系统调用读普通文件数据块时使用预读技术

- Unix V6++预读字符块的原则：对某个文件，考察相邻2次读操作所在的逻辑块，如果前一次的逻辑块号 + 1 = 当前逻辑块号，判定为顺序读。**同步读入当前逻辑块时候，异步读入下一个逻辑块。**

概念

（1）文件偏移量 offset。第0个字节，offset是0；第1个字节，offset是1 。。。是每个字节在文件中的逻辑地址

（2）文件的逻辑块是文件中的数据块。文件中 [0,512）字节组成0#逻辑块，[512，1024) 字节组成1#数据块 。。。

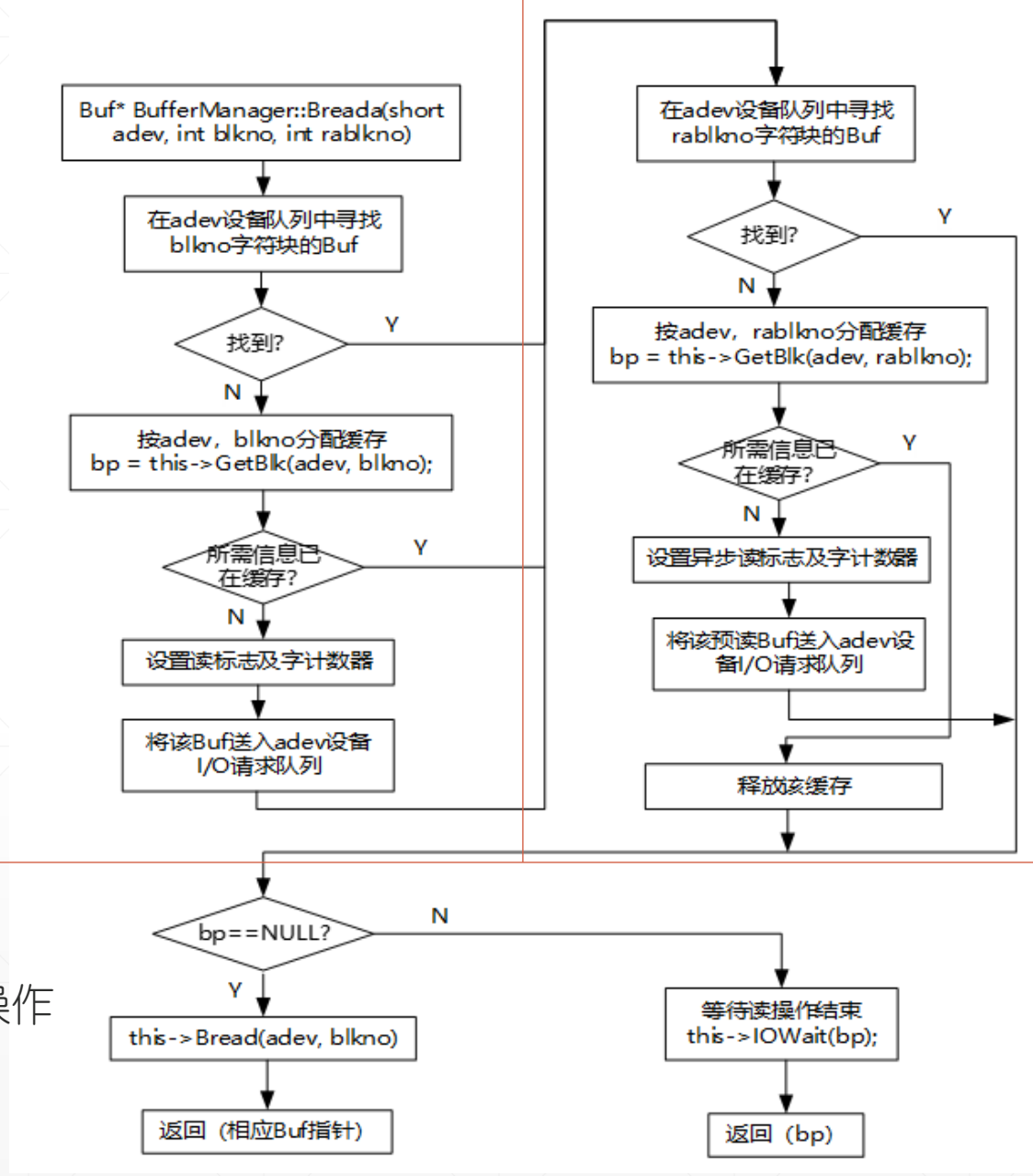
Breada(adev, blkno, rablkno)

- 入口参数
  - adev，文件所在的磁盘设备号
  - blkno，当前块。当前逻辑块的物理块号
  - rablkno，预读块。下个逻辑块的物理块号

- 功能
  - 同步读入当前块，异步读入预读块

adev，磁盘设备号
blkno，当前块
rablkno，预读块

1、将磁盘中的当前块读入缓存



2、异步读入预读块

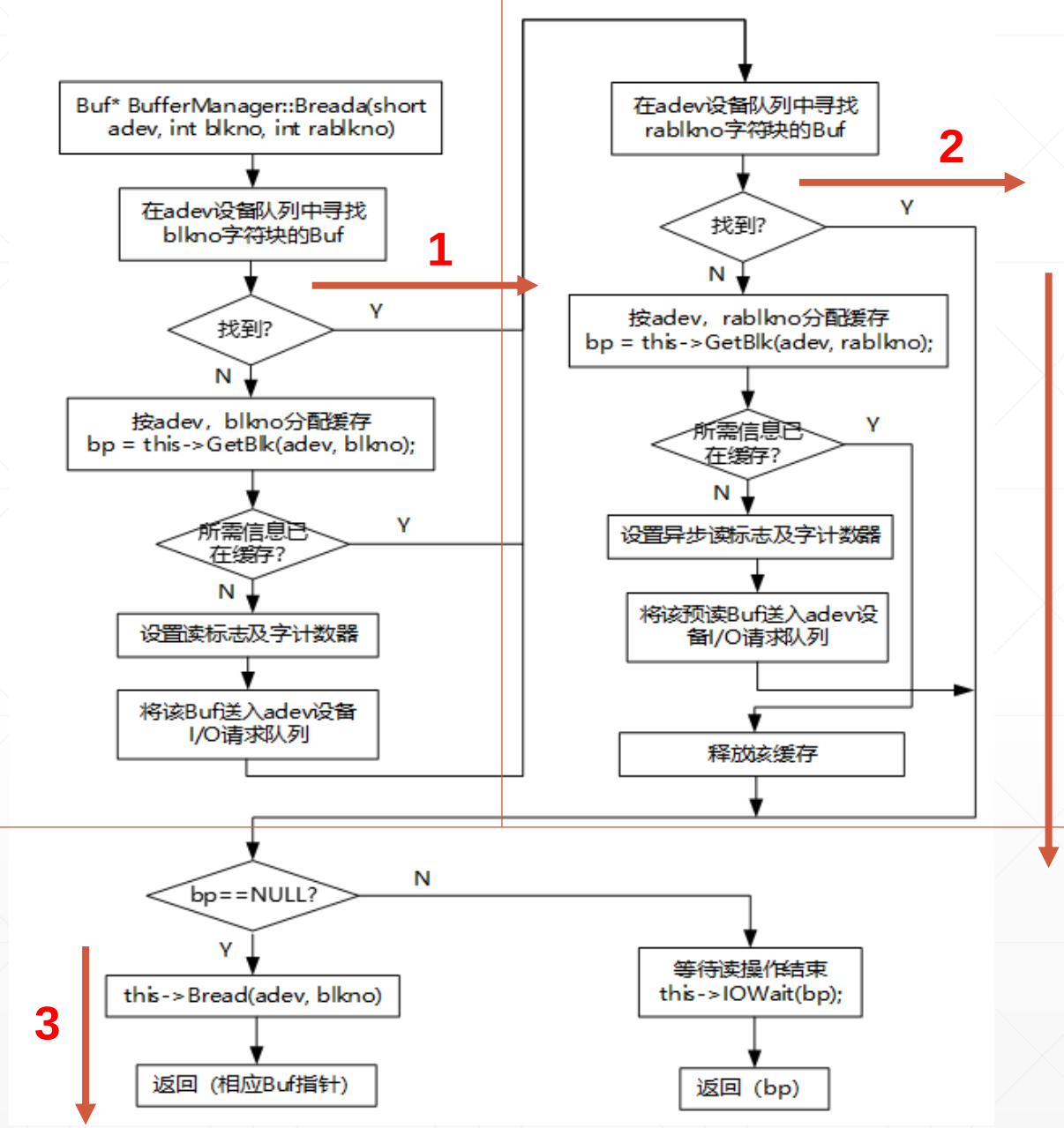
（io完成时，中断处理程序置1 B_DONE，清0 B_BUSY、解锁预读块）

3、同步当前块的读操作

adev，磁盘设备号
blkno，当前块
rablkno，预读块

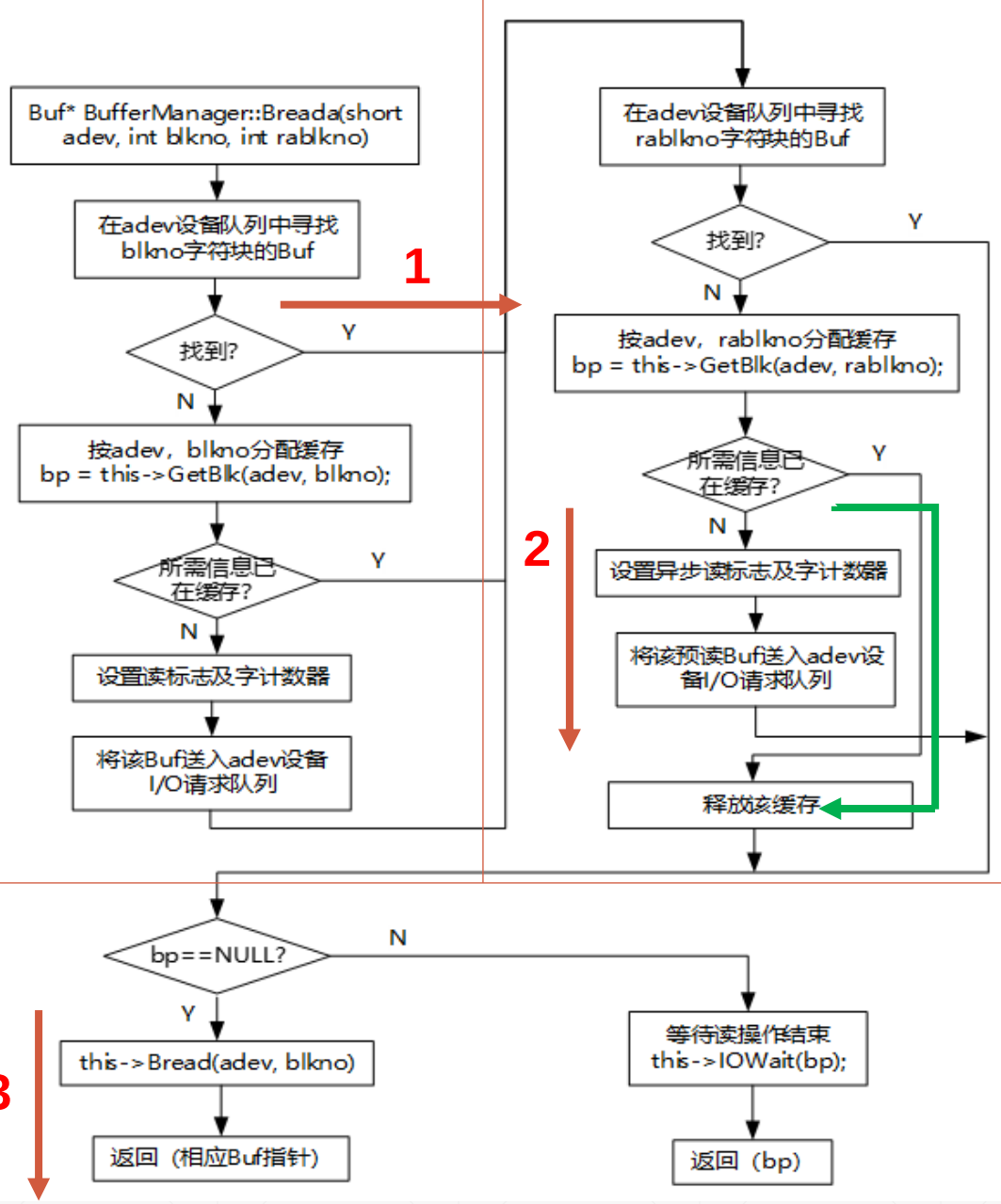
当前块、预读块缓存均命中

同步当前块



adev，磁盘设备号
blkno，当前块
rablkno，预读块

当前块缓存命中
预读块缓存不命中



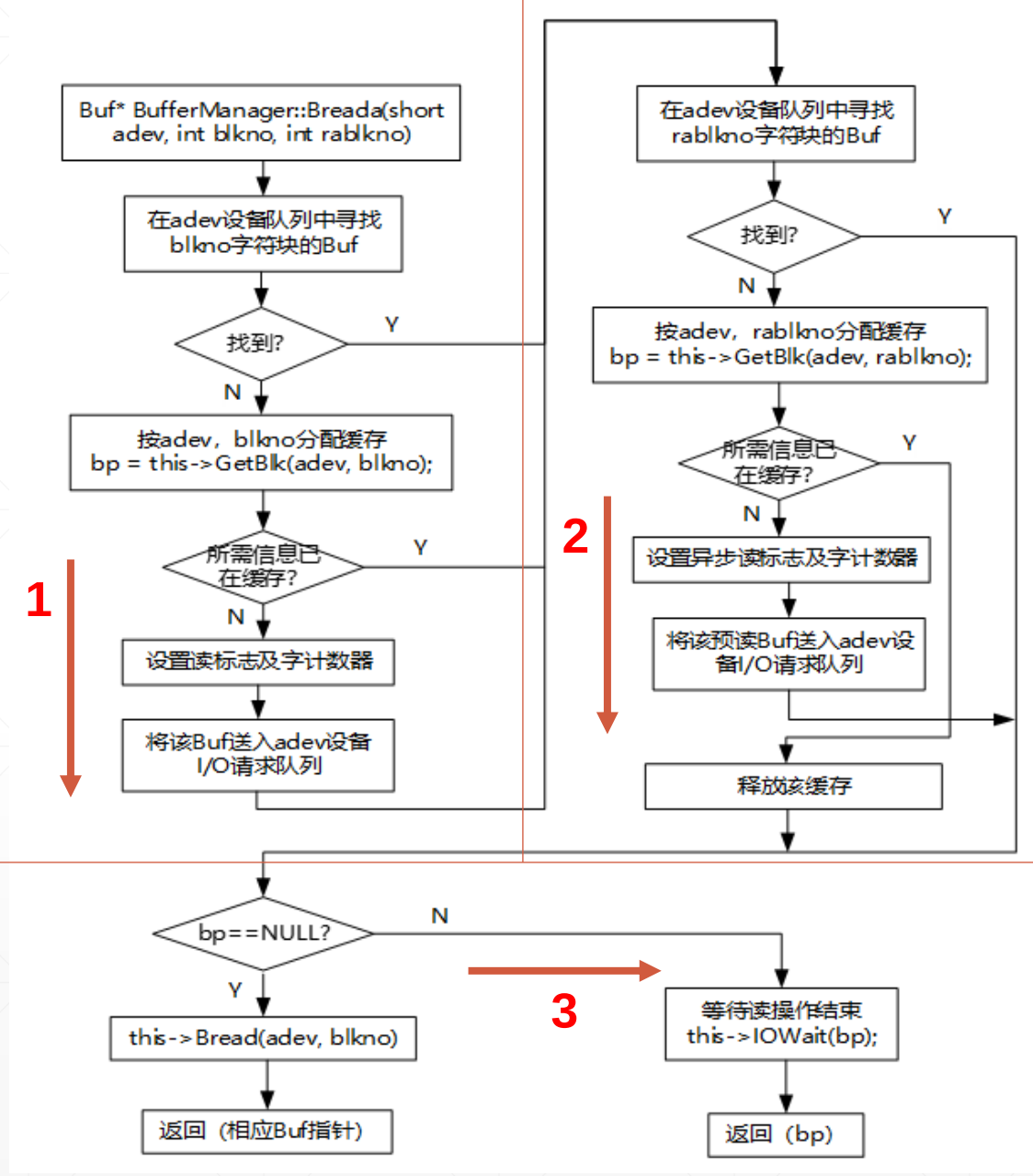
2、为预读块分配缓存块，构造异步IO请求块，送io队列尾部

构造预读块会很快的。除非自由缓存队列空，在这种情况下，进程会入睡等待自由缓存。

PA放弃CPU后，会有当前块被淘汰的可能，会有其它进程申请空闲缓存装rablkno的可能，如若后者先被调度，会首先发起rablkno的IO请求，预读块缓存命中，PA走绿色路径。

adev，磁盘设备号
blkno，当前块
rablkno，预读块

当前块缓存不命中
预读块缓存不命中



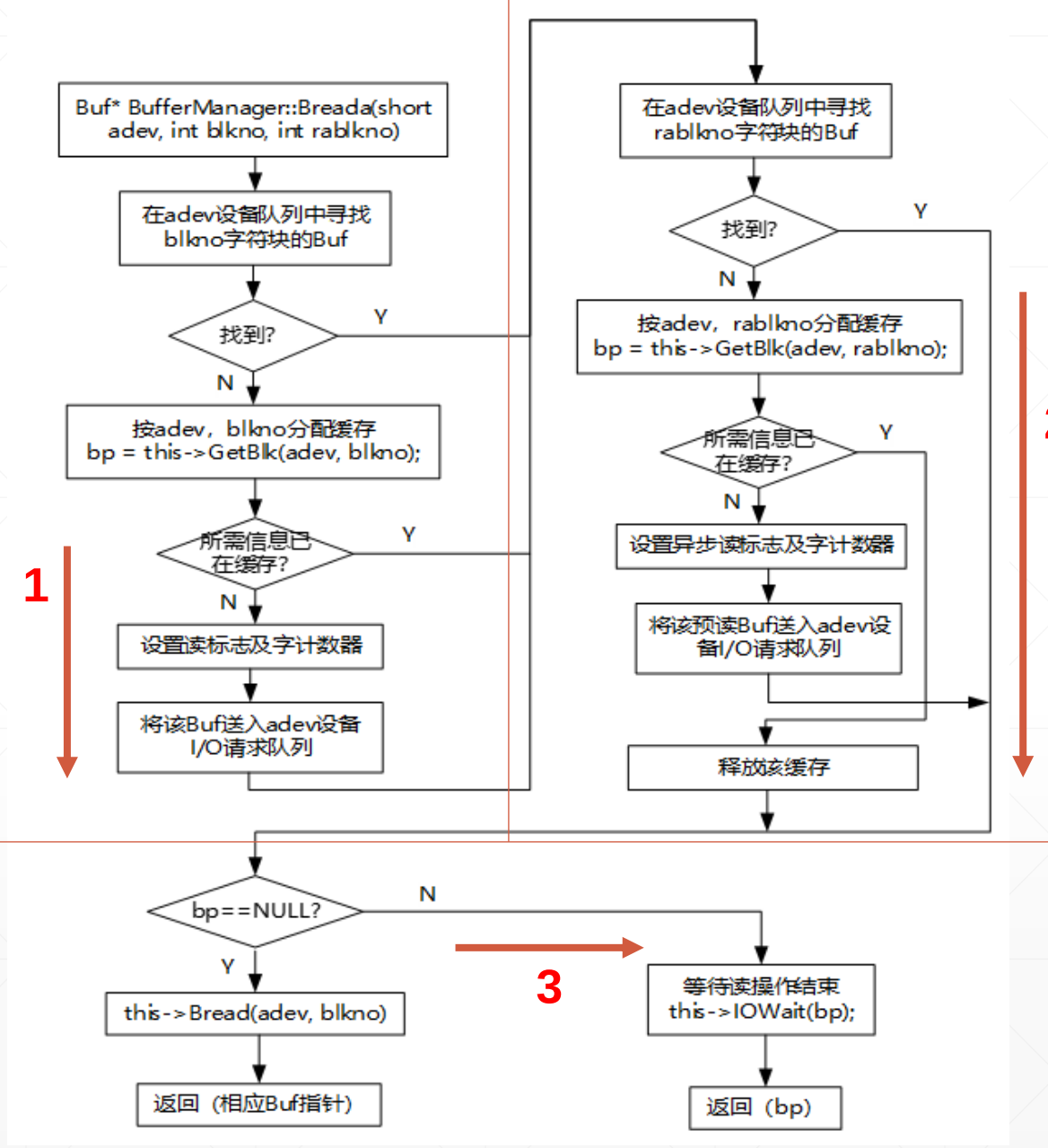
1、构造当前块的IO请求块（同步块），送io队列尾。

2、构造预读块的IO请求块（异步块），送io队列尾。

3、同步等待IO操作结束。

adev，磁盘设备号
blkno，当前块
rablkno，预读块

当前块缓存不命中
预读块缓存命中



1、构造当前块的IO请求块（同步块），送io队列尾。

2、预读块缓存命中，什么也不要做

3、同步等待IO操作结束。

Unix V6++的read系统调用，核心部分

read系统调用同步读入磁盘文件中的连续的一块数据，送进程用户区。

- IO参数
  - m_offset，数据在文件中的偏移量
  - m_base，进程用户区首地址
  - m_count，进程需要读入的数据量
- 返回值
  - 0，EOF文件尾，未读到数据
  - 正整数，读入的数据量不足，剩余字节数

预读标识：i_lastr：　每个使用中的文件一个i_lastr，记录上次 IO的逻辑块号。

read( ) 系统调用框架(readi 核心部分)

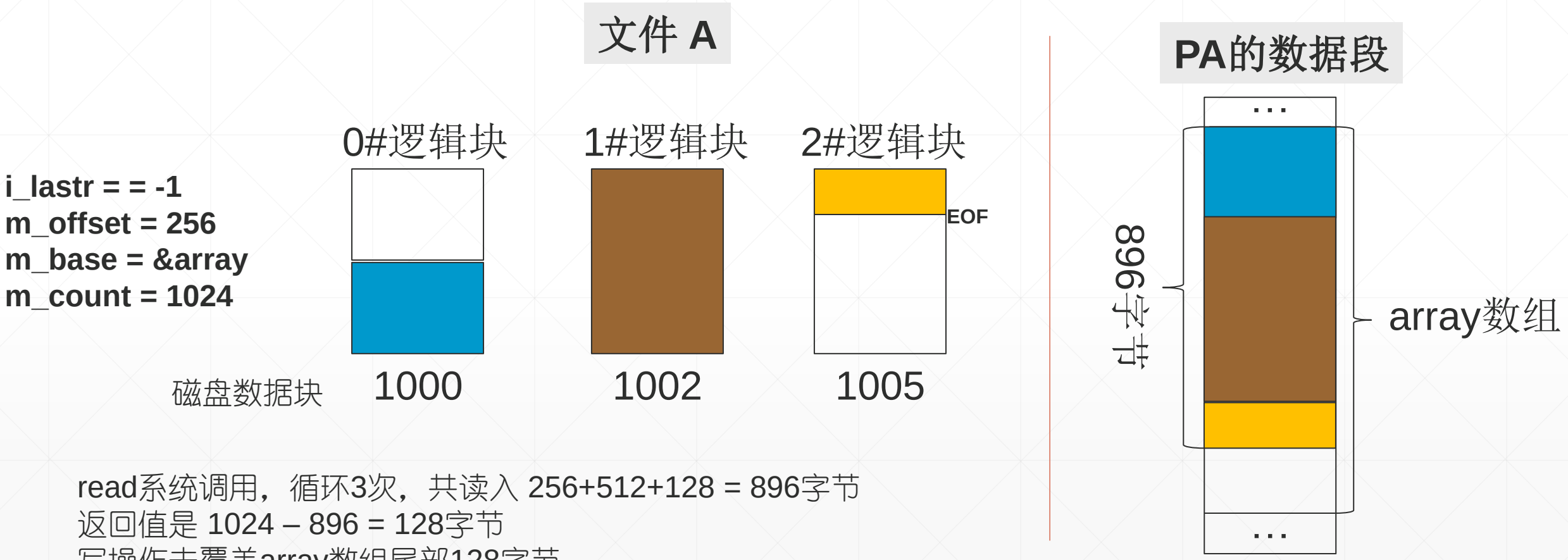
循环 （1 次读入1个逻辑块）

- 逻辑块号　bn = m_offset / 512
- 块内偏移量　o = m_offset % 512
- 计算本块中要读取的字节数 n
  - 不是文件的最后一块　n=min(512 – o, m_count)
  - 是，n=min[文件长度 f_size % 512 – o, m_count]

- 用 bn 查文件地址表，得当前块的物理块号 blkno
- 用 bn+1 查文件地址表，得预读块的物理块号 rablkno（0表示当前块是文件的最后一块）
- if(bn == i_lastr+1）//顺序读的判断
  bp=breada(dev,blkno,next)
  else
  bp=bread(dev,blkno)

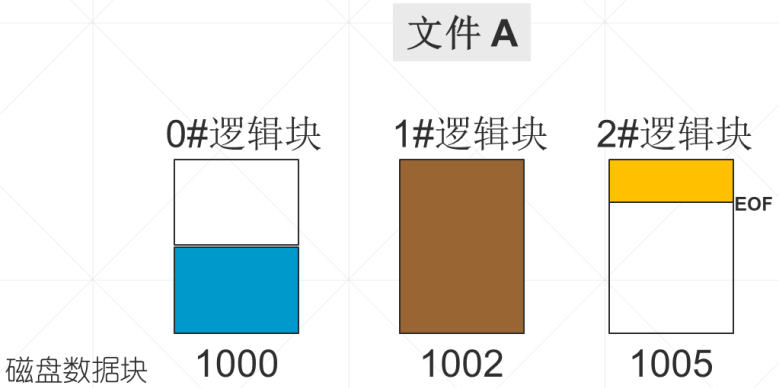
- iomove(bp->b_addr+o，m_base，n)
- brelse(bp)
- m_offset+=n
- m_base+=n
- m_count-=n
- if （ m_count == 0 | m_offset == f_size ）
  return （ m_count ）

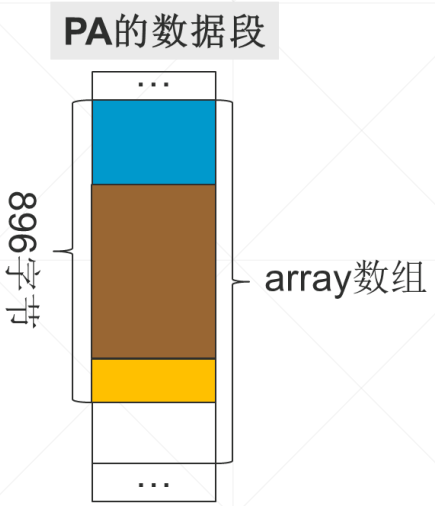
例：假设文件A长1152字节。应用程序数组array，长1024字节。试分析系统调用n=read(fd,&array,1024) 的执行过程。



read系统调用，循环3次，共读入 256+512+128 = 896字节
返回值是 1024 – 896 = 128字节
写操作未覆盖array数组尾部128字节

i_lastr = = -1
m_offset = 256
m_base = &array
m_count = 1024





每次循环时使用的工作变量（进程的 user结构，u_IOParam（IO参数））

逻辑块	m-offset	m-base	m-count
0#	256	&array	1024
1#	512	&array+256	1024-256=768
2#	1024	&array+768	768-512=256
	1152	…	256-128=128

每次循环使用的缓存读写操作

逻辑块	i_lastr（前	当前块bn	块读写操作	IOmove函数的参数	i_lastr（后
0#	-1	0	breada(0,1000,1002)	bp->b_addr+256, &array, 256	0
1#	0	1	breada(0,1002,1005)	bp->b_addr, &array+256, 512	1
2#	1	2	bread(0,1005)	bp->b_addr, &array+768, 128	2

read系统调用对缓存的使用

逻辑块	i_lastr（前	当前块bn	块读写操作	IOmove函数的参数	i_lastr（后
0#	-1	0	breada(0,1000,1002)	bp->b_addr+256，&array，256	0
1#	0	1	breada(0,1002,1005)	bp->b_addr, &array+256，512	1
2#	1	2	bread(0,1005)	bp->b_addr, &array+768，128	2

- 假设PA执行read系统调用时，所有数据块缓存不命中。
- breada(0,1000,1002)，淘汰自由缓存队列队首的2个缓存，用来装1000#扇区和1002#扇区。放IO请求队列，排一起，睡眠等待1000#扇区IO完成，将数据送入array数组
- breada(0,1002,1005)，淘汰自由缓存队列队首缓存，装1005#扇区，送IO请求队列，breada睡眠，等待复用1002#扇区，置b_wanted标识，被唤醒后将数据送入array数组
- bread(0,1005)，等待复用1005#扇区，置b_wanted标识，被唤醒后将最后一块数据送入array数组
- EOF，read系统调用返回。

磁盘中断处理程序：

- 1000#扇区 IO完成，唤醒PA。启动1002#扇区 IO
- 1002#扇区IO完成，缓存块解锁，唤醒PA。启动1005#扇区 IO
- 1005#扇区IO完成，缓存块解锁，唤醒PA。IO请求队列空，中断处理程序直接返回。

现代操作系统中，预读的作用

- 分配给文件的物理块基本上是相邻的。
- 如果缓存不命中，将 当前块 和 预读块 一并送入io请求队列可以有效减少磁臂移动总距离，提高磁盘工作效率。

预读技术的优点（补）

- 减少磁头移动距离，特别有利于改善顺序读时 IO 的平均耗时。

- 例： PA进程顺序读 fileA 的全部内容。与此同时，进程PB需要读取 9999#扇区中另一个文件的内容。假设，所有数据缓存不命中， fileA的0#逻辑块和1#逻辑块分别存放在1000，1002#物理块。
  - fd = open(fileA,***)
  - read(fd,array,512);
  - read(fd,array,512);

1000

1002

- 使用预读技术，读入当前块1000时，异步读入1002。第二次read系统调用缓存命中概率很高，不会引发IO操作。IO请求队列 …… → 1000,1002 → 9999，读取这3个扇区，磁臂移动8999根磁道。
- 不使用预读技术，读入1000，1002块要执行2次read系统调用。第一次read系统调用进程一定会睡眠放弃CPU，随后PB上台读取9999#扇区 …… ，PA恢复运行读1002。
  - IO请求队列 …… → 1000 → 9999 → 1002，读取这3个扇区，磁臂移动 8999 + 8997根磁道。性能远不及预读。