unix 中数据缓冲区高速缓冲的设计

作者：vcjmhg
原文链接：https://ld246.com/article/1588079207956
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

[TOC]

1. 概述

操作系统对文件系统的一切存取操作，内核都能通过每次直接从磁盘上读或往磁盘上写来实现。磁盘和RAM 的速度之间差别很大。由于两者速度的不匹配性，在操作系统实际运行的过程中可能会出现以问题：

1. 磁盘机械运动速度大大低于处理的运行速度；
2. 多线程并发运行，少量的磁盘（通道），I/O 操作将会成为瓶颈所在；
3. 数据访问的随机性，磁盘忙闲不均。

为了解决上面的问题，Unix 设计者在设计内核时，通过保持一个称为数据缓冲区高速缓冲（ buffer cche）（简称高速缓冲）的内部数据缓冲区池来试图减小对磁盘的存取频率。具体的解决办法如下：

1. 建立一个成为数据缓冲的高速缓冲的内部数据缓冲区池（buffer pool）来存放使用的数据；
2. 写数据时

把数据尽量长时间的保存在缓冲池中，**延迟**写出到磁盘上，以备后续使用。

也就是说在写数据时，并不是真正的把数据直接写在磁盘上，而是先写到缓冲池中，供后续进程使用尽量少的减少与磁盘交互的频率。

3. 读数据时

现在缓冲池中查找已有的数据，如果没有，再从磁盘中读取，并保存的缓冲池中，使用的时候再从缓池取。

从写数据和读数据的操作过程中我们可以看出，通过缓冲区的引入，使得进程更多的与缓冲区来进行换数据，尽量少的减少磁盘的读写频率，提高读写速度。

原文链接：unix 中数据缓冲区高速缓冲的设计

2. 缓冲区的设计

缓冲区池由若干缓冲区组成，每一个缓冲区由两部分组成：一个含有磁盘上的数据的存储器数组及一用来标识该缓冲区的缓冲头部（buffer header）。其示意图如下所示：

但是，由于数据缓冲区的首部和存储区之间有一一对应的关系，所以通常把两者统称为缓冲区。

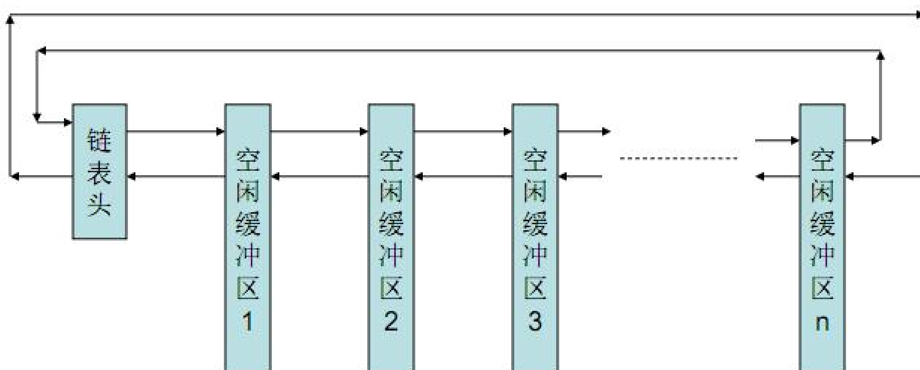
注意：缓冲区是缓冲区池中数据存储的基本单位。

2.1 缓冲区头部

缓冲区的头部定义如下：

```
struct buf{
    缓冲区标注            //标识缓冲区的的状态即是否被使用
    缓冲区连接指针          //向前向后串成链表
    空闲缓冲区链表指针      //用来链接空闲缓冲区
    设备号              //用来标识缓冲区
    块号
    union{              //缓冲区中的数据类型
        数据块
        超级快
        柱面块
        i节点块
    }b_un
    其他控制信息
}
```

为了便于理解我们用一个图来表示其结构。



原文链接：unix 中数据缓冲区高速缓冲的设计

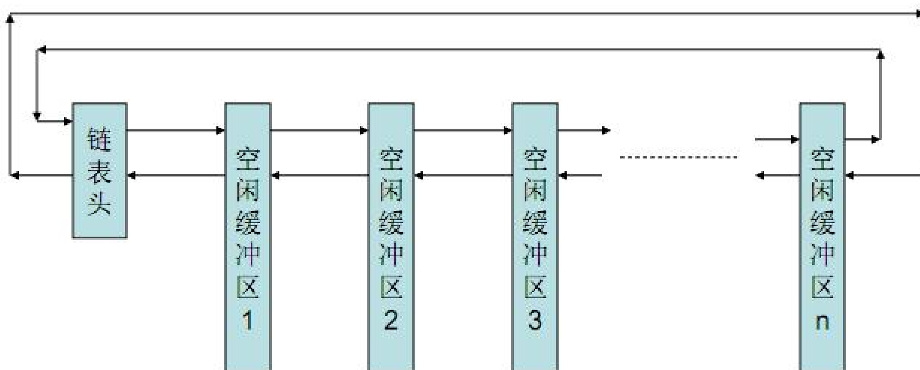
2.2 缓冲区的结构

在详细说明缓冲区的结构之前我们有必要了解一下缓冲区在设计的过程中所遵循的原则：

1. 存放有刚使用过的数据尽量长时间地保留在内存中，以便马上还要使用时能在内存中找到；

2. 需要腾出内存空间时，把很久都未使用过（即最近最少使用）的数据交换到磁盘上去。这些数据马还要使用的可能性最小。

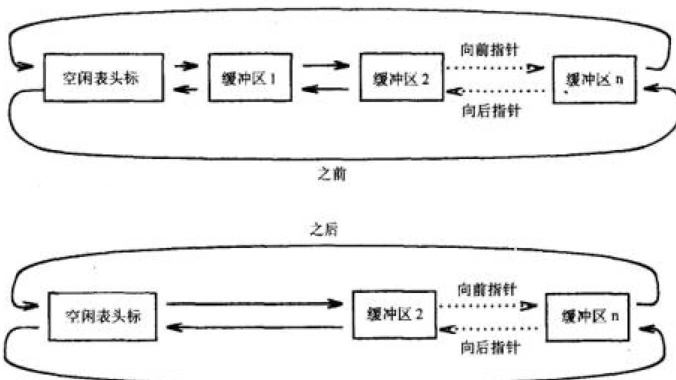
缓冲区在实现的过程中，其核心维护了一个**空闲缓冲区链表**，它按照最近被使用的先后次序排列。空链表是一个以空闲缓冲区链表头开始的“双向循环链表”。链表的开始和结束都以链表头为标志。其体结构如下图所示：



缓冲区操作的具体过程如下：

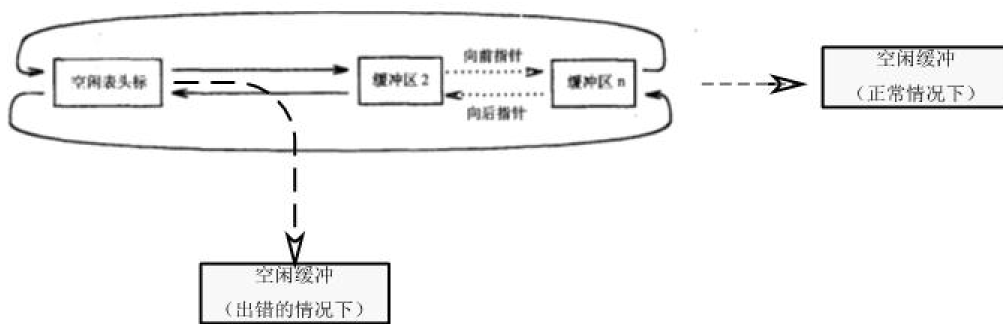
1. 取用任意空闲缓冲区

从空闲缓冲区链表的表头位置取下一个空闲缓冲区，后面的空闲缓冲区依次向前移动。以上图为例，空闲缓冲区时，会取走空闲缓冲区 1，然后空闲缓冲区 2-n，会一次向前移动。



1. 释放一个空闲缓冲区

把这个装有数据的空闲缓冲区附加到空闲链表的链尾，只有当该空闲缓冲区所装数据出错时才挂到链的后边。



2. 取用装有指定内容的空闲缓冲区

从链表头开始查找，找到后取下使用，用完后放到链尾。

当系统不断从链头取用空闲缓冲区，又把使用过的（装有数据的）缓冲区挂到链尾，一个装有有效数的缓冲区就会逐渐向链表头移动。在链表头位置的就是“最近最少使用”的空闲缓冲区。

另一方面，为了提高缓冲区的使用效率，**避免在取用缓冲区的时候逐个判断缓冲区中的内容**，缓冲区设计的时候按照不同的用途将空闲缓冲区分为四类：

- **0#**空闲缓冲区链表：存放系统文件超级快
- **1#**空闲缓冲区链表：存放通常使用的数据块
- **2#**空闲缓冲区链表：存放延迟写、无效数据或者错误内容
- **3#**空闲缓冲区链表：存放没有对应存储空间的缓冲区首部

空闲缓冲区按照不同的用途进行分类，这样有一个最大的好处，程序在使用不同数据的时候只需要按数据的类型到制定的某个空闲缓冲区链表中查找而不需要查找所有的空闲缓冲区链表。

另外可能某些读者会有疑问，我们设计 **3#缓冲区链表**有什么用？按理来说缓冲区和外存的空间应该一一对应的怎么会出现没有对应存储空间的缓冲区这应的情况发生哪？

为了回答这个问题，我们首先要知道，unix 在设计之初，系统的稳定性就是其重要的指标，我们设定样一个场景，某台服务器在运行过程中与某个缓冲区对应的磁盘坏掉了。在这种情况下，为了保证系的稳定性，我们不可能重启服务器，重新实现缓冲区和磁盘空间的映射。因此我们设计**出 3#空闲缓区链表**，这样，上边没有存储空间的缓冲区都会被挂载到该空闲缓冲区链表中。当系统运维人员，更完存储空间后，系统再将该缓冲区从 3#空闲缓冲区中调出，从而保证系统运行的稳定性。

当核心需有一个空闲缓冲区时，它根据要装入的数据类型，从相应的空闲缓冲区链表的表头位置取下个空闲缓冲区，装入个磁盘数据块；

根据该数据块所对应的设备号和块号数据对计算其 hashed（散列、杂凑）值，根据其 hashno 的值入到相应 hash 链表的链头。

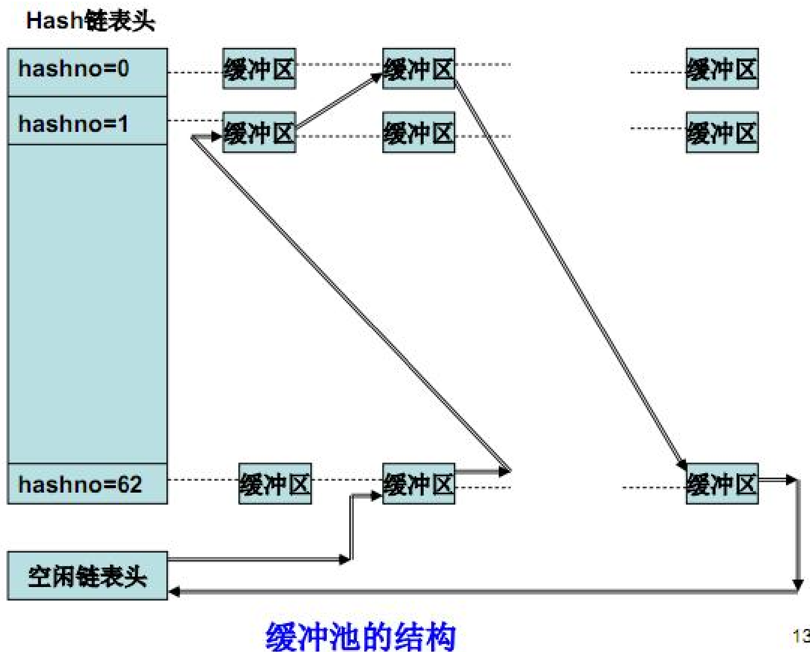
其中 hashed 计算规则如下：

hashno =((diskno + blkno) / RND) % BUFHSZ

- disko：设备号
- blkno：块号
- BUFHSZ：最大 hash 值，通常为 63。
- RND：随机数，其值为：RND= MAXBSIZE/ DEV BSIZE MAXBSIZE：最大文件系统块的大小 DEV

● BSIZE：物理设备块大小

经过前边知识的铺垫下边我们就可以详细了解下缓冲池的具体结构：



缓冲池的结构

从图中我们可以看到，一个缓冲区只有当它是**空闲状态**时，它才同时处在 hash 链表和空闲链表中。果不空闲，则它只能处在某一个 hash 链表中在空闲缓冲区链表中的缓冲区**一定在某个 hash 链表中**在 hash 链表中的缓冲区不一定在空闲链中。不存在脱离 hash 链表的另一个空闲的缓冲区链表。

缓冲池中的缓冲区个数是固定不变的，每个缓冲区在不同时刻存放着不同的磁盘数据块，具有不同的 ash 值，因此处在不同的 hash 链表中缓冲区中的数据与某个磁盘数据块一一对应，这种对应有两个点：

①一个磁盘数据块在缓冲池中最多只能有一个副本；

②缓冲区与数据块的对应是动态的，LRU 数据块将被释放。

缓冲区的使用过程如下：

如果**要找特定缓冲区**，根据 has hno 从相应的 hash 链表的表头处开始逐个向后查找；如果找到，则接取用，并将其移动到 hash 链的链头；如果未找到，则从相应的空闲缓冲区链表的表头处取下一个闲缓冲区，填入相应数据，重新计算其 hashed，并放到新的 hash 链表的表头；

释放缓冲区时，将该缓冲区仍保留在原 hash 队列中，同时挂接到空闲缓冲区链表的表尾。（同时在个队列中）

2.3 缓冲区的检索算法

在 UNX 文件系统中的下层，即直接与逻辑存储设备联系的部分，包含如下基本算法：

● gebk：申请一个缓冲区

● brelse：释放一个缓冲区

● bread：读一个磁盘块

● bread：读一个磁盘块，预读另一个磁盘块

● bwrite：写磁盘块

2.3. 申请一个缓冲区算法 getblk

根据缓冲池的结构，核心申请一个缓冲区分配个磁盘块时，可能出现的五种典型状况：

①该块已在 hash 队列中，并且缓冲区是空闲的；

②hash 队列中找不到该块，需从空闲链表中分配一个缓冲区；

③hash 队列中找不到该块，在从空闲链表中分配一个缓冲区时，发现该空闲缓冲区标记有“延迟写，核心必须写出缓冲区内容到磁盘上，再重新分配一个空闲缓冲区

④hash 队列中找不到该块，并且空闲链表已空；

⑤该块已在 hash 队列中，但该缓冲区目前状态为“忙”。

具体算法的思路如下：

```
算法 getblk
输入:文件系统号
块号
输出:现在能被磁盘块使用的上了锁的缓冲区
{
    while(没有找到缓冲区)
    {
        if(块在散列队列中)
        {
            if(空闲链表中无缓冲区)    //第五种情况
            {
                sleep(等候“缓冲区变为空闲”事件);
                   continue;
            }
            为缓冲区标记上"忙";      //第一种情况

            从空闲链表上摘下缓冲区;
            return(缓冲区);
        }
      else    //块不在散列队列中
      {
          if(空闲链表中无缓冲区)    //第四种情况
          {
              sleep(等待“任何缓冲区为空闲”事件);
              continue;    //回到while循环
          }
          //第二种情况找到一个空闲缓冲区
          把旧散列队列中摘下缓冲区;
          把缓冲区投入到新的散列队列中去;
          return(缓冲区);
      }
    }
}
```

2.3.2 释放一个缓冲区算法 brelse

```
算法 brelse
输入:上锁状态的缓冲区
输出:无
```

```
{
    唤醒正在等待"无论哪个缓冲区变为空闲"这一事件的所有进程;
    唤醒正在等待"这个缓冲区变为空闲"这一事件的所有进程;
    提高处理机执行级别以封锁中断;
    if(缓冲区内容有效且缓冲区非"旧")
        将缓冲区送入空闲链表尾部;
    else
        将缓冲区送入空闲链表头部;
    降低处理机执行级别以允许中断;
    给缓冲区解锁;
}
```

2.3.3 读一个磁盘块 bread

整个过程如下：

1. 由 getblk 算法申请一个可用的缓冲区
2. 如果缓冲区中的内容有效，则直接返回该缓冲区
3. 如果缓冲区中的内容无效，则启动磁盘去读所需的数据块
4. 等待磁盘操作完成后返回

```
算法 bread
输入:文件系统号
输出:含有数据的缓冲区
{
    得到该块的缓冲区(算法 getblk);
    if(缓冲区数据有效)
        return(缓冲区);
    启动磁盘读;
    slep(等待"读盘完成"事件);
    return(缓冲区)
}
```

2.3.4 读一个磁盘并预读另一个磁盘块 breada

预读的前提：

程序是在一个有限的空间内运行，程序对数据的访问是可预见的。

预读的命中率：

不一定达到 100%,但良好的系统结构和算法可使命中率达到较高的水平

预读的结果：

放在缓冲池内，以免需要的时候再去启动磁盘读数据块。

```
算法 bread
输入:(1)立即读的文件系统块号
     (2)异步读的文件系统块号
输出:含有立即读的数据的缓冲区
{
   if(第一块不在高速缓冲中)
   {
```

```
        为第一块获得缓冲区(getblk);
        if(缓冲区内容无效)
            启动磁盘度;
    }
    if(第二块不在高速缓冲中)
    {
        为第二块获得缓冲区(getblk);
        if(缓冲区数据有效)
            释放缓冲区(brelse);
        else
            启动磁盘读;
    }
    if(如果第一块在高速缓冲中)
    {
        读第一块(bread);
        return 缓冲区;
    }
    sleep(第一个缓冲区包含有效数据的事件);
    return 缓冲区
}
```

2.3.5 写餐盘块 bwrite

写操作分为两种，一种是同步写，另一种是异步写，这两种操作的根本区别在于本进程在进行写操作，是否等待磁盘驱动程序完成操作后所发出的中断信号。如果等则是同步写，否则为异步写。

```
算法 bwrite
输入:缓冲区
输出:无
{
    启动磁盘写;
    if(IO同步)
    {
        sleep(等待"O完成"事件);
        释放缓冲区( brelse);
    }
    else if(缓冲区标记着延迟写)
    {
       为缓冲区做标记以放到空闲缓冲区链表头部;
    }
}
```

3. 总结

unix 操作系统引入高速缓冲之后带来了极大的便利，但同时也有一些不足之处。下边我们分别总结下诉缓冲的优点以及其缺点。

优点:

1. 提供了对磁盘块的统一的存取方法
2. 消除了用户对用户缓冲区中数据的特殊对齐需要
3. 减少了磁盘访问的次数，提高了系统的整体 MO 效率

原文链接：unix 中数据缓冲区高速缓冲的设计

4. 有助于保持文件系统的完整性

缺点：

1. 数据未及时写盘而带来的风险
2. 额外的数据拷贝过程，大量数据传输时影响性能

Reference

[1] unix 操作系统的设计

原文链接：unix 中数据缓冲区高速缓冲的设计