存储设备

- 寄存器
- 高速缓冲区
- 内存
- 磁盘

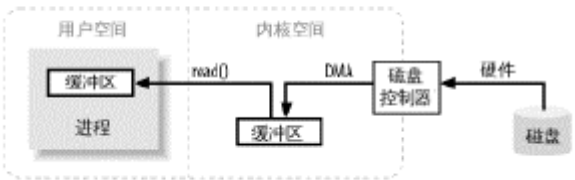
i/o概念

缓冲区操作

所谓"输入 / 输出"讲的无非就是把数据移进或移出缓冲区。

所有 I/O 都直接或间接通过内核空间，它执行一个系统调用（有时称为陷阱）将控制权移交给内核：硬件通常不能直接与用户空间交互（没有将硬件抽象为接口）。其次，像磁盘这样基于块存储的硬件设备操作的是固定大小的数据块，而用户进程请求的可能是任意大小的或非对齐的数据块。在数据往来于用户空间与存储设备的过程中，内核负责数据的分解、再组合工作，因此充当着中间人的角色。

I/O 缓冲区操作简图



内核空间与用户空间

Intel 的 CPU 将特权等级分为 4 个级别：Ring0~Ring3。linux只是用了Ring0，Ring3，Ring3为用户态。

ring 3无法运行多条指令并写入多个寄存器，最值得注意的是：

- 无法改变自己的戒指！否则，它可以将自己设置为响铃0并且响铃将是无用的。

  换句话说，无法修改当前确定当前响铃的当前segment descriptor。
- 无法修改页面表：How does x86 paging work?

  换句话说，无法修改CR3寄存器，并且分页本身会阻止修改页表。

  这可以防止一个进程看到其他进程的内存，以确保安全/易于编程。
- 无法注册中断处理程序。这些是通过写入内存位置来配置的，这也可以通过分页来防止。

  处理程序在环0中运行，会破坏安全模型。

  换句话说，不能使用LGDT和LIDT指令。
- 无法执行`in`和`out`等IO指令，因此可以进行任意硬件访问。

  否则，例如，如果任何程序可以直接从磁盘读取，则文件权限将毫无用处。

更确切地说，感谢Michael Petch：操作系统实际上可以在环3上允许IO指令，这实际上是由Task state segment控制的。

如果戒指3首先没有这样做，那么戒指3是否允许这样做是不可能的。

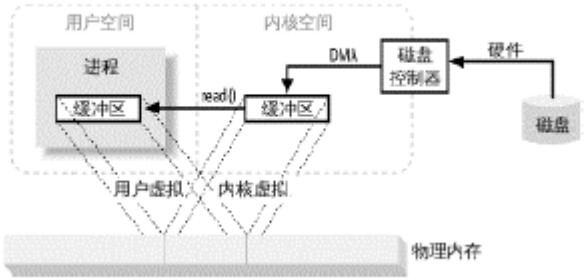
Linux总是不允许它。另见：Why doesn't Linux use the hardware context switch via the TSS?

虚拟内存

这样做好处颇多，总结起来可分为两大类：

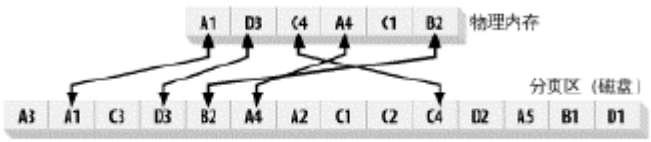
1. 一个以上的虚拟地址可指向同一个物理内存地址。
2. 虚拟内存空间可大于实际可用的硬件内存。 • 面向文件的 I/O 和流 I/O • 多工 I/O（就绪性选择）
3. 把内核空间地址与用户空间的虚拟地址映射到同一个物理地址，这样，DMA 硬件（只能访问物理内存地址）就可以填充对内核与用户空间进程同时可见的缓冲区，但前提条件是，内核与用户缓冲区必须使用相同的页对齐，缓冲区的大小还必须是磁盘控制器块大小（通常为 512 字节磁盘扇区）的倍数。操作系统把内存地址空间划分为页，即固定大小的字节组。

内存空间多重映射



分页技术

用于分页区高速缓存的物理内存



把内存页大小设定为磁盘块大小的倍数，这样内核就可直接向磁盘控制硬件发布命令，把内存页写入磁盘，在需要时再重新装入。结果是，所有磁盘 I/O 都在页层面完成。对于采用分页技术的现代操作系统而言，这也是数据在磁盘与物理内存之间往来的唯一方式。 现代 CPU 包含一个称为内存管理单元（MMU）的子系统，逻辑上位于 CPU 与物理内存之间。该设备包含虚拟地址向物理内存地址转换时所需映射信息。当 CPU 引用某内存地址时，MMU负责确定该地址所在页（往往通过对地址值进行移位或屏蔽位操作实现），并将虚拟页号转换为物理页号（这一步由硬件完成，速度极快）。如果当前不存在与该虚拟页形成有效映射的物理内存页，MMU 会向 CPU 提交一个页错误。

页错误随即产生一个陷阱（类似于系统调用），把控制权移交给内核，附带导致错误的虚拟地址信息，然后内核采取步骤验证页的有效性。内核会安排页面调入操作，把缺失的页内容读回物理内存。这往往导致别的页被移出物理内存，好给新来的页让地方。在这种情况下，如果待移出的页已经被碰过了（自创建或上次页面调入以来，内容已发生改变），还必须首先执行页面调出，把页 内容拷贝到磁盘上的分页区。

文件I/O

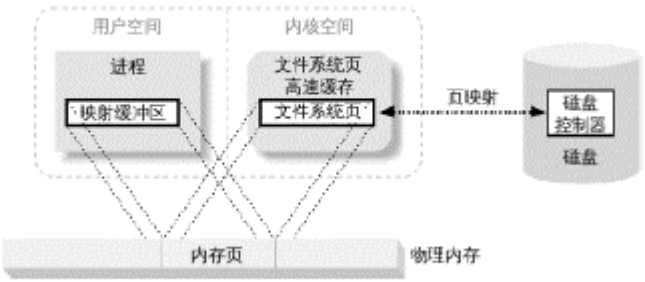
文件 I/O 属文件系统范畴，文件系统与磁盘迥然不同。磁盘把数据存在扇区上，通常一个扇区512 字节。磁盘属硬件设备，对何谓文件一无所知，它只是提供了一系列数据存取窗口。在这点上，磁盘扇区与内存页颇有相似之处：都是统一大小，都可作为大的数组被访问。

文件系统是更高层次的抽象，是安排、解释磁盘（或其他随机存取块设备）数据的一种独特方式。您所写代码几乎无一例外地要与文件系统打交道，而不是直接与磁盘打交道。是文件系统定义了文件名、路径、文件、文件属性等抽象概念。

文件系统把一连串大小一致的数据块组织到一起。有些块存储元信息，如空闲块、目录、索引等的映射，有些包含文件数据。单个文件的元信息描述了哪些块包含文件数据、数据在哪里结束、 最后一次更新是什么时候，等等。

内存映射文件

用户内存到文件系统页的映射



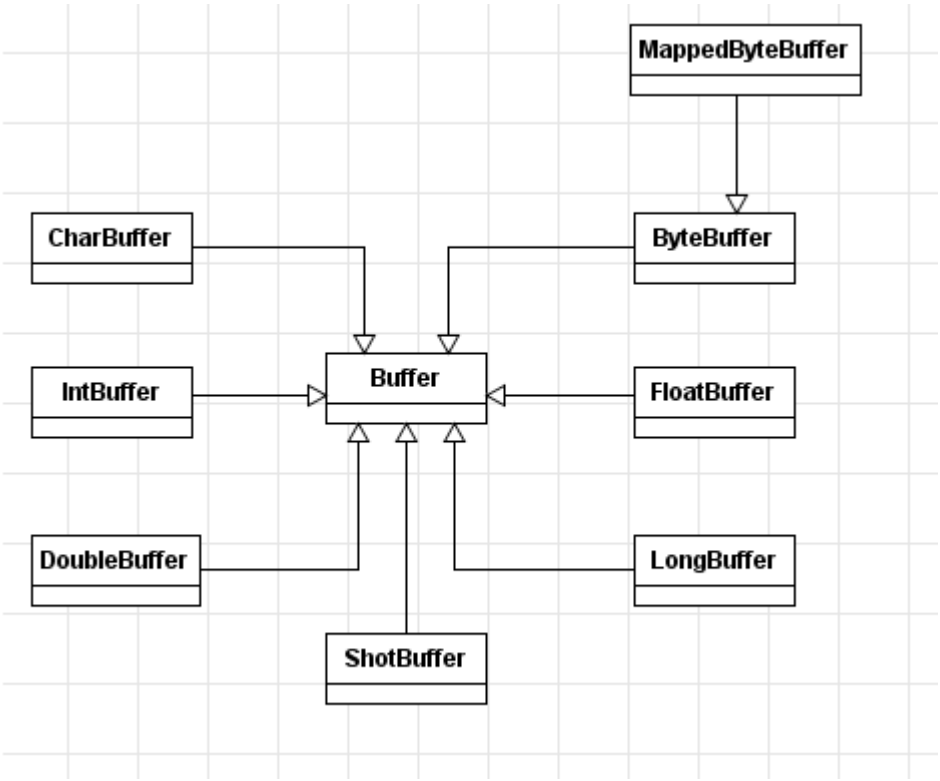
在处理大量数据时，如果数据缓冲区是按页齐的，且大小是内建页大小的倍数，那么，对大多数操作系统而言，其处理效率会大幅提升。

文件锁定

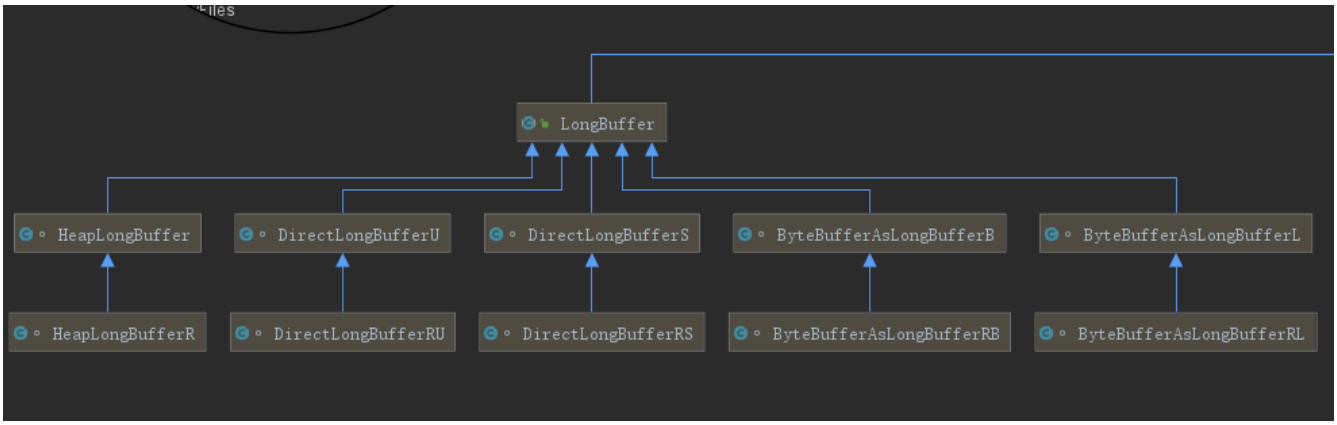
“文件锁定”从字面上看有锁定整个文件的意思（通常的确是那样），但锁定往往可以发生在更为细微的层面，锁定区域往往可以细致到单个字节。锁定与特定文件相关，开始于文件的某个特定 字节地址，包含特定数量的连续字节。这对于协调多个进程互不影响地访问文件不同区域，是至关重要的。 文件锁定有两种方式：共享的和独占的。多个共享锁可同时对同一文件区域发生作用；独占锁则不同，它要求相关区域不能有其他锁定在起作用。

缓冲区

家谱图



每个都根据系统和使用场景进一步细分



属性

0 <= mark <= position <= limit <= capacity

属 性	作 用
capacity	容量，指缓冲区能够容纳的数据元素的最大数量，这一容量在缓冲区创建时被设定，并且永远不能被改变
limit	上界，指缓冲区的第一个不能被读或写的元素，或者说是，缓冲区中现存元素的计数
position	位置，指下一个要被读或写的元素的索引，位置会自动由相应的get()和put()函数更新
mark	标记，指一个备忘位置，调用mark()来设定mark=position，调用reset()来设定postion=mark，标记未设定前是未定义的

链式调用

存取

可以是相对的或者是绝对的

get

put

翻转

flip:The limit is set to the current position and then the position is set to zero. If the mark is defined then it is discarded.

释放

clear()函数将缓冲区重置为空状态。它并不改变缓冲区中的任何数据元素，而是仅仅将上界设为容量的值，并把位置设回0

压缩

compact

比较

缓冲区的比较即equals方法，缓冲区的比较并不像我们想像得这么简单，两个缓冲区里面的元素一样就是相等，两个缓冲区相等必须满足以下三个条件：

1、两个对象类型相同，包含不同数据类型的buffer永远不会像等，而且buffer绝不会等于非buffer对象。

2、两个对象都剩余相同数量的元素，Buffer的容量不需要相同，而且缓冲区中剩余数据的索引也不必相同。但每个缓冲区中剩余元素的数目（从position到limit）必须相同。

3、在每个缓冲区中应被get()函数返回的剩余数据元素序列必须一致。

批量移动

```
public abstract class CharBuffer
    extends Buffer
    implements Comparable<CharBuffer>, Appendable, CharSequence, Readable
{
    ...
    public CharBuffer get(char[] dst){...}
    public CharBuffer get(char[] dst, int offset, int length){...}
    public final CharBuffer put(char[] src){...}
    public CharBuffer put(char[] src, int offset, int length){...}
    public CharBuffer put(CharBuffer src){...}
    public final CharBuffer put(String src){...}
    public CharBuffer put(String src, int start, int end){...}
    ...
}
```

字节缓冲区

字节缓冲区和其他缓冲区类型最明显的不同在于，它们可能成为通道所执行I/O的源头或目标，如果对NIO有了解的朋友们一定知道，**通道只接收ByteBuffer作为参数**(操作系统是字节序列)。

在JVM中，字节数组可能不会在内存中连续存储，或者无用存储单元收集可能随时对其进行移动。在Java中，数组是对象，而数据存储在对象中的方式在不同的JVM实现中各有不同。出于这一原因，引入了直接缓冲区的概念。直接缓冲区被用于与通道和固有I/O线程交互，它们通过使用固有代码来告知操作系统直接释放或填充内存区域，对用于通道直接或原始存取的内存区域中的字节元素的存储尽了最大的努力。

通常非直接缓冲不可能成为一个本地I/O操作的目标，如果开发者向一个通道中传递一个非直接ByteBuffer对象用于写入，通道可能会在每次调用中隐含地进行下面的操作：

1、创建一个临时的直接ByteBuffer对象

2、将非直接缓冲区的内容复制到临时缓冲中

3、使用临时缓冲区执行低层次I/O操作

4、临时缓冲区对象离开作用域，并最终成为被回收的无用数据

这可能导致缓冲区在每个I/O上复制并产生大量对象，而这种事都是我们极力避免的。

直接缓冲区使用的内存是通过调用本地操作系统方面的代码分配的，绕过了标准JVM堆栈。建立和销毁直接缓冲区会明显比具有堆栈的缓冲区更加破费。

建议将直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。一般情况下，最好仅在直接缓冲区能在程序性能方面带来明显好处时 分配它们。

直接字节缓冲区还可以通过 FileChannel 的 map() 方法 将文件区域直接映射到内存中来创建。该方法返回 MappedByteBuffer 。 Java 平台的实现有助于通过 JNI 从本机代码创建直接字节缓冲区。如果以上这些缓冲区中的某个缓冲区实例指的是不可访问的内存区域，则试图访问该区域不会更改该缓冲区的内容，并且将会在访问期间或稍后的某个时间导致抛出不确定的异常。

字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 isDirect() 方法来确定。提供此方法是为了能够在性能关键型代码中执行显式缓冲区管理

直接缓冲区也存在着一些缺点：

（1）不安全；

（2）消耗更多，因为它不是在JVM中直接开辟空间。这部分内存的回收只能依赖于垃圾回收机制，垃圾什么时候回收不受我们控制；

（3）数据写入物理内存缓冲区中，程序就失去了对这些数据的管理，即什么时候这些数据被最终写入从磁盘只能由操作系统来决定，应用程序无法再干涉。

创建缓冲区

```
public abstract class CharBuffer
      extends Buffer implements CharSequence, Comparable
{
      // This is a partial API listing

      public static CharBuffer allocate (int capacity)

      public static CharBuffer wrap (char [] array)
      public static CharBuffer wrap (char [] array, int offset,
  int length)

      public final boolean hasArray( )
      public final char [] array( )
      public final int arrayOffset( )
}
```

如果您想提供您自己的数组用做缓冲区的备份存储器，请调用wrap()函数：

```
char [] myArray = new char [100]; CharBuffer charbuffer = CharBuffer.wrap (myArray);//这段代码构造了一个新的缓冲区对象，但数据元素会存在于数组中。这意味着通过调用put()函数造成的对缓冲区的改动会直接影响这个数组，而且对这个数组的任何改动也会对这个缓冲区对象可见.
CharBuffer charbuffer = CharBuffer.wrap (myArray, 12, 42);//这个函数并不像您可能认为的那样，创建了一个只占用了一个数组子集的缓冲区。这个缓冲区可以存取这个数组的全部范围；offset和length参数只是设置了初始的状态。
```

复制缓冲区

Duplicate()函数创建了一个与原始缓冲区相似的新缓冲区。两个缓冲区共享数据元素，拥有同样的容量，但每个缓冲区拥有各自的位置，上界和标记属性。对一个缓冲区内的数据元素所做的改变会反映在另外一个缓冲区上。

您可以使用asReadOnlyBuffer()函数来生成一个只读的缓冲区视图。这与duplicate()相同，除了这个新的缓冲区不允许使用put()，并且其isReadOnly()函数将会返回true。对这一只读缓冲区的put()函数的调用尝试会导致抛出ReadOnlyBufferException异常。