《openEuler内核编程》

课程讲稿

第八章 第3讲

块设备

软件所制

第八章 第3讲 块设备

**学时：**2学时

**教学目的：**系统学习块设备与块设备驱动，深入了解块设备与块设备驱动原理。

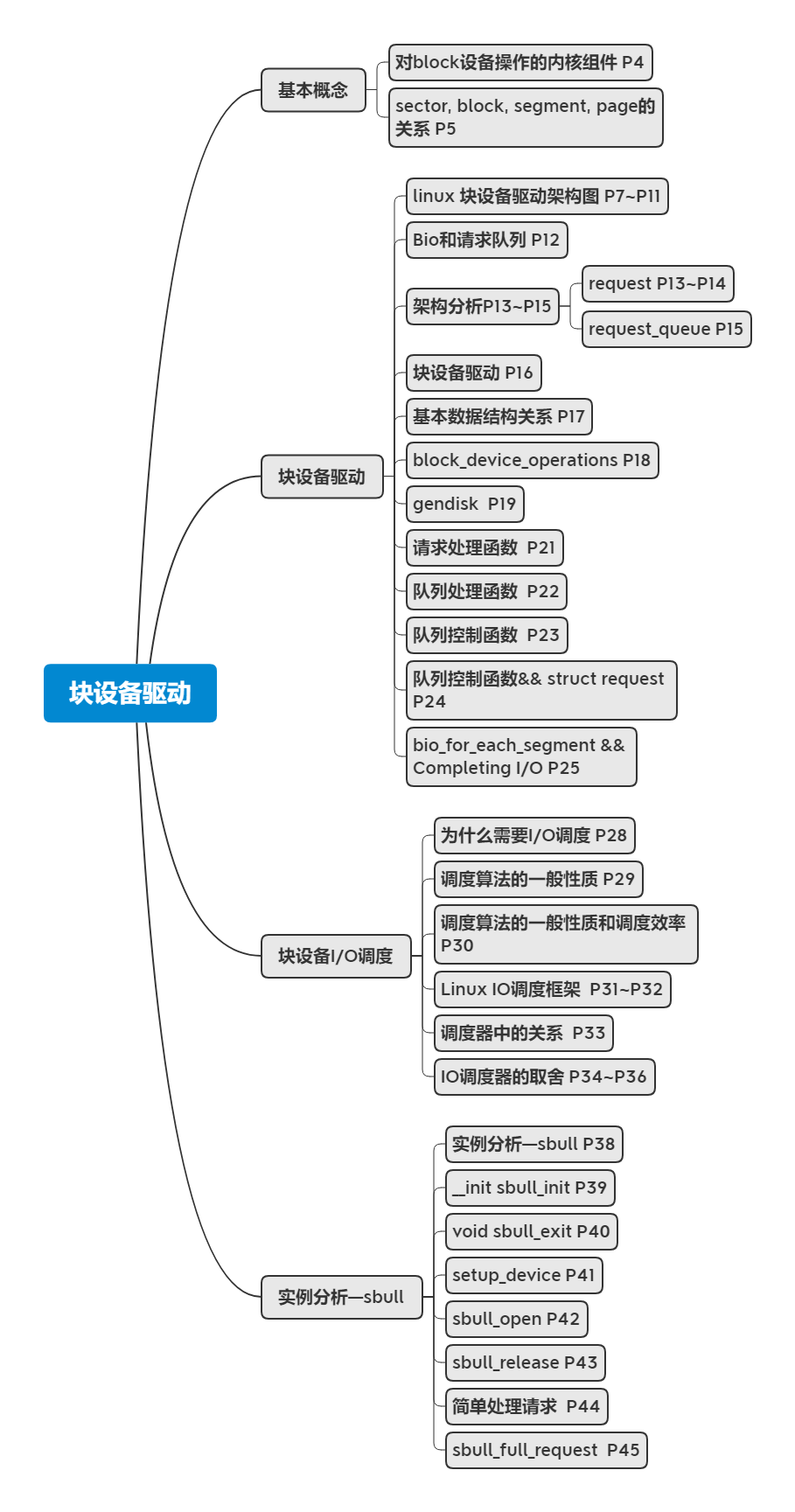
**课程时间线：**



**课外参考读物：**

《LINUX设备驱动程序（第三版）》

**知识框图：**

****

**PPT讲稿：**

1. 本节我们来看本章的第三节内容，块存储设备

2. 这里是本章的结构，在上一节课我们介绍了Linux设备驱动及其分类后，介绍了第一个大分类字符设备驱动程序，然后我们轮到介绍块设备驱动的讲解，但是在讲解块设备驱动之前，我们先花一节课的时间来讲一下块存储设备的知识，为之后驱动的讲解打好基础。

3. 这里是本节的内容，首先我们先来介绍磁盘。

4. 磁盘大家一定都很熟悉，但是我们这里介绍的不是普通的磁盘，而是要深入的了解磁盘运行的原理，那么我们这里首先要介绍的就是磁盘控制器了：

磁盘控制器就是右边我们看到，Disk上边的一整个框框就是磁盘控制器，它包含了三个部分：Interface、DRAM cache和Control logic。

首先我们先来介绍整个控制器的原理：硬盘控制器即磁盘驱动器适配器。是计算机与磁盘驱动器的接口设备。它接收并解释计算机来的命令，向磁盘驱动器发出各种控制信号。检测磁盘驱动器状态，按照规定的磁盘数据格式，把数据写入磁盘和从磁盘读出数据。磁盘控制器类型很多，但它的基本组成和工作原理大体上是相同的，它主要由与计算机系统总线相连的控制逻辑电路，微处理器，完成读出数据分离和写入数据补偿的读写数据解码和编码电路，数据检错和纠错电路，根据计算机发来的命令对数据传递，串并转换以及格式化等进行控制的逻辑电路，存放磁盘基本输入输出程序的只读存储器和用以数据交换的缓冲区等部分组成。

然后我们来分别介绍一下磁盘控制器的三个部分：首先第一部分是对外接口Interface，它承接了与磁盘外交互的工作。在对外接口这里有很多种标准：第一类是IDE类型，IDE控制器经常制作在主板中，最多能够支持4个硬盘，每秒传输数据最多可达66Megabytes(MB)，但是许多IDE控制器仅能达到33Megabytes。IDE控制器有几种变体，常见的就是ATA：AdvancedTechnologyAttachment（附加高级技术），ATA是IBM发明的装在硬盘里的第二个控制器，能够大大加快数据的传输速度。ATA又可以分为以下几种规定：

ATA-1：原始的硬盘控制器，与IDE是同一个概念。

ATA-2：通常看作快速ATA(FASTATA)或增强的IDE(EIDE)。EIDE最先由WesternDigital推出，实用新的BIOS，容许管理504MB容量的硬盘。Quantum和Seagate同时推出FastATA，能够更快的传输数据，支持大于504MB的容量，可以联接4个设备，包括CD-ROM和磁带机等。

ATA-3：ATA-3紧跟着ATA-2推出，但是性能没有多少改善，通常ATA-3等同于EIDE或FastATA。

ATA-4：ATA-4又称作UltraATA或UltraDMA，包含ATAPI4规定。ATAPI是"ATAttachmentwithPacketInterface"的简称，允许CD-ROM和磁带机与ATA硬盘共享ATA总线，ATA-4是今天电脑使用最多的控制器。

ATA-5：ATA-5是正在开发中的规定，能够支持超过66MB速度的传输，同时通过改善内部算法提高了可靠性。要求使用新的UDMA80电缆，新的主板将支持ATA-5。

第二类是SCSI类型，SCSI是非常先进的硬盘控制器，但是通常它的价格高于IDE因此难于普及。SCSI能够支持多件硬盘和CD-ROM、扫描仪等设备，可以用菊花链的方式联接多达30件设备和外围，最适合需要巨大容量的录音棚采用。SCSI控制器传统上就比IDE快，更能够达到每秒160MB的速度，正在开发的新控制器甚至可以达到每秒320MB。SCSI也分为如下几种规定：

SCSI-1：SCSI1是1986年的原始规定，第一个SCSI标准。最多允许联接7件设备，最大传输速度每秒5MB，在当时具有划时代的意义。

SCSI-2：SCSI2又称为FastSCSI（快速SCSI），在SCSI1的基础上做了诸多改善,但是直到1994年之前没有普遍应用。SCSI-2精简了一些冗余的选项，增加了可靠性，把传输速度提高到每秒10MB。

SCSI-3：又称为SCSI160，是极快的SCSI标准。使用32bitPCI槽的控制器最高传输速度为每秒80MB，使用64bitPCI槽的控制器最高传输速度为每秒160MB，一个控制器可以联接14件设备。

SCSI3已经慢慢的降低了价格，它将是24bit/96KHz录音硬盘的最佳选择。

然后接下来是缓存，也就是中间的DRAM cache，用于缓冲数据，提高数据的读取速度。

最下面是Control logic也就是控制逻辑，主要控制了磁盘的读写操作、缓存替换规则以及完成失效检测和恢复。

5. 在这里我们详细介绍一下磁盘缓存的内容。

磁盘缓存是为了减少CPU透过I/O读取磁盘机的次数，提升磁盘I/O的效率，用一块内存来储存存取较频繁的磁盘内容；因为内存的存取是电子动作，而磁盘的存取是I/O动作，感觉上磁盘I/O变得较为快速。

通过磁盘缓存，一方面起到了提速的效果，另一方面亦可以减少实际的磁盘操作，有效的保护磁盘免于重复的读写操作而导致的损坏。

使用的块替换策略是LRU，也就是最近最少使用原则，那么从这个策略我们就可以脑补出来：如果访问是具有局部性的，比如说我最近100次访问都是在这个附件进行读取，那么这个cache的加入是可以极大的提高读的效率。

但是从另一方面来看，缺点就是如果你使用了cache，然后我执行了写的操作将这部分数据进行了修改，那么你是需要一个额外的机制来保证cache中的数据是最新版本的。

磁盘缓存不仅是用于读，也是用于写的，可以说磁盘缓存是磁盘与IO接口之间的桥梁。当进行写操作时，将下载到的数据先保存于系统为软件分配的内存空间中（这个内存空间被称之为“内存池”），当保存到内存池中的数据达到一个程度时，再将数据保存到硬盘中。

6. 磁臂和磁头。有一个细节是磁臂上其实是有少量的DRAM用于缓存的，不过这个可以当作黑盒处理，同样是避免IO频繁的操作，我们在这里提一嘴带过即可。

磁头就是直接与磁盘接触的部分，它装在一个促动器上，而促动器是可以通过磁臂进行移动的。

当进行读写操作之时，希望读写某磁道的某扇区。前提条件是磁盘是在不断旋转的，我们无法控制旋转到哪，那么想要读取数据，第一步就是我们要定位磁道到目标磁道：这一点是通过移动磁臂即可实现的。

当磁道定位完成后我们就可以等待扇区移动了，当目标扇区旋转到磁头下方时，进行读写操作。

7. 这里是磁盘的构成，如果对于刚刚的操作没有明白我们可以看着这里模拟一遍。

磁盘表面的同心圆，位沿着每条磁道顺序排列。实际上是磁盘表面有无数的同心圆，然后这些同心圆的每个间隙就是一条磁道。

然后我们可以看到其实是有很多盘面是在同一起叠放的，那么所有盘片上的同一条磁道，就是同一个柱面。每个zone有1000-5000个柱面，其中一个为备用柱面（spare）。换句话说就是每一个单片的磁道，都有这么多同心圆的间隙，然后大家叠放在一起的所有单片，叠放在一起是柱面。

8. 这里我们进入软件操作系统的层面来看磁盘扇区。当面对一个新电脑时，是需要扇区的创建的，这个操作可以在操作系统安装前，也可以在操作系统完全完成后。当需要进行扇区创建时，首先需要磁盘的格式化，然后逻辑块地址映射到物理块地址。

在扇区之中，每一个扇区在头部保存了一些额外信息，比如：ID，损坏标志位，等等，那么实际上用于存取数据的空间是被占用了一些的，实际的大小典型数据是512B。在扇区的尾部则是ECC检验码。

当发现有错误产生时，如果发现扇区产生了错误，首先用ECC纠错，如果纠错不能够成功完成的话，则用备用扇区代替。同时也意味着这片扇区是有问题的，坏扇区标注头部后不再使用。

9. 开始介绍磁盘寻址方式，我们如何在磁盘中定位。 很久以前， 硬盘的容量还非常小的时候，人们采用与软盘类似的结构生产硬盘。也就是硬盘盘片的每一条磁道都具有相同的扇区数。由此产生了所谓的CSH 3D参数 (Disk Geometry)。即磁头数(Heads)，柱面数(Cylinders)，扇区数(Sectors)，以及相应CHS寻址方式。CHS寻址模式将硬盘划分为磁头（Heads）、柱面(Cylinder)、扇区(Sector)。

磁头(Heads)：每张磁片的正反两面各有一个磁头，一个磁头对应一张磁片的一个面。因此，用第几磁 头就可以表示数据在哪个磁面。

柱面(Cylinder)：所有磁片中半径相同的同心磁道构成“柱面"，意思是这一系列的磁道垂直叠在一起，就形成一个柱面的形状。简单地理解，柱面数=磁道数。

扇区(Sector)：将磁道划分为若干个小的区段，就是扇区。虽然很小，但实际是一个扇子的形状，故称为扇区。每个扇区的容量为512字节。

随着科技大发展，磁盘容量大幅提升。远远超过了8GB寻址范围，如何对8GB之外空间进行寻址？历史上曾经CHS从24位扩展到多28位，实现寻址128GB，但是面对现在磁盘2TB容量还是无能为力，下面我们请出最终解决方案LBA。

10. 逻辑块寻址模式。在 LBA 模式下，我们知道硬盘上的一个数据区域由它所在的磁头、柱面（也就是磁道）和扇区所唯一确定。早期系统就是直接使用磁头柱面和扇区来对硬盘进行寻址（这称 为CHS寻址），这需要分别存储每个区域的三个参数（这称为3D参数），使用时再分别读取三个参数，然后在送到磁盘控制器去执行。由于系统用8b来存储磁 头地址，用10b来存储柱面地址，用6b来存储扇区地址，而一个扇区共有512B，这样使用CHS寻址一块硬盘最大容量为256 \* 1024 \* 63 \* 512B = 8064 MB(1MB = 1048576B)（若按1MB=1000000B来算就是8.4GB）。随着硬盘技术的进步，硬盘容量越来越大，CHS模式无法管理超过8064 MB的硬盘，因此工程师们发明了更加简便的LBA寻址方式。在LBA地址中，地址不再表示实际硬盘的实际物理地址（柱面、磁头和扇区）。LBA编址方式将 CHS这种三维寻址方式转变为一维的线性寻址，它把硬盘所有的物理扇区的C/H/S编号通过一定的规则转变为一线性的编号，系统效率得到大大提高，避免了 烦琐的磁头/柱面/扇区的寻址方式。在访问硬盘时，由硬盘控制器再这种逻辑地址转换为实际硬盘的物理地址。

不管CHS（寻址方式）也好，还是LBA（寻址方式）也好。磁盘存储寻址都需要通过 cylinder、head、 sector这三个变量来实现；CHS、LBA都是一个数字，CHS按照固定格式把24个bt位分成 cylinder、head、 sector：LBA则需要通过求模运算得出cylinder、head、sector。即由chs值可以直接获得cylinder、head、sector这三个变量的值，而由LBA值则需要通过运算间接得出 cylinder、head、 sector这三个变量的值。

11. 第一块磁盘是1956年制作出来的，我们可以看到最先制作出来的磁盘存储内容很少，与我们现在的磁盘相差很远。

12. 这里是一些磁盘的外形参数，根据不同的需求，有不同的外形大小和不同的存储大小。

13. 摩尔定律

摩尔定律是由英特尔（Intel）创始人之一戈登·摩尔（Gordon Moore）提出来的。其内容为：当价格不变时，集成电路上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍。换言之，每一美元所能买到的电脑性能，将每隔18-24个月翻一倍以上。这一定律揭示了信息技术进步的速度。 尽管这种趋势已经持续了超过半个世纪，摩尔定律仍应该被认为是观测或推测，而不是一个物理或自然法。

实际上摩尔定律也并非“一成不变”，1965年刚刚提出来时说的是12个月翻一倍，而1975年则重新调整为18-24月翻一倍。

所以，正因为每种技术的发展都会遇到天花板，物理上的极限也就是摩尔定律的天花板了，随着晶体管尺寸的不断缩小，量子隧穿效应将会很难解决，（估计）等到3nm工艺量产后就无限接近这个极限了。而目前在7nm工艺量产下，摩尔定律已经几乎失效。

14. 我们来讨论一下磁盘的性能问题。如果我们想要读取一份数据，那么我们需要的总时间是由多方面组成的：

我们收到了一份读数据的请求，那么第一步就是需要进行寻到，把磁头移动到正确的磁道上：典型的寻道时间是3.5~9.5ms。

接下来我们需要等待目标扇区旋转到磁头下方，典型的时间是：4~8ms （对应的磁盘旋转速度上线限分别是15000RPM和7200RPM）

然后我们就可以开始读取数据了：典型的数据传输带宽是70~250 MB/sec。

那么这里我们来看一个例子，通过上述的我们讲解，同学们能否算出来有效的带宽是多少？

15. 那么我们现在来考虑一个问题，经过之前我们的分析，我们知道无论你传输多少数据，你的寻道时间和等待旋转的时间都是固定的。所以说我们一次传输的数据如果很少的话，那么实际上的效率会非常低。那么我们来计算一下需要一次传输多少数据才能达到磁盘带宽的90%？

（计算）

对于小粒度的访问，时间主要花在寻道时间和旋转时间上。磁盘的传输带宽被浪费，那么我们的应对措施就是通过调度算法来减少寻道开销

16. 接下来我们来介绍本节课的第二部分内容，磁盘调度的算法。

17. 第一种就是最简单的FIFO算法，first in first out，先入先出算法，按请求顺序进行服务。好处是公平，而且服务顺序是应用预期的。但是坏处是由于请求到来的随机性，经常会长距离寻道，效率不高，而且可能会有极端的情况导致横扫整个磁道的坏处出现。

18. 思考到FIFO的坏处，那么我们很容易想到这个算法逻辑：每次我们选择，都选择磁头移动距离最短的请求。这样做的好处是每次的寻道时间都是最短的。但是坏处是会产生饥饿。就如示例之中，183其实是很早就到达的请求，但是由于183与当前的访问距离很远，直到最后才访问到它，可能此时这个访问已经失效或者已经来不及了，也就是饥饿现象。

19. 为了解决饥饿现象，那么产生了Scan算法。磁头按一个方向到另一端，再折回，按反方向回到这端，不断往返。只服务当前移动方向上寻道距离最近的请求。然后如果磁盘移动方向上没有请求，就折回。好处是会消除饥饿，即使是最坏情况，在一次往返的时间内也一定会处理，有时间上限。坏处就是由于方向一般情况下是不变的，所以如果在当前方向的反方向出现请求，需要一段时间才会处理。

20. 那么为了处理scan算法的往返时间处理问题，就有了C-SCAN算法：与SCAN算法几乎相同，唯一的不同之处就是Cscan算法在折回时是不服务请求的，直接返回到头部开始继续处理。它将最迟处理上限由2倍的总距离变为一个总距离（从例子中来看，是从400减少到200）

21. 现在请同学们思考一下，在这些算法之中，各自的优劣是否能明显的想起来，你最喜欢哪种算法？互相之间有没有哪些算法是明显优于或者劣于某些算法的情况存在？

22. 接下来我们介绍下一个知识点：磁盘阵列

23. 磁盘阵列（Redundant Arrays of Independent Disks，RAID），有“独立磁盘构成的具有冗余能力的阵列”之意。磁盘阵列是由很多块独立的磁盘，组合成一个容量巨大的磁盘组，利用个别磁盘提供数据所产生加成效果提升整个磁盘系统效能。利用这项技术，将数据切割成许多区段，分别存放在各个硬盘上。磁盘阵列还能利用同位检查（Parity Check）的观念，在数组中任意一个硬盘故障时，仍可读出数据，在数据重构时，将数据经计算后重新置入新硬盘中。

主要思想是计算块的校验（ XOR），将校验块保存在一块磁盘上，当一个磁盘坏掉时，通过校验块和其它磁盘的数据恢复坏盘上的数据。

好处是由于这种额外的效果，可靠性自然是提高了。坏处是成本上就会提高，而且控制器会变得复杂。

24. RAID分为几个级别，分别如下：

1、RAID 0

RAID 0是组建磁盘阵列中最简bai单的一种形式，只需要2块以上的du硬盘即可，成本zhi低，可以提高整个磁盘的性能和吞吐量。RAID 0没有提供冗余或错误修复能力，但实现成本是最低的。

2、RAID 1

RAID 1主要是通过二次读写实现磁盘镜像，所以磁盘控制器的负载也相当大，尤其是在需要频繁写入数据的环境中。为了避免出现性能瓶颈，使用多个磁盘控制器就显得很有必要。

3、RAID2：带海明码校验

RAID 2 同RAID 3类似， 两者都是将数据条块化分布于不同的硬盘上， 条块单位为位或字节。然而RAID 2 使用一定的编码技术来提供错误检查及恢复。

25.

4、RAID3：带奇偶校验码的并行传送

RAID3访问数据时一次处理一个带区，这样可以提高读取和写入速度。校验码在写入数据时产生并保存在另一个磁盘上。

5、RAID4：带奇偶校验码的独立磁盘结构

RAID4和RAID3很象，不同的是，它对数据的访问是按数据块进行的，也就是按磁盘进行的，每次是一个盘。

6、RAID5：分布式奇偶校验的独立磁盘结构

RAID5的奇偶校验码存在于所有磁盘上，其中的p0代表第0带区的奇偶校验值，其它的意思也相同。RAID5的读出效率很高，写入效率一般，块式的集体访问效率不错。

26.

RAID6：带有两种分布存储的奇偶校验码的独立磁盘结构

RAID6技术是在RAID 5基础上，为了进一步加强数据保护而设计的一种RAID方式，实际上是一种扩展RAID 5等级。与RAID 5的不同之处于除了每个硬盘上都有同级数据XOR校验区外，还有一个针对每个数据块的XOR校验区。当然，当前盘数据块的校验数据不可能存在当前盘而是交错存储的，具体形式见图。这样一来，等于每个数据块有了两个校验保护屏障（一个分层校验，一个是总体校验），因此RAID 6的数据冗余性能相当好。但是，由于增加了一个校验，所以写入的效率较RAID 5还差，而且控制系统的设计也更为复杂，第二块的校验区也减少了有效存储空间。

27.

RAID7：优化的高速数据传送磁盘结构

RAID 7等级是至今为止，理论上性能最高的RAID模式，因为它从组建方式上就已经和以往的方式有了重大的不同。基本成形式见图，你会发现在，以往一个硬盘是一个组成阵列的“柱子”，而在RAID 7中，多个硬盘组成一个“柱子”，它们都有各自的通道，也正因为如此，你可以把这个图分解成一个个硬盘连接在主通道上，只是比以前的等级更为细分了。这样做的好处就是在读/写某一区域的数据时，可以迅速定位，而不会因为以往因单个硬盘的限制同一时间只能访问该数据区的一部分，在RAID 7中，以前的单个硬盘相当于分割成多个独立的硬盘，有自己的读写通道，效率也就不言自明了。

28. 接下来介绍闪存SSD的发展

29. 闪存是一种电子式可清除程序化只读存储器的形式，允许在操作中被多次擦或写的存储器。这种科技主要用于一般性数据存储，以及在计算机与其他数字产品间交换传输数据，如储存卡与U盘。闪存是一种特殊的、以宏块抹写的EPROM。早期的闪存进行一次抹除，就会清除掉整颗芯片上的数据。

闪存是一种非易失性存储器，即断电数据也不会丢失。因为闪存不像RAM（随机存取存储器）一样以字节为单位改写数据，因此不能取代RAM。

闪存卡（Flash Card）是利用闪存（Flash Memory）技术达到存储电子信息的存储器，一般应用在数码相机，掌上电脑，MP3等小型数码产品中作为存储介质，所以样子小巧，有如一张卡片，所以称之为闪存卡。根据不同的生产厂商和不同的应用，闪存卡大概有SmartMedia（SM卡）、Compact Flash（CF卡）、MultiMediaCard（MMC卡）、Secure Digital（SD卡）、Memory Stick（记忆棒）、XD-Picture Card（XD卡）和微硬盘（MICRODRIVE）这些闪存卡虽然外观、规格不同，但是技术原理都是相同的。

30. 前面说过，闪存的读写单位为页，而页的大小一般为4KB或8KB，但我们的操作系统读写数据是按HDD的扇区尺寸进行的（512Byte（字节）），更麻烦的是闪存擦除以块作单位，而且未擦除就无法写入，这导致操作系统现在使用的文件系统根本无法管理SSD，需要更换更先进、复杂的文件去解决这个问题，但这样就会加重操作系统的负担。

而为了不加重操作系统的负担，SSD采用软件的方式把闪存的操作虚拟成磁盘的独立扇区操作，这就是FTL。因FTL存在于文件系统和物理介质（闪存）之间，操作系统只需跟原来一样操作LBA即可，而LBA到PBA的所有转换工作，就全交由FTL负责。

我们可以从图中看到FTL负责了中间的交互过程。

31. 那么FTL的实现有三种方法：页映射、块映射和日志块映射

32. 块映射：

1）划分标准，以闪存的块为映射粒度

2）注意，一个用户逻辑块可以映射到任意一个闪存的物理块，但是映射前后每个页在块中的偏移不变

3）优缺点，映射表只存储块的映射->所需存储映射表的空间小；

　　　　  连续大尺寸的读写性能好，但是小尺寸的写性能很差；

　　　　（即使只写一个逻辑页，也需要把整个物理块数据读上来，然后改变逻辑页的数据再整个块写入->性能差）

4）用户空间被化成一个个逻辑区域，每个区域与闪存块的大小相同

33. 页映射：

1）划分标准，以闪存的页为映射粒度 ，一个逻辑页可以映射到任一物理页中

2）注意，由于闪存页远比闪存块多，因此需要更多的空间来存储映射表

3）优缺点，性能很好，尤其体现在随机写上

4）用户空间被化成一个个逻辑区域，每个区域与闪存页的大小相同

34. 混合映射

1）划分标准，块映射和页映射的结合

2）一个逻辑块映射到任一物理块，但是块中的每个页的偏移并不是固定不变的，块内采用页映射的方式，一个逻辑块的逻辑页可以映射到对应物理块的任一页，因此它所需要的空间以及性能都是介于块映射和页映射之间的。

3）用户空间划分成一个个逻辑区域，逻辑区域和闪存块大小相同，每个逻辑块对应一个闪存块，但逻辑块内部有分成一个个逻辑页，与对应闪存页随即对应。

35. 垃圾回收和磨损均衡

磨损均衡（GC）：

首先我们应当明确，磨损均衡是闪存的一种抹平技术。那么何为抹平技术？要先了解闪存。

闪存是目前比较流行的只读存储器，属于电擦除可编程只读存储器的升级，可以反复擦写，这就是抹平。但是闪存内的区块有抹写次数限制，对于同一个区块进行重复的抹除和写入，会造成区块读取速度变慢，甚至损坏。

磨损均衡名字中的均衡，就意味着要尽量平均使用闪存内的每个存储区快，避免部分给过度使用变成坏区块，而部分区块使用频率过低。其目的最终还是要延长闪存高效使用的时间。

垃圾回收：

当一个物理页的数据需要更新时，固态硬盘先将新数据写入到别的空的物理页中，再将原物理页的逻辑地址重定向，同时把原物理页标为无效。但是随着更新次数的增加，闪存中会出现大量的无效页。垃圾回收也就应运而生。

在了解闪存的垃圾回收之前，我们先了解以下磁类介质的读写问题。磁类介质（磁带或磁盘）在写入新数据时，无论该区块是否有旧数据（旧数据为不再需要的数据），直接进行覆盖即可。但是闪存和磁类介质不同，闪存是电子类介质，在写入新数据时，区块必须是空白的。所以如果要写入的区块存在旧数据，那么必须先对旧数据进行擦除。所以我们可以预见，当写入的数据达到闪存容量后，一定会发生擦除行为。擦除行为，也就被称为“垃圾回收”。

36. 这里是我们这节课的总结：

• 磁盘是复杂的

• 磁盘的密度按摩尔定律增长

• 大块读写才能获得高带宽

• 需要磁盘调度来减少寻道开销

• RAID提高了可靠性和I/O带宽

• 闪存有取代传统磁盘的趋势

• 闪存的不同读写特性: FTL和GC

37. 同学们我们这节课开始进入第八章内容的第四节：块设备驱动。

38. 这里是本章的课时结构，在上一节课的内容中我们介绍了关于块设备的内容，接下来我们开始介绍基于块设备基础上的块设备驱动。

39. 这里是本节的课时内容。分为四个部分，我们首先先来介绍一下块设备驱动相关的一些基本概念

40. 在讲解块设备驱动之前，首先先介绍一些等会讲解我们会用到的概念：

~通用块设备层：是一个用于隐藏硬件细节，提供block设备的抽象视图。它提供了通用的数据结构描述 "disks" 和 "disk partitions"。

~IO调度："I/O scheduler "(I/O调度器)根据内核制定的策略对未决的(pending) I/O 数据传送请求进行排序和调度。提高I/O 调度器的效率也是影响整个系统对块设备上数据管理效率的一个方面。

~块设备：最后，block device 设备驱动程序，完成和硬件的具体交互。

41. 这里是四个概念之间的关系：sector, block, segment, page

Page是最大的概念，它包含了segment和属于自己的block buffer；其中segment又包含了属于自己的一些block buffer，然后无论是page自己的block buffer还是segment的block buffer，每一个block buffer包含了自己的两个sector。这些就是sector, block, segment, page这四个概念之间的关系。

Sector是存放数据的连续区域单位，每一个块是512字节，也就意味着存放数据一定是512的整数倍。其存放数据本身是通过一个sector\_t结构体来定义。

Block是文件系统管理数据的单位，通过存放不同数量的block来进行存储，所以一定是sector的整数倍。其存放sector的数量受限于页page的大小，不能超过page的大小。

42. 介绍完一些基础概念之后我们开始正式介绍块设备驱动。

43. 我们先回忆一下块设备的概念，在上一节课进行了介绍：是一种具有一定结构的随机存取设备，对这种设备的读写是按块进行的，他使用缓冲区来存放暂时的数据，待条件成熟后，从缓存一次性写入设备或者从设备一次性读到缓冲区。

这里是linux 块设备驱动架构图。我们可以从图中看到，在linux系统中，从linux的虚拟文件系统一直到调用硬件的通路是这样的：虚拟文件系统——块设备文件系统（此时是映射层）——通用块层——IO调度层——块设备驱动——硬件。

其中前两层是文件系统，中间三层是内核空间，最后是硬件。

我们可以看到在内核空间和硬件的右边标识了很多的结构体，这些结构体是进行块设备驱动调用的重要部分，我们接下来会对这些标识出来的bio结构体、request结构体、gendisk结构体等进行介绍。

44. 我们现在开始介绍bio结构体。当一个进程被Read时,首先读取cache 中有没有相应的文件，这个cache由一个buffer\_head结构读取。如果没有，文件系统就会利用块设备驱动去读取磁盘扇区的数据。于是read()函数就会初始化一个bio结构体，并提交给通用块层。通常用一个bio结构体来对应一个I/O请求。

这里是bio的内核结构。我们可以细致的来看一下。首先保存的是第一个sector扇区（bio包含sector），然后是bio作为一个IO请求，处理完该请求后需要找到下一个，所以有一个BIO类型的指针来指示，这里大家可以理解为以一个链表的形式来链接。

后面是一些保存状态、命令、读写操作、优先级的信息。后面有一些关于数量和索引的内容。

45. 这里是后面的结构部分。需要注意的是实际vec的列表是在这里bi\_io\_vec保存。

46. bio的核心就是刚刚我们重点指出来得bi\_io\_vec数组，它由bio\_vec组成（也就是说bio由许多bio\_vec组成）。Bio\_vec得内核定义如下。

bio\_vec描述一个特定的片段，片段所在的物理页，块在物理页中的偏移页，整个bio\_io\_vec结构表示一个完整的缓冲区。当一个块被调用内存时，要储存在一个缓冲区，每个缓冲区与一个块对应，所以每一个缓冲区独有一个对应的描述符，该描述符用buffer\_head结构表示

47. 这里是刚刚所说得buffer\_head的内核结构。

48. 这里是一个BIO和请求队列得关系，可以看到在事件得处理过程中bio所起到的作用。

49. 这里是request得结构体。

提交工作由submit\_bio()去完成，通用层在调用相应的设备IO调度器，这个调度器的调度算法，将这个bio合并到已经存在的request中，或者创建一个新的request，并将创建的插入到请求队列中。最后就剩下块设备驱动层来完成后面的所有工作。（Linux系统中，对块设备的IO请求，都会向块设备驱动发出一个请求，在驱动中用request结构体描述）

50. request结构体的后半部分

51. 这里是请求队列的数据结构。可以看到只列出来了中间两个核心函数：request\_fn和make\_request\_fn。

下面这里是我们重点介绍的一些函数：

request\_queue\_t \*blk\_init\_queue(request\_fn\_proc \*rfn, spinlock\_t \*lock)

第一个参数是指向"请求处理函数"的指针，该函数直接和硬盘打交道，用来处理数据在内存和硬盘之间的传输。该函数整体的作用就是为了分配请求队列，并初始化。

typedef void (request\_fn\_proc)(struct reqest\_queue \*q)

该函数作为上述函数（request\_queue\_t \*blk\_init\_queue(request\_fn\_proc \*rfn,spinlock\_t \*lock)）的参数,主要作用就是处理请求队列中的bio，完成数据在内存和硬盘之间的传递。（注意：该函数参数中的bio都是经过i/o调度器的）

typedef int (make\_request\_fn)(struct request\_queue \*q,struct bio \*bio)

该函数是的第一个参数是请求队列，第二个参数是bio,该函数的作用是根据bio生成一个request（所以叫制造请求函数）。

注意：在想不使用I/O调度器的时候，就应该在该函数中实现，对每一传入该函数的bio之间进行处理，完成数据在内存和硬盘的之间的传输，这样就可以不使用"request\_fn\_proc"函数了。（所以可以看出来，如果使用i/o调度器，make\_request\_fn函数是在request\_fn\_proc函数之前执行）

52. 在此我们可以看出来，块设备驱动的一些性质：

块设备驱动与字符设备驱动程序是完全不同的两种驱动。

在文件系统中，是以块为单位进行随机存取的。（随机存取：给定任意地址，读取数据。有别于顺序存取只能按顺序读取数据）

不同的体系结构，对block的大小定义可能不同，比如有些系统定义一个 block 大小是 4096 字节。一般定义是sector的整数倍皆可，一个sector是512字节。

Block 设备一般和磁盘相关。

int register\_blkdev(unsigned int major, const char \*name);相当于是一个入口函数，在/proc/devices 中创建一入口。

如果第一个参数是 0 ，那么内核动态分配主设备号。

53. 接下来我们介绍一下之前我们讲到的数据结构之间的关系：

可以看到block\_device调用block\_device，然后二者同步参与了gendisk结构体的调用参数。然后形成了调用，进入request\_queue队列之中。

所有块设备描述符被插入到全局链表中，有all\_dbdevs指向。

bd\_list：链表级联指针有块设备描述符成员变量。

54. 接下里我们继续介绍其他的结构体。在这里介绍一个设备操作结构体，block\_device\_operations。

在结构体中主要是四个相关操作：open、release、revalidate\_disk和getgeo。

前三个函数通过函数名就可以看出来分别看出来函数的作用：当设备打开和关闭时将调用open和release函数。

revalidate\_disk该函数被调用来响应一个介质改变，它给驱动一个机会来进行必要的工作使得新介质准备好。

而getgeo是用于得到块设备相关参数，如：柱面数、扇区数、开始扇区等。

55. Linux内核中，用gendisk结构体表示一个磁盘设备或分区，块设备驱动程序的设计主要就是围绕gendisk这个数据结构展开的。一个简单的块设备驱动程序框架如下：

在init函数中分配、设置、添加一个gendisk;

设计gendisk结构体fops成员包含的操作函数；

设计gendisk结构体queue(请求队列)成员的请求处理函数。

下面说明gendisk的主要成员及设置方法。

56. 这里是对gendisk的一个总结: struct gendisk 是一动态结构，不能在驱动程序中按照常规方法分配这个结构，只能按照内核提供的函数来分配。它所使用的分配空间和删除空间是通过特殊的函数来进行的。并且在初始化结构之后要向系统添加磁盘。

57. 这里是请求处理函数。在block驱动之中最重要的是请求函数。

块设备的I/O操作方式与字符设备存在较大的不同，因而引入了request\_queue、request、bio等一系列数据结构。在整个块设备的I/O操作中，贯穿于始终的就是“请求”，字符设备的I/O操作则是直接进行不绕弯，块设备的I/O操作会排队和整合。

驱动的任务是处理请求，对请求的排队和整合由I/O调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。

尽管在块设备驱动中仍然存在block\_device\_operations结构体及其成员函数，但其不再包含读写一类的成员函数，而只是包含打开、释放及I/O控制等与具体读写无关的函数。块设备驱动的结构相当复杂的，但幸运的是，块设备不像字符设备那么包罗万象，它通常就是存储设备，而且驱动的主体已经由Linux内核提供，针对一个特定的硬件系统，驱动工程师所涉及到的工作往往只是编写少量的与硬件直接交互的代码。

request\_queue\_t \*blk\_init\_queue(request\_fn\_proc \*rfn, spinlock\_t \*lock);

该函数的第1个参数是请求处理函数的指针，第2个参数是控制访问队列权限的自旋锁，这个函数会发生内存分配的行为，故它可能会失败，因此一定要检查它的返回值。这个函数一般在块设备驱动的模块加载函数中调用。

58. 之前我们说了bio的bio\_vec函数是非常重要的核心函数，我们不应该直接访问bio的bio\_vec成员，而应该使用bio\_for\_each\_segment()宏来进行这项工作，可以用这个宏循环遍历整个bio中的每个段。其中参数Bvec指向当前 bio\_vec入口，Segno是当前segment编号。

当操作结束时，driver必须通过bio\_endio函数通知块设备子系统。

下面这一部分我们在一开始的时候讲过page、block和sector的关系。

59. 我们在这个图里可以看到，I/O Scheduler是一个控制请求顺序的东西，当有新得IO请求的时候I/O Scheduler进行接收请求，并维护一个请求队列，当前置请求处理完毕后I/O Scheduler从请求队列中获取请求并进行分发。

60. 接下来我们介绍块设备IO调度相关的知识。

61. 为什么我们需要IO调度？因为磁盘寻道在计算机系统中是最慢的操作，如果没有合适的I/O调度器，对系统性能影响非常大。

I/O scheduler 可以安排磁头在一个方向上移动，减少 seek 次数，像之前我们所介绍的电梯算法一样。这样就可以在全局范围内获得高吞吐量。

62. IO调度算法的一般性质：为了提高整个磁盘的吞吐量，则：1.对请求重新排序，从而减少寻道时间2.合并请求而降低请求数量

换句话说是忽略某些单个设备处理的时间，提高整体请求的处理效率。另一方面除了提高整体的总效率以外，还需要提供一个公平读写的条件，避免饿死情况出现。

63. 在IO调度算法中，我们需要考虑的事情包括旋转时间、寻道时间、传送时间、以及I/O 调度器的工作，其中I/O 调度器的工作包括排序请求队列、合并请求队列、分发I/O请求。

我们需要注意的是，在这里我们所有列举出来的事情都是在正式的操作之前需要进行的操作，也就是说我进行一次IO调度我是一定需要这些操作的。与我调度的内容和大小都无关。那么我们就不得不提到调度效率的问题了。上一节课我们在此也做过讨论：如何才能使得效率在90%以上？也就是需要提高单次操作的内容大小才可以保证一个效率的问题。

64. Linux elevator 是能附加的不同I/O调度器的抽象层，提供了一些函数供block layer使用

机制：系统提供了一些函数可以对队列进行merge

策略：sorting method, elevator提供：操作的办法是把请求放到队列中，然后从队列中分发请求。

65. 接下来介绍一下I/O调度的4种算法：

1)CFQ(完全公平排队I/O调度程序)

特点:

在最新的内核版本和发行版中,都选择CFQ做为默认的I/O调度器,对于通用的服务器也是最好的选择.

CFQ试图均匀地分布对I/O带宽的访问,避免进程被饿死并实现较低的延迟,是deadline和as调度器的折中.

CFQ对于多媒体应用(video,audio)和桌面系统是最好的选择.

CFQ赋予I/O请求一个优先级,而I/O优先级请求独立于进程优先级,高优先级的进程的读写不能自动地继承高的I/O优先级.

工作原理:

CFQ为每个进程/线程,单独创建一个队列来管理该进程所产生的请求,也就是说每个进程一个队列,各队列之间的调度使用时间片来调度,

以此来保证每个进程都能被很好的分配到I/O带宽.I/O调度器每次执行一个进程的4次请求.

2)NOOP(电梯式调度程序)

特点:

在Linux2.4或更早的版本的调度程序,那时只有这一种I/O调度算法.

NOOP实现了一个简单的FIFO队列,它像电梯的工作主法一样对I/O请求进行组织,当有一个新的请求到来时,它将请求合并到最近的请求之后,以此来保证请求同一介质.

NOOP倾向饿死读而利于写.

NOOP对于闪存设备,RAM,嵌入式系统是最好的选择.

电梯算法饿死读请求的解释:

因为写请求比读请求更容易.

写请求通过文件系统cache,不需要等一次写完成,就可以开始下一次写操作,写请求通过合并,堆积到I/O队列中.

读请求需要等到它前面所有的读操作完成,才能进行下一次读操作.在读操作之间有几毫秒时间,而写请求在这之间就到来,饿死了后面的读请求.

3)Deadline(截止时间调度程序)

特点:

通过时间以及硬盘区域进行分类,这个分类和合并要求类似于noop的调度程序.

Deadline确保了在一个截止时间内服务请求,这个截止时间是可调整的,而默认读期限短于写期限.这样就防止了写操作因为不能被读取而饿死的现象.

Deadline对数据库环境(ORACLE RAC,MYSQL等)是最好的选择.

4)AS(预料I/O调度程序)

特点:

本质上与Deadline一样,但在最后一次读操作后,要等待6ms,才能继续进行对其它I/O请求进行调度.

可以从应用程序中预订一个新的读请求,改进读操作的执行,但以一些写操作为代价.

它会在每个6ms中插入新的I/O操作,而会将一些小写入流合并成一个大写入流,用写入延时换取最大的写入吞吐量.

AS适合于写入较多的环境,比如文件服务器

AS对数据库环境表现很差.

66. 这里是调度器中的一些互相调用的关系。

67. I/O调度器看起来可以提高访问速度，但是这是并不是最快的，因为I/O调度过程会花费很多时间。最快的方式就是不使用I/O调度器（但是要能做到IO调度器的功能）。

要脱离I/O调度器，就必须了解请求队列request\_queue，因为I/O调度器和请求队列是绑定在一起的。其关系如图。

Request\_queue调用elevator\_queue调用调度相关的函数。

从图中我们也可以看出来请求队列request\_queue 中的elevator指针式指向I/O调度函数的。

68. 这里是对bio的处理过程，我们来进行具体分析：

（1）当需要读写一个数据的时候，通用块层，会根据用户空间的请求，生成一个bio结构体。

（2）准备好bio后，会调用函数generic\_make\_request()函数，函数原形如下：

void generic\_make\_request(struct bio \*bio)

（3）该函数会调用底层函数：

static inline void \_generic\_make\_request(struct bio \*bio);

（4）到这里会分层两种情况：

第一种，调用请求队列中自己定义的make\_request\_fn()函数，那问题来了，系统怎么知道这个自己定义函数在哪里呢？由内核函数blk\_queue\_make\_request()函数指定，函数原形：

void blk\_queue\_make\_request(struct request\_queue \*q,make\_request\_fn \*mfn);

第二种，使用请求队列中系统默认\_\_make\_request()函数，函数原形

static int \_\_make\_request(struct request\_queue \*q,struct bio \*bio);

该函数会启动I/O调度器，对bio进行调度处理，bio结构或被合并到请求队列的一个请求结构的request中。最后调用request\_fn\_proc()将数据写入或读出块块设备。

69. 不使用i/o调度器(blk\_alloc\_queue())

bio的流程完全由驱动开发人员控制，要达到这个目的，必须使用函数blk\_alloc\_queue()来申请请求队列，然后使用函数blk\_queue\_make\_requset()给bio指定具有request\_fn\_proc()功能的函数Virtual\_blkdev\_make\_request来完成数据在内存和硬盘之间的传输（该函数本来是用来将bio加入request中的）。

使用i/o调度器（blk\_init\_queue()）

bio先经过\_\_make\_request()函数，I/O调度器，和request\_fn\_proc()完成内存和硬盘之间的数据传输。该过程使用函数blk\_init\_queue()函数完成队列的初始化，并指定request\_fn\_proc()。

70. 接下来我们来看一个实例来进行分析。以sbull来进行例子的解析：

71. 首先先来了解一下sbull是什么：

Sbull完成基于内存的ramdisk功能

加载sbull后，我们可以在内存虚拟的disk上进行分区、创建、删除、读写文件。

教材中的例子会因为释放设备一段时间后而让模拟块设备的内存清空。为了避免这种情况，可以修改部分代码

72.这里是sbull的析构函数

73. 这里是sbull的设备驱动setup启动函数。

memset (dev, 0, sizeof (struct sbull\_dev));

初始化 dev 所指内容为0，

然后指定了设备的size和data。

初始化并指向了一个timer，将此数据结构内的data和function分别进行了初始化为dev和sbull的定义结构。

74.

这里是进行实际调用时候的open函数。

在这其中，包括去掉"介质移除"定时器操作（del\_timer\_sync(&dev->timer)）

对设备进行PV操作（spin\_lock(&dev->lock);和spin\_unlock(&dev->lock);）避免其他操作产生冲突。

在对设备的操作中，check\_disk\_change(inode->i\_bdev);修改检测状态，dev->users++;将设备的用户数+1.

75.

这里是释放函数，在进行调用完成后释放掉对设备的使用控制。

基本上是之前函数的反向操作：在PV操作之内，

然后用户数-1，对dev的设置进行取消，将timer进行修改。

76. 进行简单的请求处理时进行调用此函数。

77.当进行复杂请求（或者说是正常请求时）调用此函数。

78. 好 同学们 这就是本节课的全部内容。