内存管理

1. 内存管理

基本思路：低端存储的容量尽量大、高端存储的速度尽量快

存储管理的功能

地址变换

将程序地址空间中使用的逻辑地址变换成主存中的地址的过程，又称地址重定位。地址变换的功能就是要建立虚实地址的对应关系。这一过程由内存管理单元（MMU）来完成。为了能够完成虚拟内存地址到物理内存地址的翻译，MMU 会有一个表单。表单中，一边是虚拟内存地址，另一边是物理内存地址。

事实上，MMU 并不会保存这一表单，它只会从内存中读取它，然后完成翻译。为此，CPU 中需要有一些寄存器用来存放表单在物理内存中的地址。在 RISC-V 架构中，该寄存器为 SATP。当操作系统将 CPU 从一个应用程序切换到另一个应用程序时，同时也需要切换 SATP 寄存器中的内容，从而指向新的进程保存在物理内存中的地址对应表单。

用户程序看不见真正的物理地址。用户只生成逻辑地址，且**认为**进程的地址空间为 0 到 max。物理地址范围从 R + 0 到 R + max，R 为基地址。

逻辑地址（虚拟地址）：用户编程序时所用的地址（或称程序地址 、虚地址 ），基本单位可与内存的基本单位相同，**也可以不相同。**

逻辑地址空间：由程序所生成的所有逻辑地址的集合；可以是一维线性空间，也可以是多维空间。

物理地址：把内存分成若干个大小相等的存储单元，每个单元给一个编号，这个编号称为内存地址，即内存单元所用的地址。

物理地址空间：与逻辑地址相对应的内存中所有物理地址的集合，一维的线性空间

主存分配

按照一定的算法把某一空闲的主存区分配给作业或进程。

内存保护

存储保护：保证用户程序(或进程映象)在各自的存储区域内操作，互不干扰。措施：

1. 上下界保护：下界寄存器存放程序装入内存后的开始地址（首址）。上界寄存器存放程序装入内存后的末地址。**（物理地址方面）**

2. 基址、限长寄存器保护：基址寄存器存放程序装入内存后的起始地址。限长寄存器存放程序装入内存后的最大长度。**（逻辑地址方面）**

对于合法的访问地址，这两者的效率是相同的，对不合法的访问地址来说，上下界存储保护浪费的 CPU 时间相对来说要多些。

虚拟存储

使用户程序的大小和结构不受主存容量和结构的限制，即使在用户程序比实际主存容量还要大的情况下，程序也能正确运行。

地址重定位

进程在执行时，会访问内存中的指令和数据。将指令和数据捆绑到内存地址可以在以下步骤的任何一步中执行。程序在编译时确定地址变换关系：如果内存位置已知，可生成绝对代码；如果开始位置改变，需要重新编译代码。 在用机器指令编程时，程序员直接按物理内存地址编程，这种程序在系统中是不能做任何移动的，否则就会出错。

加载时(静态地址变换)

静态地址变换是在程序装入内存时完成从逻辑地址到物理地址的转换的。在一些早期的系统中都有一个装入程序（加载程序），它负责将用户程序装入系统，并将用户程序中使用的访问内存的逻辑地址转换成物理地址。 **如果存储位置在编译时不知道，则必须生成可重定位代码。**

优点：实现简单，不要硬件的支持

缺点：程序一旦装入内存，移动就比较困难，有时间上的浪费。在程序装入内存时要将所有访问内存的地址转换

执行时(动态地址变换)

动态地址变换是在程序执行时由系统硬件完成从逻辑地址到物理地址的转换的。**如果进程在执行时可以在内存中移动，则地址绑定要延迟到运行时。**

- 需要硬件对动态地址变换的支持，例如基址和限长寄存器
- 基址寄存器这时称为重定位寄存器。用户进程所生成的地址在送交内存之前，都将加上重定位寄存器的值

动态地址变换是由硬件执行时完成的，程序中不执行的程序就不做地址变换的工作，这样节省了CPU 的时间。系统中设置了重定位寄存器，重定位寄存器的内容由操作系统用特权指令来设置，比较灵活。**实现动态地址变换必须有硬件的支持，并有一定的执行时间延迟。**现代计算机系统中都采用动态地址变换技术。

动态地址变换技术的优点：

（1）具有给一个用户程序**任意分配**内存区的能力；

（2）可实现虚拟存储；

（3）具有**重新分配**的能力；

（4）对于一个用户程序，可以分配到多个不同的存储区

动态加载

一个子程序只有在调用时才被加载。

- 更好的内存空间利用率；不用的子程序不会被装入内存
- 当需要大量的代码来处理不经常发生的事情时是非常有用的
- 不需要操作系统的特别支持，通过程序设计实现

动态链接

链接被推迟到执行时期

- 二进制映像中对每个库程序的引用都有一个存根。 存根是小的代码片，用来定位合适的保留在内存中的库程序
- 存根用子程序地址来替换自己，并开始执行子程序
- 动态链接需要操作系统的帮助。操作系统需要检查子程序是否在进程的内存空间，或是允许多个进程访问同一内存地址

连续存储空间管理

主存通常被分为两部分：

- 用于驻留操作系统：为操作系统保留的部分，通常用中断向量保存在内存低端
- 用于用户进程 ： 保存在内存高端

每个程序（作业）占据主存中连续的空间，按管理方式的不同分为：

- 单用户连续存储管理
- 固定分区存储管理
- 可变分区存储管理

采用连续内存分配时，每个进程位于一个连续的内存区域

- 动态定位：专门设置一对地址寄存器（上限/下限寄存器），硬件地址转换机构对相应的地址进行比较
- 静态定位：地址转换时检查其绝对地址是否落在为其分配的用户分区

单用户连续存储管理

又称单分区模式，适用于单用户情况，任何时刻**主存储器**中最多只有**一道**程序。主存空间划分为系统区和用户区。

地址转换与存储保护：

- 地址转换：物理地址 = 界限地址 + 逻辑地址
- 多采用静态重定位，采用栅栏寄存器进行存储保护
- 动态重定位，采用定位寄存器进行存储保护

单用户连续存储管理的缺点： 同单道程序的缺点，系统利用率低

固定分区存储管理

又称定长分区或静态分区模式，是满足**多道程序设计**需要的**最简单的**存储管理技术。比较适合已知程序（作业）大小和出现频率的情形 。

基本思想： 给进入主存的用户作业划分一块连续存储区域，把作业装入该连续存储区域，若有多个作业装入主存，则它们可并发执行。

实现： 系统启动时，系统操作员根据作业情况静态地把可分配的主存储器空间（用户空间）分割成若干个连续的区域，每个区域的位置固定，大小可相同也可不同，每个分区在任何时刻最多只装入一道程序执行。

作业调度策略：所有作业排成一个队列，每个等待作业被选中时，排到一个能够装入它的最小分区的等待队列。**该调度方式可能导致分区使用不均匀。**

固定分区存储管理的缺点

- 实际系统运行时，往往无法预知分区大小（太大，等同于 “单用户分区模式” ）
- 主存空间利用率仍然较低
- 无法适应动态扩充主存

- 分区数目预先确定，限制了多道运行程序的数量

可变分区内存分配

根据一组空闲分区来分配大小为 n 的请求，常用方法有： 首次适应(first fit)、最佳适应(best fit)、最坏适应(worst fit)、下次适应(next fit)、快速适应(quick fit)。在存储速度和存储资源的利用上，首次适应和最佳适应要比最差适应好。

基本思想：按作业的大小划分分区，但划分的时间、大小和位置均动态确定，系统在作业装入主存执行之前并不建立分区

通用数据结构：已分配区表、未分配区表。

1. 首次适应算法

首次适应算法是按空闲区首址升序的（即空闲区表是按空闲区首址从小到大）方法执行的。分配时从表首开始，以请求内存的大小逐个与空闲区进行比较，找到**第一个满足要求**的空闲区为止；若大于请求区，就将该空闲区的**一部分**分配给请求者，然后，修改空闲区的大小和首址。修改空闲区有两种方法：从空闲区头开始、从空闲区尾开始。

回收时，首先考察释放区是否与系统中的某个空闲区相邻，若相邻则合并成一个空闲区，否则，将释放区作为一个空闲区**按首址升序**的规则插入到空闲区表适当的位置。

这种算法的实质是**尽可能地利用低地址部分的空闲区**，而尽量地保证高地址部分的大空闲区，使其不被切削成小的区，其目的是保证在已有大的作业的到来有足够大的空闲区来满足请求者。

2. 最佳适应算法

最佳适应算法是将申请者放入与其**大小最接近**的空闲区中，使得**切割后的空闲区最小。**若系统中有与申请区大小相等的空闲区，这种算法肯定能将这种空闲区分配给申请者。（首次适应法则不一定）

最佳适应算法的空闲区表按**空闲区大小升序**方法组织。分配时，按申请的大小逐个与空闲区大小进行比较，找到一个满足要求的空闲区，就说明它是最适合的（即最佳的）。

这种算法最大的缺点是**分割后的空闲区将会很小**，直至无法使用，而造成浪费。

3. 最坏适应算法

为了克服最佳适应算法把空闲区切割得大小的缺点，人们提出了一种最坏适应算法，即每次分配时，总是将**最大**的空闲区切去一部分分配给请求者，其依据是当一个很大的空闲区被切割了一部分后可能仍是一个较大的空闲区，**避免了空闲区越分越小的问题。**

最坏适应算法的空闲区表是按**空闲区大小降序**的方法组织的（从大到小的顺序）。分配时总是**取表中的第一个表目**，若不能满足申请者的 要求，则表示系统中无满足要求的空闲区，分配失败；否则，将从该空闲区中分配给申请者，然后修改空闲区的大小，并将它插入到空闲区表的适当位置。

4. 下次适应算法

与首次适应算法类似，不同的是每次找到合适的空闲的分区时就记住它的位置，以便下次就从该位置开始往下查找，而不是每次都像首次适应算法那样从头开始查找。

这种算法的总体结果通常要比首次适应算法差。由于它经常会在内存的末尾分配存储分区，所以位于存储空间末尾的最大分区会被撕裂成小的外部碎片，因此必须经常不断地进行存储紧凑。在该算法中应采取循环查找方式，即最后上个空闲区的大小仍不能满足要求时，应再从第一个空闲区开始查找，故又称为循环造就算法。

分页式存储管理

传统内存管理技术的问题

- 进程规模增大
- 存储空间碎片化（压缩）问题
- 交换时的长时间延迟

解决思路

- **进程规模增大**

  更精细地划分内存，任何页面都可以映射到任何帧。

- **存储空间碎片化（压缩）问题**

  在这样的设计下，进程不需要是连续的，因此不需要压缩。

- **交换时的长时间延迟**

  只交换进程的一部分。

基本原理

允许进程的物理地址空间可以是不连续的，即允许作业存放在若干个不相邻的分区中，这样既可免去移动内存信息而产生的工作量，又可充分利用主存空间，尽量减少主存碎片。

将程序（作业）地址空间分成大小相等的页面，称为页（Page）；同时把物理内存也分成与大小固定的块。称为帧（Frame）。页、帧大小由硬件来决定，通常为 2 的幂。当一个用户程序装入内存时，以页面为单位进行分配。

基本概念

帧：主存空间按物理地址分成多个大小相等区，每个区称为块(帧)（又称 frame）

页：程序（作业）按逻辑地址分成多个大小相等的区，每个区称为一个页面 (page)，大小与帧大小相等

虚地址（逻辑地址）：页号和页偏移

内存映射

可选的数据结构

链表：访问时间 O(n)，所以查找时间会随着地址规模增大而变长

数组：固定的访问时间，但是需要较大规模的连续存储空间

哈希表：固定的访问时间，规模有限，但是会出现重叠

二叉树：优秀的摊销预期时间，但会带来较多的内存读写操作，且难以在硬件上实现

页表

页表是**页式存储管理**的数据结构。它包括用户程序空间的页面与内存块的对应关系、页面的存储保护和存取控制方面的信息。在实际的系统中，为了节省存储空间，在页表中可以省去页号这个表目。其包括：

- Page-table base register (PTBR)：页表基址寄存器指向页表
- Page-table length register (PRLR)：页表限长寄存器表明页表的长度

动态重定位：让程序的指令执行时动态地进行地址变换，给每个页面设立重定位寄存器，**重定位寄存器的集合便称页表。**页表是操作系统为每个用户作业建立的，用来记录程序页面和主存对应页框的对照表。

现代的大多数计算机系统，都支持非常大的逻辑地址空间(2^32~2^64)。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。例如，假设页面大小为 4KB，逻辑地址空间为 32 bit，则页表条目数为 2^20 个。如果每个条目占用 4 Byte 存储空间，则页表本身将占据 4MB 大小的空间。

为解决页表规模过大占用内存空间的问题，引入了**多级页表**。多级页表将页表进行分页。每个页面的大小与内存物理块的大小相同，并为它们进行编号。这样可以离散地将各个页面分别存放在不同的物理块中，为此再建立一张页表，称为**外层页表（页表目录）**，即第一级页表，其中的每个表目是存放某个页表的物理地址。第二级才是页表（其中每个物理块上的页表叫做页表分页），其中的每个表目所存放的才是页的物理块号。

对于上面的例子， 4MB = 1024 * 4KB。为此可以增加含有 1024 个表项的外层页表（页表目录）。一级页表占据的空间大小为 4KB。

在二级页表系统中，一次按逻辑地址的主存访问需要访问三次主存：一次访问页目录、一次访问页表、一次访问具体的数据。

稀疏的页面目录

包含指向页表的指针与 NULL 空指针。在大多数情况下，实际有效的页表约为 2 ~ 3 页，对应 12 ~ 16KB而不是 4MB。

虚地址转换

基本步骤

页号从 0 开始计算

虚地址（逻辑地址、程序地址）以十六进制、八进制、二进制的形式给出：

按页的大小分离出页号和位移量（低位部分是位移量，高位部分是页号），将位移量直接复制到内存地址寄存器的低位部分，以页号查页表，得到对应页装入内存的块号，并将块号转换 成二进制数填入地址寄存器的高位部分，从而形成内存地址。简而言之，内存地址 = 块号 × 页大小 + 位移量。

虚地址以十进制数给出：

- 页号 = 虚地址 整除 页大小
- 位移量 = 虚地址 mod 页大小
- 以页号查页表，得到对应页装入内存的块号
- 内存地址 = 块号 × 页大小 + 位移量

例如：有一系统采用页式存储管理，有一作业区大小是 8KB，页大小为 2KB，依次装入内存的第 7、 9 、 A 、 5 块。则逻辑地址 0AFE 转化为物理地址为 4AFE。

```
0000 1010 1111 1110 -> 00001-01011111110 (5bit 页号 + 11bit 页内地址偏移)
0100 1010 1111 1110 <- 01001-01011111110 (5bit 块号 + 11bit 块内地址偏移)
```

例如：有一系统采用页式存储管理，有一作业大小是 8KB，页大小为 2KB，依次装入内存的第7、9、10、5 块。则逻辑地址 3412 转换成物理地址为 19796。

```
页号 = 3412 % 2048 = 1 -> 块号 = 9
页内地址偏移 = 3412 mod 2048 = 1364 -> 块内地址偏移 = 1364
故物理地址 = 9 * 2048 + 1364 = 19796
```

可能的问题

在分页式存储管理的设计下，处理器会进行**两次内存访问**：

1. 将 %esi 中的地址拆分为页码、页内偏移量

2. 将页码添加到页表基址寄存器，从**内存中获取页表条目 (PTE)**
3. 将帧地址与页内偏移量连接
4. **从内存中获取程序数据到 %eax**

存储保护

读写保护由与每个帧相关联的保护位来实现，可以自定义读写权限。此外，有效无效位附在页表的每个表项中。“有效” 表明相关的页属于进程的逻辑地址空间，并且是一个有效的 (Valid) 页；“无效” 表明页不在进程的逻辑地址空间中，或 valid 但在外存中。

共享页

共享代码：一段只读（可重入）代码副本可由进程共享。一般而言，共享代码出现在进程的逻辑地址空间的相同位置。 如：文本编辑器，窗口系统等。

私有代码和数据：每个进程保留代码和数据的私有副本。私有代码和数据的页可以出现在逻辑地址空间的任何地方。

碎片问题

连续分区管理方式存在的问题：每个程序总是要求占用连续的存储空间，经过一段时间的运行将会产生许多碎片（不连续的容量较小的分区），为接纳新的作业，往往需要通过移动已有的主存内容来产生容量较大的分区。移动技术实现复杂，并不可避免地导致管理开销增大。

碎片：在采用分区存储管理的系统中，会形成一些非常小的分区，最终这些非常小的分区不能被系统中的任何用户（程序）利用而浪费。 造成这样问题的主要原因是用户程序装入内存时是整体装入的。

- 外部碎片：当整个内存空间可以满足一个请求，但它不是连续的。这些不联系区域对应着外部碎片
- 内部碎片：分配的内存（2 的整数幂）可能比申请的内存大一点。这两者之间的差值对应着内部碎片

传统内存管理技术

移动技术：移动内存内容，把一些小的空闲内存结合成一个大的块。只有在执行时期进行动态重定位 (relocation)，才有可能进行移动。

交换技术：把暂时不用的某个程序及数据的一部分或全部从内存移到 外存中去，或把指定的程序或数据从外存读到相应的内存中，并将控制权转给该程序的一种内存扩充技术。交换可能让总的物理地址空间大于内存的物理地址空间。

如果内存空间需求大于当前可用容量，则仅通过移动无法满足内存需求。为了保证每个进程 100 % 利用系统内存，将每个进程轮流调入调出内存。然而这样做会有如下的缺点：

- 交换时进程无法继续运行
- 对于每一个进程，都需要额外的时间（秒量级）进行调度

覆盖技术：将大程序划分为一系列的覆盖，每个覆盖是一个相对独立的程序单位，把程序执行时不需要同时装入内存的覆盖构成一组，称为覆盖段。一个覆盖段内的覆盖共享同一存储区域，该区域成为覆盖区，其大小由对应的覆盖段内最大的覆盖决定。

分页式存储管理中的碎片

没有外部碎片（没有连续性要求），有内部碎片。最坏情况下，一个需要 n 页再加 1B 的进程，需要 n+1 个帧，几乎有一个帧大小的内部碎片。

相联存储器 TLB

通常页表存放在主存中，因此按逻辑地址访问某个主存地址内容时，需要涉及二次主存访问，效率较低。**相联存储器（Translation Look-aside Buffer, TLB）**是一个专用的高速缓冲存储器，用于存放最近被访问的部分页表，是分页式存储管理的重要组成部分。

相联存储器采用并行搜索模式，即同时查找其中存储的结果。如果页号 A 在相联寄存器内，则得到对应的帧号；否则查找内存中的页表来得到帧号。

TLB 只记录最后一层页表，没有分层 TLB。

基于 TLB 的有效内存访问时间

相联存储器的查找需要时间：ε

访问内存一次需要时间：t

命中率：在相联存储器中找到页号的概率，概率与相联存储器的大小有关。设为 α

则**有效访问时间（Effective Access Time， EAT）为** $(t + \varepsilon)\alpha + (2t + \varepsilon)(1 - \alpha)$

分段式存储管理

为提高主存空间的利用率，存储管理方式变更为： 固定分区→动态分区→分页方式。为满足程序设计和开发的要求（以模块为单位的装配、共享和保护），出现了分段式存储管理。一个程序是一些段的集合，一个段是一个逻辑单位，如：

- 主函数
- 过程，函数，方法，对象
- 局部变量，全局变量
- 堆栈
- 符号表，数组

实施分配时，可基于可变分区存储管理的原理，以段为单位进行主存分配。段共享通过不同作业段表中的项指向同一个段基址来实现。段表中保存了界限和基地址，将物理内存划分为了不同长度的段。逻辑地址被分为两部分：段号和段偏移。如果对段有非法引用，则会陷入操作系统。

基本概念

逻辑地址：段号和段内地址

段表、作业表

分页和分段的主要区别

(1)页是信息的物理单位，分页是由于系统管理的需要，减少外部碎片，提高内存利用率。段则是信息的逻辑单位，它含有一组其意义相对完整的信息，是为了能更好地满足用户的需要。

(2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定， 决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

(3) 分页的作业地址空间是一维的，即单一的线性地址空间； 而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

Linux 在 x86 上的实现

使用最小的分段来保持内存管理实现更具可移植性。段分为六个部分：内核代码、内核数据、用户代码（由所有用户进程共享，使用逻辑地址）、用户数据（同样共享）、任务状态（每个进程硬件上下文）、LDT。

分为内核模式、用户模式。

小结

比较不同内存管理策略（连续分配、分页、分段）时，考虑以下方面

- **硬件支持：**寄存器, 页表, 段表
- **性能：**快速寄存器、页表、**TLB**
- **碎片：**固定分区，内部碎片；多个分区、分段，外部碎片
- **重定位**

  紧缩，在内存中移动程序

  要求在执行时逻辑地址能动态重定位
- **共享**

  要求分页或分段
- **保护**

  页、段表中的保护位

  基址、限长寄存器

思考题

例：在可变分区存储管理下，按地址排列的内存空闲区为：10K 、4K 、20K 、18K 、7K 、9K 、12K 和 15K 。对于下列的连续存储区的请求：(1) 12K 、10K 、9K , (2) 12K 、10K 、15K 、18K ， 试问：使用首次适应算法、最佳适应算法、最差适应算法和下次适应算法，分别是哪个空闲区被使用?

答：空闲分区如下表所示

分区号	分区长
1	10K
2	4K
3	20K
4	18K
5	7K
6	9K
7	12K
8	15K

对于请求 1：

首次适应算法

12KB 选中分区 3 ， 这时分区 3 还剩 8KB 。10KB 选中分区 1 ， 恰好分配故应删去分区 1 。9KB 选中分区 4 ， 这时分区 4 还剩 9KB 。

最佳适应算法

12KB 选中分区 7，恰好分配故应删去分区 7。10KB 选中分区 1，恰好分配故应删去分区 1。9KB 选中分区 6，恰好分配故应删去分区 6。

最差适应算法

12KB 选中分区 3，这时分区 3 还剩 8KB。10KB 选中分区 4，这时分区 4 还剩 8KB。9KB 选中分区 8，这时分区 8 还剩 6KB。

下次适应算法

12KB 选中分区 3，这时分区 3 还剩 8KB。10KB 选中分区 4，这时分区 4 还剩 8KB。9KB 选中分区 6，恰好分配故应删去分区 6。

对于请求 2：

首次适应算法

12KB 选中分区 3，这时分区 3 还剩 8KB。10KB 选中分区 1，恰好分配故应删去分区 1。15KB 选中分区 4，这时分区 4 还剩 3KB。最后无法满足 18KB 的申请，应该等待。

最佳适应算法

12KB 选中分区 7，恰好分配故应删去分区 7。1OKB 选中分区 1，恰好分配故应删去分区 1。15KB 选中分区 8，恰好分配故应删去分区 8。18KB 选中分区 4，恰好分配故应删去分区 4。

最差适应算法

12KB 选中分区 3，这时分区 3 还剩 8KB。10KB 选中分区 4，这时分区 4 还剩 8KB。15KB 选中分区 8，恰好分配故应删去分区 8。最后无法满足 18KB 的申请，应该等待。

下次适应算法

12KB 选中分区 3，这时分区 3 还剩 8KB。1OKB 选中分区 4，这时分区 4 还剩 8KB。15KB 选中分区 8，恰好分配故应删去分区 8。最后无法满足 18KB 的申请，应该等待。

例：一台计算机的内存空间为 1024 个页面，页表放在内存中，访问内存的开销是 500ns。为了减少开销，使用了有 32 个字的快表，TLB 查找速度为 100ns。则要把平均开销降到 200ns 需要的快表命中率是多少?

答：设快表命中率是 x，则内存命中率为 1-x。于是：1200 (1-x) + 100x = 200，解方程得 x = 90.9%。

2. 虚拟内存

虚拟内存

一些基本事实

- 数组、链表和表通常分配了比实际所需要更多的内存
- 程序的某些选项或特点可能很少使用。即使需要完整程序，也并不是在某时刻同时需要

采用虚拟内存的优点

- 保存部分程序在内存中，可运行一个比物理内存大的多的程序
- 逻辑地址空间能够比物理地址空间大
- 可以有更多程序同时运行
- 允许若干个进程共享地址空间

- 进程创建高效

实现方式

- 请求分页存储管理(Demand paging)
- 请求分段存储管理(Demand segmentation)
- 请求段页式存储管理

局部性原理

时间局部性： 如果程序中的某条指令一旦执行， 则不久以后该指令可能再 次执行；如果某数据被访问过， 则不久以后该数据可能再次被访问。产生时间局限性的典型原因，是由于在程序中存在着大量的循环操作。

空间局部性： 一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。

虚拟存储管理

理论基础——作业为什么能够部分装入和部分对换?

作业信息（程序）的局部性，使得在作业信息不完全装入主存的情况下能够保证其正确运行

- 空间局部性，一段时间内，仅访问程序代码和数据的一小部分；
- 时间局部性，最近访问过的程序代码和数据，很快又被访问的可能性很大。

与对换技术的区别

虚拟存储管理：

- 以页或段为单位处理
- 进程所需主存容量大于当前系统空闲量时仍能运行

对换技术（中级调度，挂起和解除挂起）：

- 以进程为单位处理
- 进程所需主存容量大于当前系统空闲量时，无法解除挂起

需要解决的主要问题

- 主存和辅存的统一管理问题
- 逻辑地址到物理地址的转换问题
- 部分装入和部分对换问题
- 磁盘 I/O 非常费时，故应当降低缺页率

请求分页式存储管理 （Demand Paging)

分页式存储管理技术的扩展，是一种常用的分页式虚拟存储管理技术。基本原理是：将作业信息被分为多个页面，其副本存放在辅助存储器中。当作业被调度运行时，仅装入需要立即访问和使用的页面，在执行过程中如果需要访问的页面不在主存中，则将其动态装入。

特点

- 需要很少的 I/O
- 需要很少的内存
- 响应快
- 支持多用户

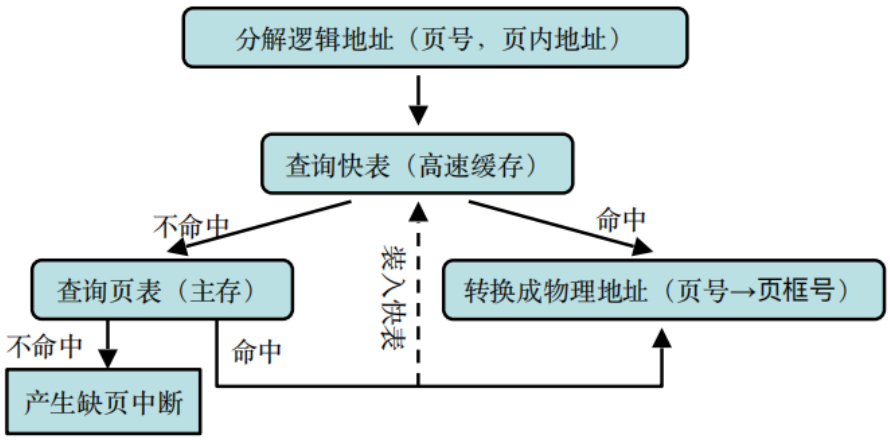
页表的扩展

增加驻留标志、缺页标志、脏页标志、访问标志、锁定标志、淘汰标志等。

硬件支持

操作系统的存储管理需要依靠低层硬件的支撑来完成，该硬件称为主存管理单元 MMU。MMU 的主要功能为完成逻辑地址到物理地址的转换，并在转换过程中产生相应的硬件中断（缺页中断、越界中断）。包括：**页表基址寄存器、快表 TLB。**

工作流程



需要页面调度时，首先查阅此页的引用

- Case1: 无效引用 ⇒ 中止
- Case2: 不在内存 ⇒ 将其调入内存

1. 缺页中断
2. 找一个空闲帧
3. 将需要的页调入空闲帧
4. 重置页表，有效位为 1
5. 重启指令

交换

当需要调页但没有空闲帧时，可以把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存，腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据，调入内存。这一过程称为交换。

实现这一过程的数据结构：其形式与内存的动态分区分配方式中所用数据结构相似，空闲分区表或空闲分区链。 在空闲分区表中的每个表目中应包含两项， 即对换区的首址及其大小，它们的单位是盘块号和盘块数。

交换可有效提高内存利用率。

页替换

1. 在磁盘上找到所需页面的位置。
2. 找到一个空闲帧。如果有一个空闲帧，则使用它；如果没有空闲帧，则使用页面替换算法来选择一个换出帧。
3. 将所需的页面读入（新的）空闲帧中。更新页表和帧表。
4. 重新启动进程

一些问题

页面装入策略：何时将一个页面装入主存?

- 请页式调入，缺页中断驱动，一次调入一页
- 预调式调入，按某种预测算法动态预测并调入若干页面）

消除策略：何时将修改过的页面写回辅存?

- 请页式清除，仅当一页被选中进行替换时，该页内容已修改则写回辅存。（清除与替换成对）
- 预约式清除，内容被修改页面成批写回辅存，写回操作在该页面被替换前，而非替换时

帧分配算法: 给每个进程分配多少帧?

页替换算法: 怎样选择要替换的帧?

请求分页的性能问题

缺页概率 $0 \le p \le 1$

EAT = (1 – *p*) x 内存访问时间 + *p* (页错误开销 + 换出开销 + 换入开销 + 重启开销)

有效访问时间（EAT） = Hit Rate x Hit Time + Miss Rate x Miss Time

例如，内存访问时间 = 200 ns，缺页时的页替换时间 = 8 ms，缺页率 p ，则

有效访问时间 = (1 – p) x 200ns + p x 8 ms

= (1 – p) x 200ns + p x 8,000,000ns

= 200ns + p x 7,999,800ns

影响缺页率的因素：页面替换算法、主存页框数 、页面大小、程序特性

写时复制 （Copy on write）

问题背景

fork( ) 会产生一个和父进程完全相同的子进程 (除了pid)。如果按传统的做法，系统会直接将父进程的数据拷贝到子进程中，拷贝完之后，父进程和子进程之间的数据段和堆栈是相互独立的。但是往往子进程都会执行 exec( ) 来做自己想要实现的功能。所以，如果按照上面的做法的话，创建子进程时复制过去的数据是没用的 (因为子进程执行 exec( )，原有的数据会被清空)。

既然很多时候复制给子进程的数据是无效的，于是就有了 Copy On Write 这项技术了，原理也很简单：fork 创建出的子进程，与父进程共享内存空间。也就是说，如果子进程不对内存空间进行写入操作的话，内存空间中的数据并不会复制给子进程，这样创建子进程的速度就很快了！(不用复制，直接引用父进程的物理空间)。
并且如果在 fork 函数返回之后，子进程第一时间 exec 一个新的可执行映像，那么也不会浪费时间和内存空间了。

换而言之，在 fork 之后 exec 之前两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，两者的虚拟空间不同，但其对应的物理空间是同一个。当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间。

如果不是因为 exec，内核会给子进程的数据段、堆栈段分配相应的物理空间（至此两者有各自的进程空间，互不影响），而代码段继续共享父进程的物理空间（两者的代码完全相同）。

而如果是因为 exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

Copy On Write 技术实现原理：

fork( ) 之后，kernel 把父进程中所有的内存页的权限都设为 read-only，然后子进程的地址空间指向父进程。当父子进程都只读内存时，相安无事。当其中某个进程写内存时，CPU 硬件检测到内存页是 read-only 的，于是触发页异常中断（page-fault），陷入 kernel 的一个中断例程。中断例程中，kernel就会把触发的异常的页复制一份，于是父子进程各自持有独立的一份。

Copy On Write 技术好处是什么?

COW 技术可减少分配和复制大量资源时带来的瞬间延时。COW 技术可减少不必要的资源分配，比如 fork 进程时，并不是所有的页面都需要复制，父进程的代码段和只读数据段都不被允许修改，所以无需复制。

Copy On Write 技术缺点是什么?

如果在 fork ( ) 之后，父子进程都还需要继续进行写操作，那么会产生大量的分页错误(页异常中断 page-fault)，这样就得不偿失。

总结:

fork 出的子进程共享父进程的物理空间，当父子进程有内存写入操作时，read-only 内存页发生中断，将触发的异常的内存页复制一份 (其余的页还是共享父进程的) 。

fork 出的子进程功能实现和父进程是一样的。如果有需要，我们会用 exec( ) 把当前进程映像替换成新的进程文件，完成自己想要实现的功能。

帧分配算法——给每个进程分配多少帧

固定分配

在一个进程的生命周期中，分配给它的帧数固定。例如：平均分配、按比例分配、优先权分配

可变分配

在一个进程的生命周期中，当进程缺页次数较多时，分配给它较多的帧，反之，则分配给较少的帧。

1. 平均分配算法

这是将系统中所有可供分配的帧，平均分配给各个进程。 例如，当系统中有 100 个帧，有 5 个进程在运行时，每个进程可分得 20 个帧。这种方式貌似公平，但实际上是不公平的，因为它未考虑到各进程本身的大小。如果有一个进程其大小为 200 页，只分配给它 20 个帧，这样，它必然会有很高的缺页率；而另一个进程只有 10 页，却有 10 个帧闲置未用，带来了性能的浪费。

2. 按比例分配算法

这是根据进程的大小按比例分配帧的算法。如果系统中共有 n 个进程，每个进程的页数为 S_i，则系统中各进程页数的总和为：S = \sum_{i=1}^{n} S_i 。又假定系统中可用的帧总数为 m ，则每个进程所能分到的帧数为 b_i ，将有：

$$b_i = \frac{S_i}{S} \times m$$

b 应该取整，因为它必须大于系统最小帧数。

3. 考虑优先权的分配算法

在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成， 应为它分配较多的内存空间。通常采取的方法是把内存中可供分配的所有帧分成两部分：一部分按比例地分配给各进程；另一部分则根据各进程的**优先权**，适当地增加其相应份额后，分配给各进程。在有的系统中，如**重要的实时控制系统**，则可能是完全按优先权来为各进程分配其帧的。

页替换算法——怎样选择要替换的帧?

1. FIFO 页替换算法

可以创建一个**FIFO** 队列来管理内存中的所有页。调入页时，将它加到队列的尾部；当必须替换一页时，将选择最旧的页（队列头）

2. 最优页替换算法

被替换的页是未来最长时间不被使用的页。该算法很难实现，因为需要未来的知识 。

最优页替换的作用：用来衡量其他算法的性能

3. 随机页替换算法

被替换的页是随机选择的页。这是 TLB 采用的一种应用方式，硬件结构简单，但性能难以预测，实时性难以保证。

4. 最近最少使用 Least Recently Used （LRU）页替换算法

LRU 替换算法为每个页记录该页最后的使用时间。当必须进行页替换时，LRU 选择最近最长未被使用的

页。

栈实现——在一个双链表中保留一个记录页访问数目的栈。每次将最近被访问的页移到栈顶。移除时，栈底部即是要找的页（最近最少使用）

计数器实现——每个页表项都关联一个使用时间域。这需要一个逻辑时钟或计数器，对每次内存引用，计数器都会增加。 每次内存引用时，时钟寄存器的内容都会复制到相应页表项的使用时间域内。进行页替换时，选择具有最小时间（或者计数器值）的页进行换出（这对应了该页上一次被使用的时间最远）。

计数器实现带来的问题：需要搜索页表、每次内存访问都需要写页表项的使用时间域、上下文切换时需要维护页表、需要考虑时钟溢出。

增加页表空间时，使用 LRU 或 最优页替换算法，具有 X 页面的内存内容将是具有 X + 1 页面的内存内容的子集。但对于 FIFO，其内容可能完全不同。

Belady 异常：页表空间中更多的帧有可能带来更多的页面错误

5. LRU 近似页替换算法

每个页都与一个特定 bit 位相关联，称为 Reference bit, 初始值为 0。 当页被访问时，将该页的 Reference bit 设为 1；替换时，选择 Reference bit 为 0 的第一个页。该操作的缺点：时间顺序未知。

时钟算法

Arrange physical pages in circle with single clock hand. Replace an old page, not the oldest page

Details:

Hardware “reference” bit per physical page:

替换 Reference bit 为 0 的第一个页。缺点：顺序未知

如果 Reference bit 为 0, 表明近期未用

Nachos hardware sets use bit in the TLB

缺页时:

沿 Circle 顺序检查

检查 reference bit: 1→used recently; clear and leave alone

0→selected candidate for replacement

Will always find a page or loop forever?

Even if all use bits set, will eventually loop around⇒FIFO

二次机会算法： FIFO + Reference bit

所有帧形成一个循环队列。每次内存访问时，访问页的 Reference bit 置为 1。 换出时，循环检查当前帧。如果引用位为 1，则将其置为 0，并跳到下一帧；如果引用位为 0，则替换该页。假如某个页被频繁访问，那么它就不会被替换出去。

增强型二次机会算法: FIFO + 引用位 + 修改位

将页表的页面分为四种类型（引用位，修改位）：

- (0, 0) 最近没有使用也没有修改过 - Best page to replace
- (0, 1) 最近没有使用但曾经被修改过 - Not quite good, need write before replacement
- (1, 0) 最近使用过，但没有被修改过 - Probably be used again soon
- (1, 1) 最近使用过并且修改过 - Probably be used again soon, and need write before replacement

当需要替换时，检查当前页属于哪一类型。替换在最低非空类型中所碰到的第一个页。缺点: 需要多次搜索整个循环队列。

6. LFU 最不经常使用页替换算法

替换具有最小计数的页。定期将计数右移一位，以形成指数衰减的平均使用次数。

7. MFU 最常使用页替换算法

因为具有最小次数的页可能刚刚被替换进来，并且可能尚未使用，所以替换最常使用的。

内存抖动

Thrashing（抖动）：频繁换入换出页

如果一个进程没有足够的页，那么缺页率将较高，这将导致 CPU 利用率低下、操作系统认为需要增加多道程序设计的道数。**策略：操作系统选择挂起一个进程，将其整体换出。**另一种控制抖动的更为直接的方法是设置可接受的缺页率，如果缺页率太低，则回收一些进程的帧；如果缺页率太高，就分给进程一些帧。

Windows 的实现

请求分页存储管理加 clustering：把所缺页的邻近页也调入内存。

Working set minimum：进程在内存中的最小页数；Working set maximum：进程在内存中的最大页数。当系统空闲内存小于门限值时，删减 working set 以恢复足够的内存。此时可以删去拥有页数超过 working set minimum 的那些进程的一部分页数。

其他考虑

预先调页（Prepaging）：将所需要的页一起调入到内存中。降低缺页率，但可能造成浪费。

减小页大小 -> 增加页表大小。

- 碎片：小页更好利用内存

- I/O 开销：寻道和延迟时间远大于传输时间，需要小页
- 局部：较小页允许每个页更精确的匹配程序局部

  增加页大小：
- 不是所有的应用程序都要求一个较大的页大小，会导致碎片的增加。
- 提供多种页大小

理想情况下，每个进程的工作集存储在 TLB 中。否则有很高的缺页率。

页面交换区：专用的磁盘区域用于保存被淘汰的页面内容，采用交换区映射表管理。

锁定主存页：某些页面在进行I/O操作时不能被替换。

小结

虚拟内存技术允许执行一个进程，它的逻辑地址空间比物理空间大。虚拟内存技术提高了多道程序程度， CPU 利用率和吞吐量。常用的页替换算法包括： FIFO (Belady 异常)、最优、 LRU、计数器等。常用的帧分配策略包括：固定、按比例、按优先级等。如果进程工作集未获得足够内存，将引起抖动。

思考题

例：某计算机有 4 个页表项，每页的装入时间、最后访问时间、访问位 R 、修改位 D 如下所示（时间用时钟点数表示），则分别用 FIFO 、LRU 、第二次机会算法会淘汰哪一页?

Page	Load time	Last visit time	R	D
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

答：FIFO 淘汰 page2 、LRU 淘汰 page1 、二次机会淘汰 page1

例：考虑下面的程序，试举例说明该程序的空间局部性和时间局部性。

```
for (i = 0; i < 20; i++)
     for (j = 0; j < 10; j++)
        a[i] = a[i] × j;
```

答：当数组元素a [0] , a[1] ， …， a [19] 存放在一个页面中时，其空间局部性和时间局部性较好，也就是说，在很短时间内执行都挂行循环乘法程序，而且数组元素分布在紧邻连续的存储单元中。当数组元素存放在不同页面中时，其时间局部性虽相同，但空间局部性较差，因为处理的数组元素分布在不连续的存储单元中。

3. Solid State Disk (SSD)

传统的机械硬盘 (HDD) 运行主要是靠机械驱动头，包括马达、盘片、磁头摇臂等必需的机械部件，它必须在快速旋转的磁盘上移动至访问位置，至少 95% 的时间都消耗在机械部件的动作上。SSD 却不同机械构造，无需移动的部件，主要由主控与闪存芯片组成的 SSD 可以以更快速度和准确性访问驱动器到任何位置。传统机械硬盘必须得依靠主轴主机、磁头和磁头臂来找到位置，而 SSD 用集成的电路代替了物理旋转磁盘，访问数据的时间及延迟远远超过了机械硬盘。SSD 有如此的"神速"，完全得益于内部的组成部件：主控--闪存--固件算法。

主控--闪存--固件算法

主控、闪存及固件算法三者的关系

SSD 最重要的三个组件就是 NAND 闪存，控制器及固件。NAND 闪存负责重要的存储任务，控制器和固件需要协作来完成复杂且同样重要的任务，即管理数据存储、维护 SSD 性能和使用寿命等。

主控

控制器是一种嵌入式微芯片(如电脑中 CPU)，其功能就像命令中心，发出 SSD 的所有操作请求——从实际读取和写入数据到执行垃圾回收和耗损均衡算法等，以保证 SSD 的速度及整洁度，可以说主控是 SSD 的大脑中枢。目前主流的控制器有 Marvell、SandForce、Samsung、Indilinx等。像Marvell各方面都很强劲，代表型号为Marvell 88SS9187/89/90主控，运用在浦科特、闪迪、英睿达等品牌的 SSD 上。

SandForce 的性能也不错，它的特点是支持压缩数据，比如一个 10M 的可压缩数据可能被他压成 5M的写入硬盘，但还是占用 10M 的空间，可以提高点速度，最大的特点是会延长 SSD 的寿命，但是 CPU 占用会高点而且速度会随着硬盘的使用逐渐小幅度降低。代表型号为 SF-2281，运用在包括 Intel、金士顿、威刚等品牌的 SSD 上。

Samsung 主控一般只有自家的 SSD 上使用，性能上也是很强悍的，不会比 Marvell 差多少。目前三星主控已经发展到第五代 MEX，主要运用在三星 850EVO、850PRO 上。

固件算法

SSD 的固件是确保 SSD 性能的最重要组件，用于驱动控制器。主控将使用 SSD 中固件算法中的控制程序，去执行自动信号处理，耗损平衡，错误校正码 (ECC)，坏块管理、垃圾回收算法、与主机设备(如电脑)通信，以及执行数据加密等任务。由于固件冗余存储至 NAND 闪存中，因此当 SSD 制造商发布一个更新时，需要手动更新固件来改进和扩大 SSD 的功能。

开发高品质的固件不仅需要精密的工程技术，而且需要在 NAND 闪存、控制器和其他 SSD 组件间实现完美整合。此外，还必须掌握 NADN 特征、半导体工艺和控制器特征等领域的最先进的技术。固件的品质越好，整个 SSD 就越精确，越高效，目前具备独立固件研发的 SSD 厂商并不多，仅有 Intel/英睿达/浦科特/OCZ/三星等厂商，希望我国能早日解决。

NAND闪存

SSD 用户的数据全部存储于 NAND 闪存里，它是 SSD 的存储媒介。SSD 最主要的成本就集中在 NAND 闪存上。NAND 闪存不仅决定了 SSD 的使用寿命，而且对 SSD 的性能影响也非常大。NAND 闪存颗粒结构及工作原理都很复杂，接下来我们会继续推出系列文章来重点介绍闪存，这里主要来了解一下大家平常选购 SSD 经常接触到的 SLC、MLC 及 TLC 闪存。

三种闪存状态

这几年 NAND 闪存的技术发展迅猛同，从企业级标准的 SLC 闪存到被广泛运用在消费级 SSD 上的 MLC 闪存再到目前正在兴起的 TLC 闪存，短短时间里，我们看到 NAND 技术显著进步。对 SLC、MLC 及 TLC 闪存怎么理解呢？简单来说，NAND 闪存中存储的数据是以电荷的方式存储在每个 NAND 存储单元内的，SLC、MLC及 TLC 就是存储的位数不同。

单层存储与多层存储的区别在于每个 NAND 存储单元一次所能存储的 “位元数”。SLC(Single-Level Cell) 单层式存储每个存储单元仅能储存 1bit 数据，同样，MLC(Multi-Level Cell) 可储存2bit数据，TLC(Trinary-Level)可储存3bit数据。一个存储单元上，一次存储的位数越多，该单元拥有的容量就越大，这样能节约闪存的成本，提高 NAND 的生产量。但随之而来的是，向每个单元存储单元中加入更多的数据会使得状态难以辨别，并且可靠性、耐用性和性能都会降低。

一颗 NAND 芯片是32G的容量，也就是说 128G 的 SSD，内部是 5 个 NAND 芯片做并行读取。读写的时候都依靠主控芯片来做控制，主控芯片还要对所有的 NAND 闪存芯片做磨损平衡，保证这帮兄弟要挂一起挂，假如是 256G 的，内部是 8 通道，并发的读写能力比 128G 会有了接近 50% 的提升。至于再大的容量，因为存储容量过大，导致主控芯片保存的地址映射表过大，性能可能会出现持平甚至下降的情况。

固态硬盘是无需移动的固态电子元件，可以直接进行数据读取。具体量化的概念就是，一台笔记本电脑在使用传统硬盘时可能需要等待 36 秒才能让操作系统完成启动，而现在如果使用固态硬盘，则只需等待不到 9 秒钟。基于这种高性能硬盘的协同工作，CPU 的运行效率也会提高，其节省能耗的优点足以使电池延长 15% 的寿命，而且它更加抗震和轻便。而传统硬盘在读取及写入数据的时候，硬盘磁头需要花费时间转动并找到数据所在的位置。

SSD 的回收

一块刚买的 SSD，你会发现写入速度很快，那是因为一开始总能找到可用的 Block 来进行写入。但是，随着你对 SSD 的使用，你会发现它会变慢。原因就在于 SSD 写满后，当你需要写入新的数据，往往需要做上述的垃圾回收：把若干个 Block 上面的有效数据搬移到某个 Block，然后擦掉原先的 Block，然后再把你的 Host 数据写入。这比最初单纯的找个可用的 Block来写耗时多了，所以速度变慢也就可以理解了。

4. 外部存储管理

存储介质

介质种类：磁盘，磁带，光盘。

物理块：在文件系统中，文件的存储设备常常划分为若干大小相等的物理块。同时也将文件信息划分成相同大小的逻辑块，所有块统一编号。以块为单位进行信息的存储、传输和分配。

磁带：永久保存大容量数据的顺序存取设备。前面的物理块被存取访问之后，才能存取后续的物理块的内容。存取速度较慢，主要用于后备存储，或存储不经常用的信息，或用于传递数据的介质。

硬磁盘的性质

独立可寻址元素：扇区

操作系统总是将扇区组按块为单位一起传输。磁盘可以直接访问它所包含的任何给定信息块（随机访问），可以按顺序或随机地访问任何文件。磁盘可以在原地重写：可以从磁盘中读取/修改/写入一个块。

一些数据：

每个表面有 500 到 20,000 多个磁道，每个轨道有 32 到 800 个扇区。**扇区是可以读取或写成的最小单元。**

分区位记录：

扇区的密度恒定，这意味着外部轨道上有更多扇区、速度随轨道位置而变化。

磁盘

磁盘：直接（随机）存取设备，存取磁盘上任一物理块的时间不依赖于该物理块所处的位置。信息记录在磁道上，多个盘片，正反两面都用来记录信息，每面一个磁头。所有盘面中处于同一磁道号上的所有磁道组成一个柱面。

磁盘物理地址形式：磁头号（盘面号）、磁道号（柱面号）、扇区号。磁盘系统由磁盘本身和驱动控制设备组成，实际存取读写的动作过程是由磁盘驱动控制设备按照主机要求完成的。

（1）一次访盘请求：读/写，磁盘地址（设备号，柱面号，磁头号，扇区号），内存地址（源/目）。

（2）完成访盘过程的三个动作：

1. 寻道（时间）：磁头移动定位到指定磁道。
2. 旋转延迟（时间）：等待指定扇区从磁头下旋转经过。
3. 数据传输（时间）：数据在磁盘与内存之间的实际传输。

（3）很多系统允许有些磁盘是可装卸的节省驱动设备成本，增加灵活性和便携性。

（4）硬盘又分为两种：

1. 固定头磁盘：每个磁道设置一个磁头，变换磁道时不需要磁头的机械移动，速度快但成本高。
2. 移动头磁盘：一个盘面只有一个磁头，变换磁道时需要移动磁头，速度慢但成本低

磁盘与硬盘的对比

比较内容	磁盘	内存
最小的写入单元	扇区	字节
原子操作	扇区	字节、字
随机访问时间	5ms	50ns
顺序访问时间	200MB/s	200-1000MB/s
成本	0.002	0.1
数据稳定性	稳定	易失性

硬盘/文件系统性能分析

指标：响应时间, 吞吐

导致延迟的因素

- 软件路径（可以通过队列进行松散建模）
- 硬件控制器
- 物理磁盘介质

当利用率接近 100% 时，会导致延迟大幅增加。

柱面：同一磁臂位置的磁带集合

读/写数据是一个三阶段的过程

- **寻道时间：**将磁臂放置在适当的轨道上（进入适当的柱面）
- **旋转延迟：**等待所需的扇区在读写头下旋转的时间
- **传输时间：**在读写头下传输位块（扇区）的速度

磁盘延迟 = 排队时间 + 控制器时间 + 寻道时间 + 旋转延迟 + 传输时间

一些数据

工业级别的寻道时间

通常在 8ms 到 12ms 的范围内，因为磁盘参考的位置可能只有宣传数量的 25% 到 33%

旋转延迟

大多数磁盘以 3,600 至 7200 RPM 旋转（部分高达15,000RPM或更多），每转分别约 16 ms 至 8 ms。对所需信息的平均旋转延迟是磁盘旋转周期的一半：8 ms 对应 3600RPM，4 ms 对应 7200RPM。

传输时间与以下因素有关

- 传输尺寸（通常是一个扇区）：每个扇区512B - 1KB
- 转速：3600RPM 到 15000RPM
- 记录密度：每英寸轨道上的 bit 数
- 直径：范围从 1 英寸到 5.25 英寸

典型值：2 到 50 MB/s

控制器的时间取决于控制器的硬件。

成本每年下降两倍（自1991年以来）

典型计算

假设：忽略排队和控制器时间。平均查找时间为 5 ms，平均旋转延迟为 4 ms，传输速率为 4MByte/s，扇区大小为1KByte。

- 对于磁盘上的随机位置，总延时为寻找(5 ms) + 旋转延迟（4 ms）+ 传输（0.25 ms）。总耗时为 10 ms ，对应 100KByte/s
- 对于柱面上的随机位置，总延时为旋转延迟（4 ms）+ 传输（0.25 ms）。总耗时为 5 ms ，对应 200KByte/s
- 对于同一磁道上的下一位置，总延时为传输（0.25 ms）。总耗时为 0.25 ms ，对应 4MByte/s

这是有效地使用磁盘的关键。)是为了最小化搜索延迟和旋转延迟。

一些讨论

制造商如何选择磁盘扇区的大小?

每个扇区之间需要间隔 100-1000 位，以允许系统测量磁盘旋转的速度，并容忍轨道长度的微小（热）变化。如果扇区大小只有 1B，则磁盘利用率只有 1%；如果扇区大小为 1KB，则磁盘利用率约为 90%，传输速率为 100KB/s；如果扇区大小为 1MB，则磁盘利用率基本上为 100%，传输速率为 4MB/s。

磁盘故障

提升鲁棒性

故障避免：通过硬件结构来防止故障的发生

容错性：防止错误演化成故障，通常是通过冗余来完成的

错误消除：通过验证（校验码）来消除潜在的错误

错误预测：估计错误的存在、产生和后果

磁盘的访问优化

循环排序

按照**数据的分布**对输入/输出请求进行排序，提高处理的效率。

假设每个磁道上保存 4 个记录（块），磁盘旋转速度是 20ms/转。当前刚刚读完记录 3。如果收到如下请求序列：读记录 4、读记录 3、读记录 2、读记录 1，则如何安排输入/输出顺序，到达理想的处理性能?

- 按请求次序读取上述记录，总的处理时间：（1/4+3*3/4）* 20 = 50 (ms)
- 按读取记录 1, 2, 3, 4 的顺序，则总的处理时间为：（1/4+1/4 * 4）* 20 = 25（ms）
- 按照读取记录 4, 1, 2, 3 的顺序，则总的处理时间为：（1/4 * 4）* 20 = 20（ms）

优化分布

按照**数据处理的规律**，合理安排其磁盘上的分布，以提高处理的效率。

假设每个磁道上划分为 10 个块，分别存放 A~J 十个逻辑记录，磁盘旋转速度是 20ms/转。如果处理程序读出每个记录后花4ms进行处理，则如何安排逻辑记录的存放位置，以达到理想的处理性能?

```
1 - A   3 - E    5 - I
4 - B   6 - F    8 - J
7 - C   9 - G
10 - D   2 - H
```

交替地址

通过数据的冗余存放来提高访问的速度。

缺点:

- 消耗较多的存储空间
- 数据一致性问题决定其较适合于数据记录总是读出使用的方式

磁盘调度（寻道）

对于移动臂磁盘设备，除了旋转位置外，还有搜索定位的问题（寻道）。常见的磁盘调度算法包括:

先来先服务 FCFS

顾名思义。

电梯调度算法 SCAN

磁盘臂从当前位置开始，先访问请求相对较少的那一侧，向另一端移动并服务请求；当它到达磁盘的另一端时，磁头的移动会反转，服务继续进行。

循环扫描算法 C-SCAN

提供了比 SCAN 更统一的等待时间

磁头从磁盘的一端移动到另一端（自里向外，从小到大），此过程中会随时服务于请求。当它到达另一端时，会立即返回磁盘的开始，中途停止响应请求。

注意到磁盘可被视为一个循环列表，最后一个扇区后就是第一个扇区。

分步扫描算法 F-SCAN

分步电梯调度算法。算法思想是，在扫描的过程中所有新产生的序列放在另外的一个队列中，当访问完当前队列之后，再访问新产生的一个队列。这种算法可以有效防止磁壁粘着现象。

C-LOOK

C-SCAN 的一种版本。磁针只到达每个方向上的最后一个请求，然后立即返回磁盘的开始，而不首先一直移动到磁盘的末端。

最短查找时间优先算法 SSTF

从当前磁头位置选择具有最小搜索时间的请求。SSTF 调度是 SJF 调度的一种形式。

可能会导致一些请求的饥饿

算法选择

- SSTF 是常见的，具有自然的吸引力。
- SCAN 和 C-SCAN 在重负载下性能更好。
- 性能取决于请求的数量和类型，磁盘服务的请求可能受到文件分配方法的影响。
- 磁盘调度算法应该是一个单独的操作系统模块——允许在必要时用不同的算法替换它。
- FCFS 与 SCAN 两种算法，单位时间内处理的 I/O 请求多即吞吐量大，但请求的等待时间可能较长。
- 扫描算法适宜于磁盘负载重的系统，但它不分具体情况扫过所有柱面造成性能不够好。没有饥饿。
- 循环扫描算法适应不断有大批量柱面均匀分布的 I/O 请求，且磁道上存放记录数量较大的情况。
- 分步扫描算法使得 I/O 请求等待时间之间的差距最小，吞吐量适中。

存储性能提升小结

磁盘服务：其速度和可靠性成为文件系统性能和可靠性的主要瓶颈，设计文件系统时应尽可能减少磁盘访问次数。

块高速缓存：系统在内存中保存一些块，逻辑上它们属于磁盘，检查所有的读请求，看所需的块是否在高速缓存中。如果在，则可直接进行读操作。否则，首先要将块读到高速缓存，再拷贝到所需的地方，如果高速缓存已满，则需要进行淘汰。

合理分配磁盘空间：分配块时，把有可能顺序存取的块放在一起，最好在同一柱面上，从而减少磁盘臂的移动次数。

磁盘调度：当多个访盘请求在等待时，采用一定的策略，对这些请求的服务顺序调整安排，旨在降低平均磁盘服务时间，达到公平、高效。

1. 公平：一个I/O请求在有限时间内满足。
2. 高效：减少设备机械运动所带来的时间浪费。
3. 磁盘调度考虑的问题：一次访盘时间 = 寻道时间+旋转延迟时间+存取时间。
    1) 减少寻道时间（活动头磁盘）。
    2) 减少延迟时间（固定头磁盘）。

独立磁盘冗余阵列 (RAID)

基本思路：

用一组较小容量的、独立的、可并行工作的磁盘驱动器组成阵列来代替单一的大容量磁盘，并加入冗余技术，数据能够以多种方式组织和分布存储。

优点：

数据的分布存储，提高了单个 I/O 请求的处理性能

数据的冗余，提高了系统的可靠性

RAID 的特性

1. RAID是一组物理磁盘驱动器，可被操作系统看作是单一逻辑磁盘驱动器;
2. 数据被分布存储在阵列横跨的物理驱动器上;
3. 冗余磁盘的作用是保存奇偶校验信息，当磁盘出现失误时它能确保数据的恢复。

方法

1. 数据划成条块被分布存储在横跨阵列中的所有磁盘上;
2. 逻辑上连续的数据条块，在物理上可被依次存储在横向相邻的磁盘驱动器上;
3. 通过阵列管理软件进行逻辑地址空间到物理地址空间的映射

容错能力

廉价磁盘的冗余阵列

几个较小的磁盘扮演一个大磁盘的作用

可以提高性能

数据在多个磁盘之间传播

对不同的磁盘的并行访问

可以提高可靠性

数据可以保持一些冗余

类型

RAID 0

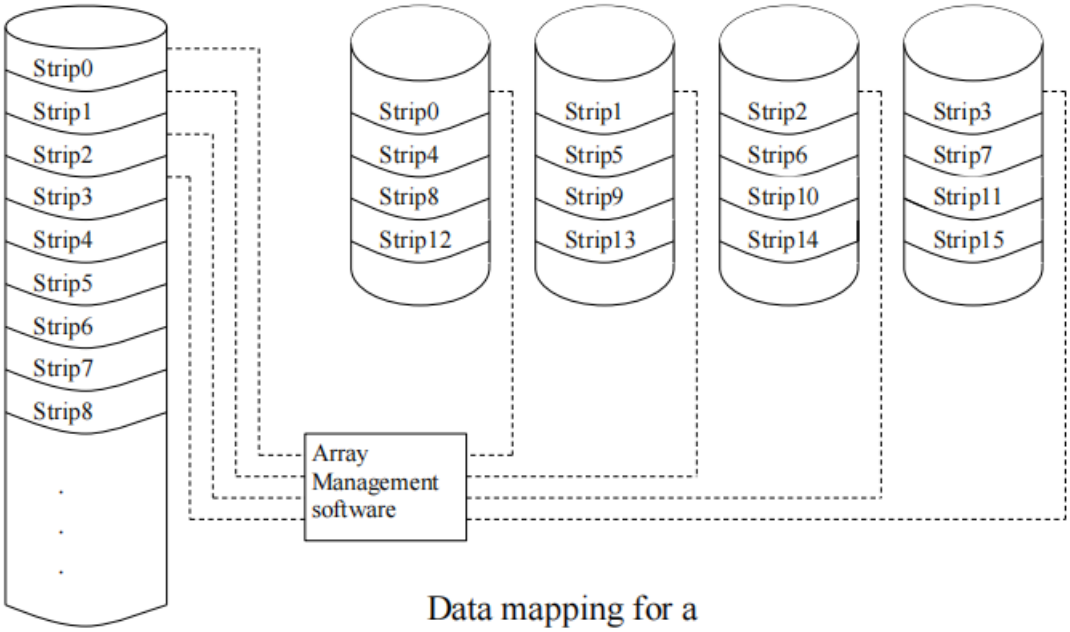
用于提高性能

数据在阵列中存储在磁盘上，以便连续的“条带”存储在不同的磁盘上，使磁盘共享负载，提高吞吐量

- 所有磁盘都可以在并行延迟下工作:
- 更少的排队延迟——每个磁盘有独自的队列

没有冗余

可靠性实际上低于单磁盘（如果阵列中任何磁盘出现故障，则整体故障）



Data mapping for a
RAID Level 0 Array

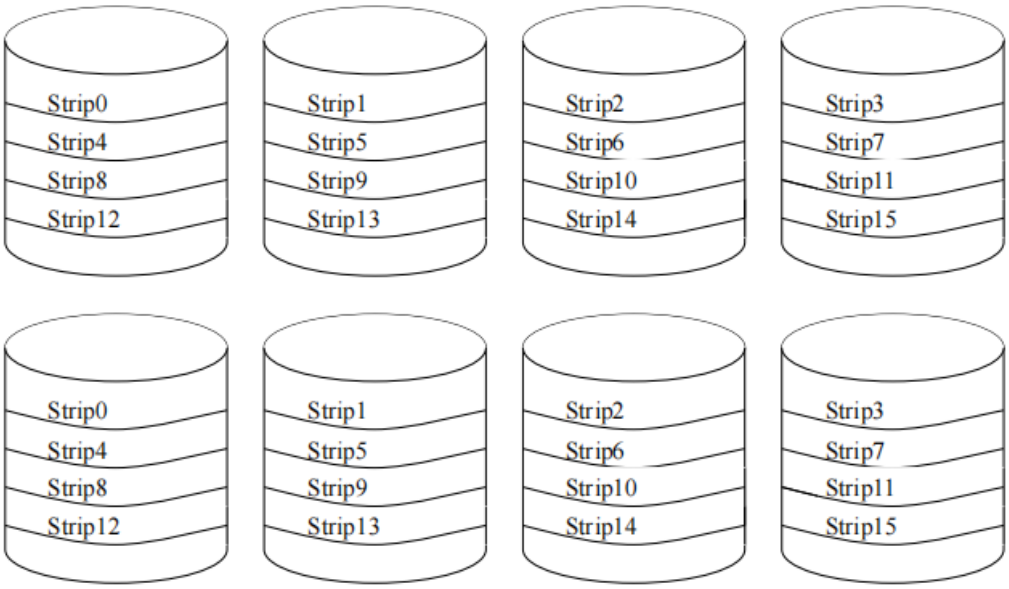
RAID 1

磁盘镜像

将磁盘两两配对，并保持相同的数据。写入必须更新两个磁盘上的副本，读取可以读取两个副本中的任何一个。

提高性能和可靠性

- 单位时间可以做更多的读取.读请求能通过包含相同请求数据中的任何一个磁盘提供服务，其中的一个所化查找和搜索时间最少
- 写操作时，要求改写对应的两个数据子块，可采用并行操作，写操作的性能由并行操作中较慢的一个决定;
- 如果一个磁盘发生故障，它的镜像仍然有数据

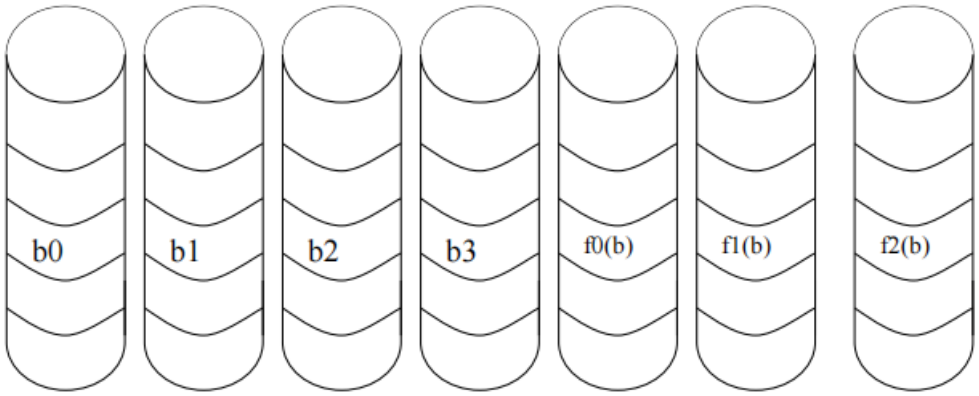


RAID Level 1 (Mirrored)

RAID 2

采用并行存取技术，驱动器的移动臂同步工作，每个磁盘的磁头都在相同位置。纠错码按照横跨的每个数据盘的相应位计算，并存储在多只校验盘的相应位的位置。

校验磁盘的数量与数据盘的多少成比例。



RAID Level 2 (Redundancy through Hamming Code)

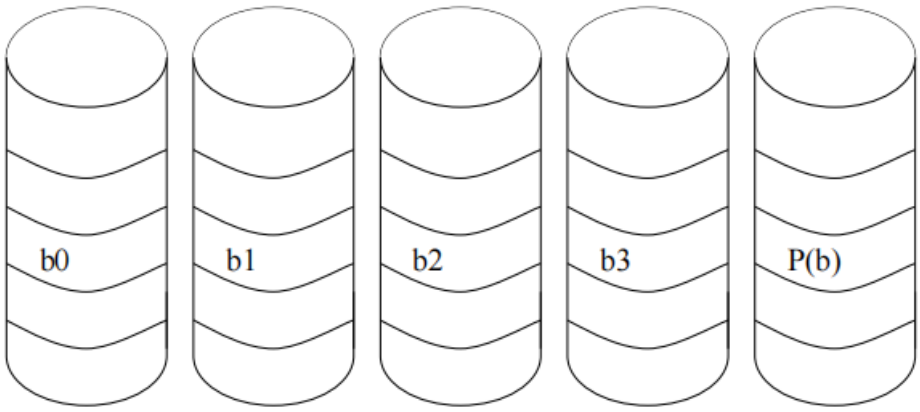
RAID 3

RAID3 仅使用一只冗余盘， 出现故障时，使用奇偶校验盘的信息校验，数据可用剩下的磁盘的信息重新构造，若 X0 到 X3 存放数据，X4 为奇偶盘，对于第 i 位的奇偶校验位可如下计算：

```
X4(i) = X3(i) ⊕ X2(i) ⊕ X1(i) ⊕ X0(i)
```

假定驱动器X1出故障，如果把X4(i)，X1(i)加到上面等式两边，得到

```
X1(i) = X4(i) ⊕ X3(i) ⊕ X2(i) ⊕ X0(i)
```



RAID Level 3 (Bit interleaved Parity)

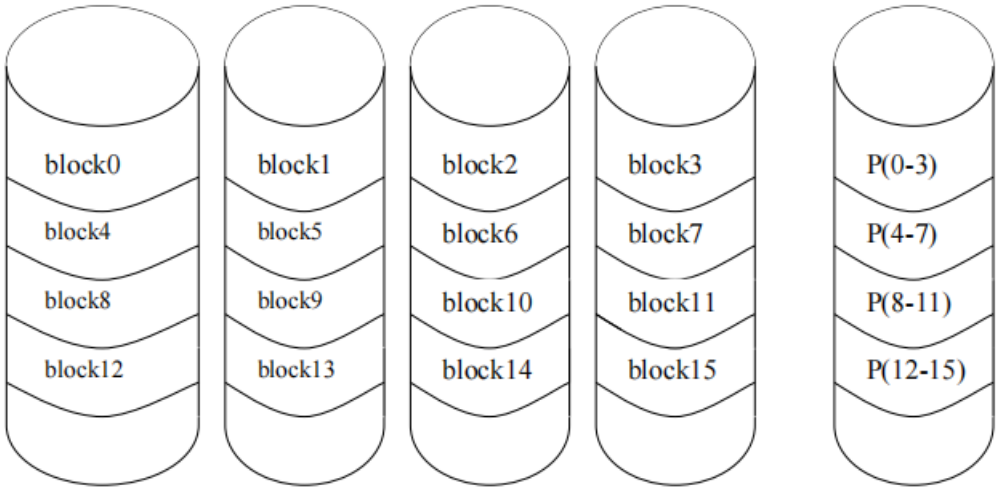
RAID 4

RAID4 和 RAID5 使用独立存取技术，在一个独立存取的磁盘阵列中，每个驱动器都可以独立地工作，所以，独立的I/O请求可以被并行地得到满足。因此独立存取阵列适合于有频繁I/O请求的应用。

每当执行一个小数据量写操作时，阵列管理软件不旦要修改用户数据，而且也要修改对应的奇偶校验位。

块交错奇偶校验

读取只访问数据所在的数据磁盘，写入必须更新数据块及其奇偶校验块。可以从任何一个磁盘上的错误中恢复使用奇偶校验和其他数据磁盘来恢复丢失的数据。

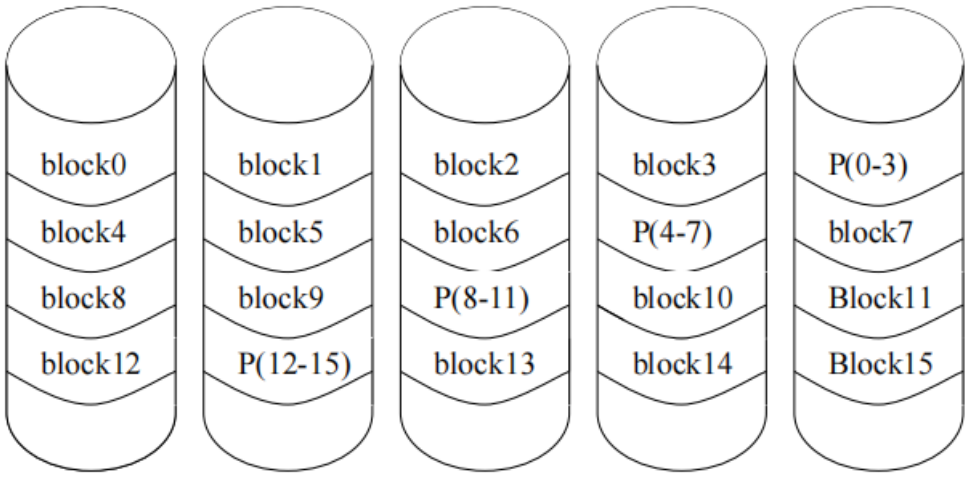


RAID Level 4 (Block level Parity)

RAID 5

RAID5 的奇偶校验码是分布横跨轮转存放在所有的磁盘上，设有 n 个磁盘的阵列，则开头的 n 个奇偶校验码螺旋式地位于 n 个磁盘上，能避免RAID4 发生的奇偶校验盘瓶颈问题。

RAID Level 5 (Block level Distributed parity)



RAID 6

增强型 RAID。RAID6 中设置了专用快速的异步校验磁盘，具有独立的数据访问通路，比低级RAID性能更好，但价格昂贵。

两个不同的（P和Q）检查块。每个保护组有 N-2 个数据块一个校验块另一个检查块（不等于校验）

当两个磁盘丢失时也可以恢复——认为P是 D 块的和、Q是D块的乘积。如果两个块丢失，求解方程，可以恢复两个块的数据。

更多的空间开销（只有 N 中的 N-2 是数据）、更多的写入开销（必须同时更新P和Q）

RAID 7

RAID7 对 RAID6 作了改进，该阵列中的所有磁盘都有较高传输速率，性能优异，但价格也很高。

驱动调度技术

什么是驱动调度?

系统运行时，同时会有多个访问辅助存储器的进程请求输入/输出操作，操作系统必须采用一种调度策略，使其能按最佳的次序执行各访问请求。

调度效率指标：

若干个输入/输出请求服务所需的总时间越少，则系统效率越高。

影响存取访问速度的因素：

- 调度算法（策略），即如何对访问请求进行优化排序
- 信息在辅助存储器上的排列方式
- 存储空间的分配方法

提高磁盘**I/O**速度的一些方法：提前读、延迟写、虚拟盘

用户对外存的要求：方便、效率、安全。

1. 在读写外存时不涉及硬件细节，使用逻辑地址和逻辑操作。
2. 存取速度尽可能快，容量大且空间利用率高。
3. 外存上存放的信息安全可靠，防止来自硬件的故障和他人的侵权。
4. 可以方便地共享，动态扩缩，携带拆卸，了解存储情况和使用情况。
5. 以尽可能小的代价完成上述要求。

外存

光盘：光盘容量大，速度快，价格便宜，可读写光盘驱动器价格贵，写过程较麻烦。光盘的空间结构与磁盘类似。

外存的特点：

（1）容量大，断电后仍可保存信息，速度较慢，成本较低。

（2）两部分组成：驱动部分+存储介质。

（3）种类很多。

（4）外存空间组织与地址、存取方式非常复杂。

（5）I/O过程方式非常复杂。