**操作系统**

**目录**

[2.10 死锁的检测与解除 1](#_Toc13430)

[第3章 存储管理 3](#_Toc24332)

[3.1 存储管理概念 4](#_Toc20137)

[3.2 分区 5](#_Toc21413)

[3.3 虚拟内存 7](#_Toc9070)

[3.4 分段 8](#_Toc18760)

[3.5 分页 10](#_Toc17348)

[3.6 页面置换算法 15](#_Toc19745)

[第4章 设备管理 18](#_Toc20426)

[4.1 I/O重定向 18](#_Toc4100)

[4.2 I/O控制方式 21](#_Toc18790)

[4.3 I/O缓冲 23](#_Toc30461)

[4.4 磁盘调度 25](#_Toc18739)

[4.5 RAID 29](#_Toc18703)

[第5章 文件系统 34](#_Toc4016)

[5.1 文件操作 34](#_Toc11297)

[5.2 文件缓冲 38](#_Toc32002)

[5.3 文件系统概念 40](#_Toc29235)

[5.4 文件存储 43](#_Toc3027)

[5.5 空闲空间管理 48](#_Toc330)

2.10 死锁的检测与解除

死锁检测与解除

**检测死锁不同于预防死锁，不限制资源访问方式和资源申请。OS可以周期性地执行死锁检测例程，检测系统中是否出现环路等待。**

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

1. **撤销进程法（abort）**：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。

2. **进程回退法（rollback）**：让一个或多个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

**哲学家就餐问题(Dining Philosophers Problem)**

之前对于哲学家就餐问题的解决算法可能产生死锁。为了使全部哲学家都能进餐，算法必须避免死锁和饥饿。可行的解决方案是**最多只允许4个哲学家同时进餐厅**，则至少有1个哲学家可以拿到两根筷子进餐。进餐完毕后放下筷子，其他哲学家就可进餐，这样就不会出现死锁和饥饿。

|  |
| --- |
| **范例：哲学家就餐问题** |
| *Semaphore* room = 4;  *Process* phiosopher(i) {      do {          wait(room);                     // 第5位哲学家将被阻塞          wait(chopstick[i]);             // 申请左筷子          wait(chopstick[(i + 1) % 5]);   // 申请右筷子          // eat          signal(chopstick[(i + 1) % 5]); // 释放右筷子          signal(chopstick[i]);           // 释放左筷子          signal(room);                   // 唤醒被阻塞的哲学家      } while(true);  } |

**第3章 存储管理**

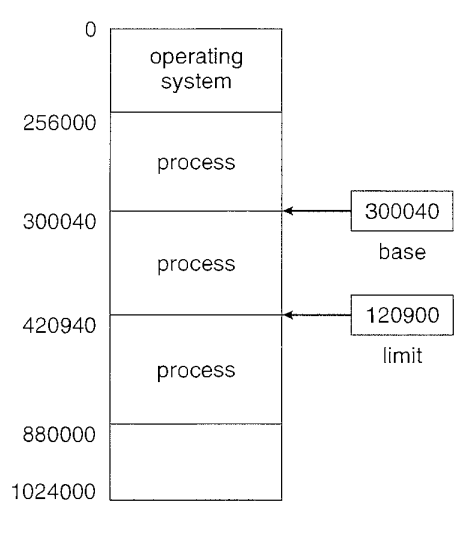
**3.1 存储管理概念**

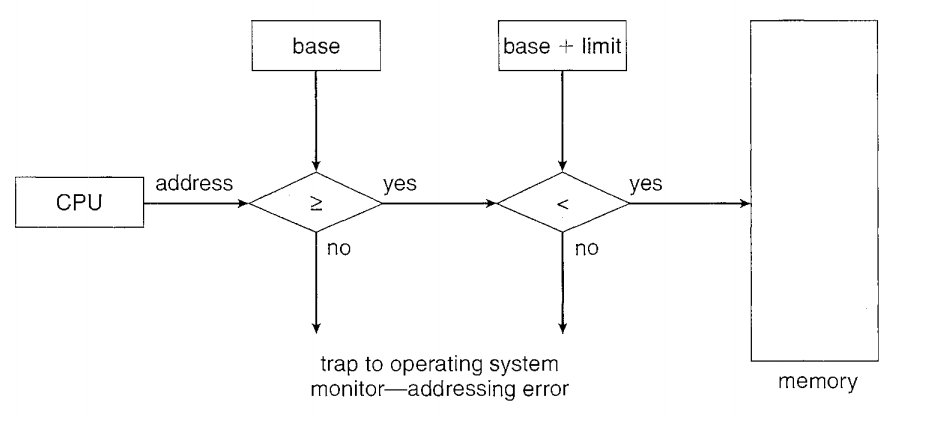
**存储管理概念**

进程调度算法能提高CPU的使用率和计算机对用户的响应速度，但是为了实现这一性能改进，必须将多个进程保存在内存中。

CPU所能直接访问的存储器只有内存和寄存器，执行指令以及指令使用的数据必须在这些直接可访问的存储设备上。如果数据不在内存中，那么在CPU使用前必须先把数据移到内存中。

**为了确保每个进程都有独立的内存空间，通过基地址寄存器（base register）和界限地址寄存器（limit register）可以确定进程可访问的合法地址和范围。**

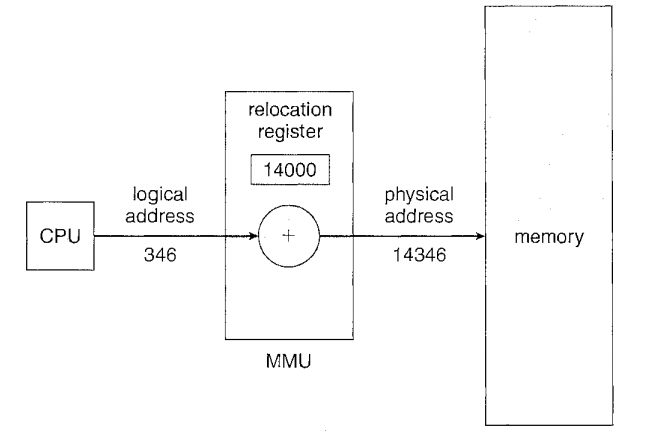




**逻辑/物理地址**

CPU所生成的地址为**逻辑地址（logical address）**，而内存单元所看到的地址为**物理地址（physical address）**。

**用户程序不会看到真正的物理地址**，只有当它作为内存地址时，它才进行相对于基地址寄存器的**重定位**，将逻辑地址转变为物理地址。



**3.2 分区**

**固定分区(Fixed Partition)**

最简单的内存分配方法之一就是**将内存分为多个固定大小的分区，每个分区只能容纳一个进程**。当一个分区空闲时，可以从输入队列中选择一个小于等于分区尺寸的进程调入到空闲分区。当进程终止时，其分区可以被其它进程所使用的。

**固定分区的好处在于实现简单、系统开销小，但是由于分区尺寸和数目固定，系统无法运行大程序，使活动进程的数目受限。同时固定分区会产生内部碎片（internal fragmentation），存储利用率不高。**

**内部碎片的产生是由于无论进程的尺寸有多大，都会占用一个分区的空间，这样就会产生内部碎片。这部分内存在分区内，但又不能被其它进程所使用的。**

**可变分区(Dynamic Partition)**

当有新进程需要内存时，为该进程查找足够大的块，如果找到，则为其分配所需的内存。

这种方法属于**动态存储分配问题**，有三种最为常用的解决方法：

1. **首次适应（first-fit）**：**分配第一个足够大的块**，每次查找从头或者从上一次首次适应结束的位置开始，一旦找到足够大的空间，就分配给进程。

2. **最佳适应（best-fit）**：查找整个列表，**分配最小的足够大的块**，这种方法可以产生最小的外部碎片。

3. **最差使用（worst-fit）**：查找整个列表，**分配最大的块**，这种方法可以产生最大的外部碎片。

**可变分区为进程分配了大小合适的分区**，消除了内部碎片，但是却产生了**外部碎片（external fragmentation）**。随着进程装入和移出内存，空闲内存空间被分为小片段。**当所有总的可用内存之和可以满足请求，但不连续时，就会产生外部碎片问题。**

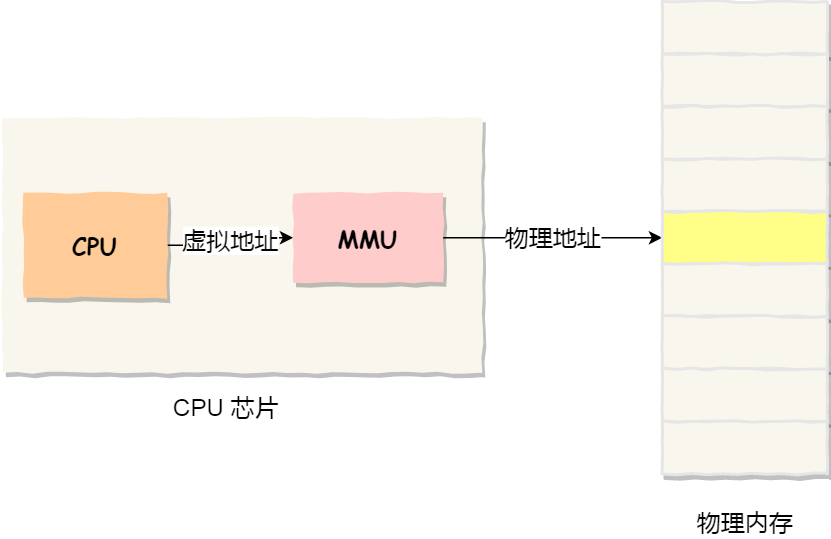
为了使外部碎片得到充分利用，利用**紧凑技术（compaction）可以将内存中的所有空闲分区拼接成一个较大的空闲分区**。即系统可以把内存中的所有进程移到内存的某一段，那么所有空闲分区将会移到内存的另一端。

**3.3 虚拟内存**

**虚拟内存(Virtual Memory)**

普通的内存管理策略都需要**在进程执行之前将整个进程放在内存中**，虚拟内存技术运行执行进程不必完全在内存中。这种方案的一个显著的优点就是**用户会感觉到系统的内存空间比实际内存大**，这种技术允许程序员不受内存存储的限制，虚拟内存也允许进程很容易地共享文件和地址空间。但是虚拟内存的实现并不容易，如果使用不当可能会极大降低性能。

操作系统为每个进程分配了一套独立的**虚拟地址**，互不干涉，但是每个进程都不能访问物理地址。操作系统会提供一种机制，**将不同进程的虚拟地址和不同内存的物理地址映射起来**。如果程序要访问虚拟地址的时候，由操作系统转换成不同的物理地址，这样不同的进程运行的时候，写入的是不同的物理地址，这样就不会冲突了。进程持有的虚拟地址会通过CPU芯片中的**内存管理单元MMU**来转换变成物理地址。

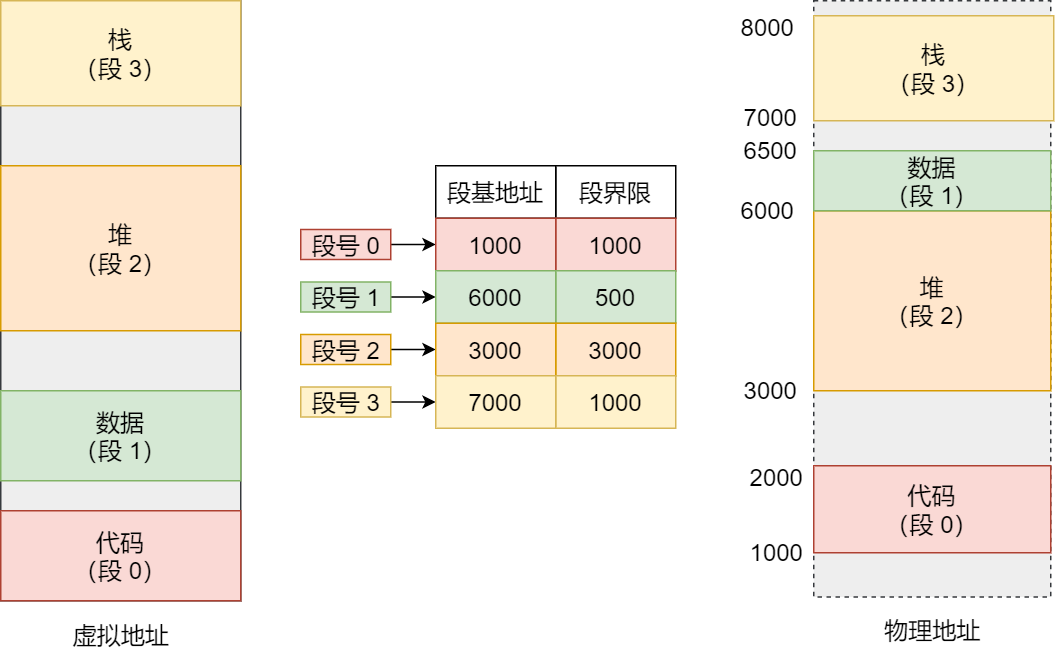


**3.4 分段**

**分段(Segmentation)**

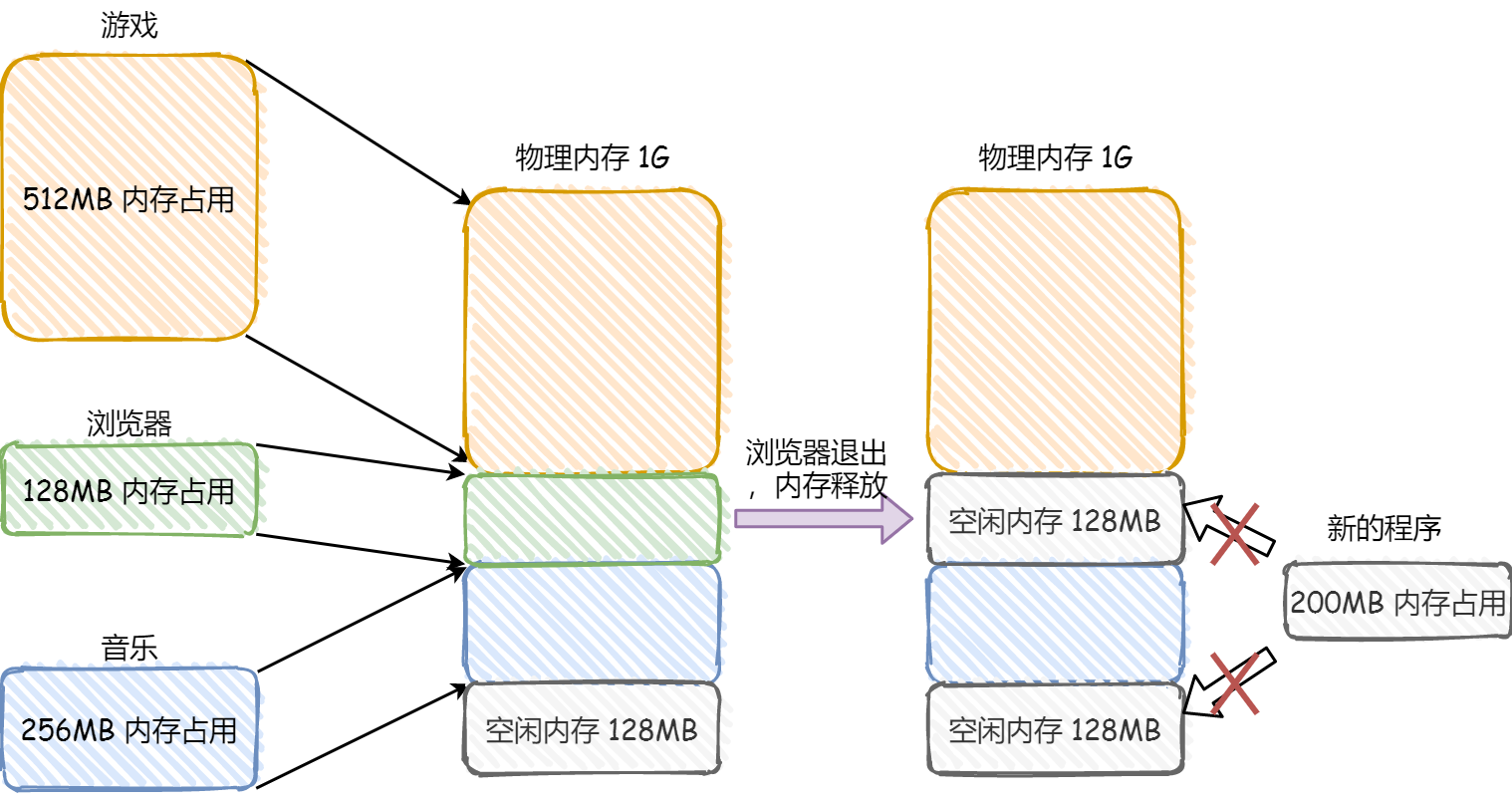
程序是由若干个逻辑分段组成的，如可由代码分段、数据分段、栈段、堆段组成。**分段是一个支持用户视角的内存管理技术，基于模块化开发时，程序员根据需要将进程分割成许多大小不一定相同的段（segment），系统则将物理内存动态划分成许多尺寸不一定相等的分区（partition）。**当一个进程被装入物理内存时，系统将为该进程的每个段独立地分配一个分区，**同一进程的多个段不必存放在连续的多个分区中**。

**段表（segment table）**用于描述进程的分段情况，记载进程的各个段到物理内存中分区的映射情况。段表的基本元素包括**段号、段的物理起始地址和段长度**。



通常，在编译程序时，编译器会自动根据程序来构造段。例如一个C编译器可能会创建代码、全局变量、堆、每个线性采用的栈、标准库函数等一系列段。在编译时链接的库可能被分配不同的段。加载程序会装入所有这些段，并为它们分配段号。

分段解决了程序本身不需要关心具体的物理内存地址的问题，但它也有一些不足之处，如**内存碎片**和**内存交换的低效率**问题。



这里的内存碎片的问题共有两处地方：

1. **外部碎片：产生了多个不连续的物理内存，导致新的进程无法被装载。**
2. **内部碎片：程序所有的内存都被装载到了物理内存，但是这个程序有部分的内存可能并不是很常使用，这也会导致内存的浪费。**

解决外部内存碎片的问题就是**内存交换**。

可以把音乐进程占用的256MB内存写到硬盘上，然后再从硬盘上读回到内存里。不过再读回的时候不能装载回原来的位置，而是紧紧跟着那已经被占用了的512MB内存后面。这样就能空缺出连续的256MB空间，于是新的200MB进程就可以装载进来。

对于多进程的系统来说，分段很容易产生内存碎片，内存交换的过程就会产生性能瓶颈。**因为硬盘的访问速度要比内存慢太多了，每一次内存交换，都需要把一大段连续的内存数据写到硬盘上。**

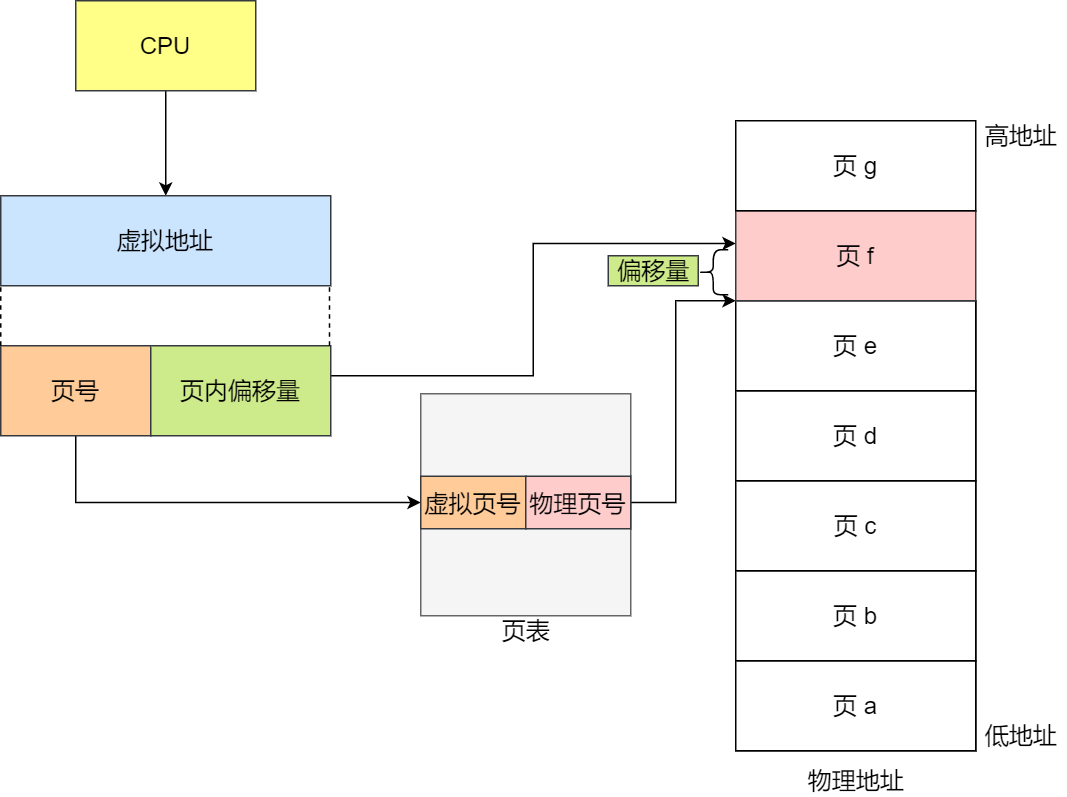
**3.5 分页**

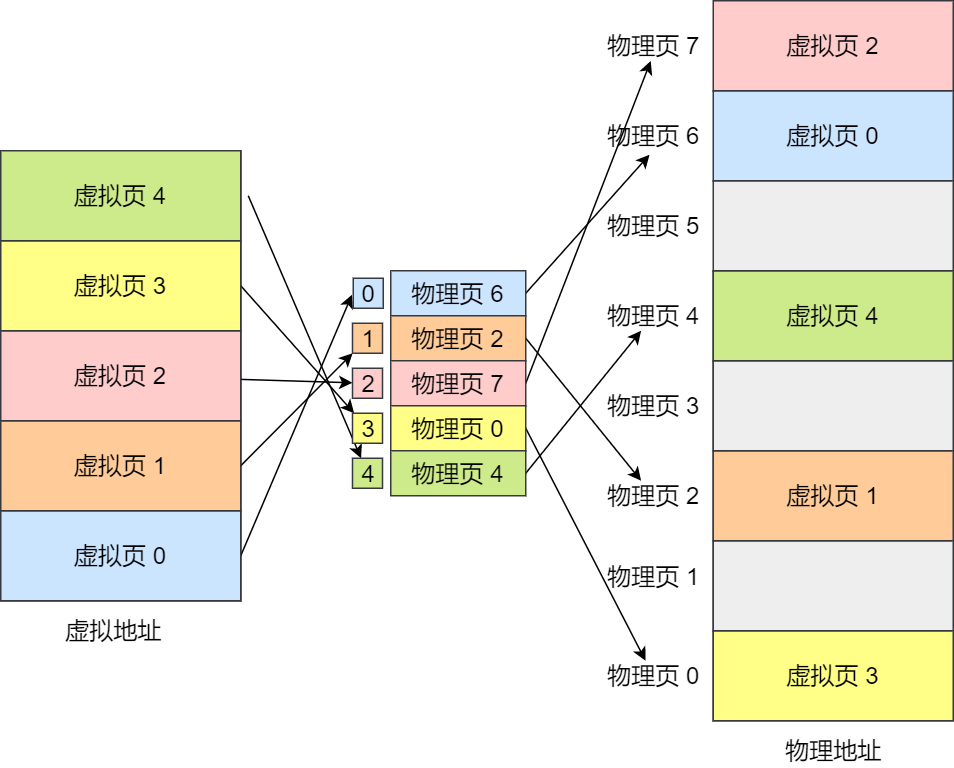
**分页(Paging)**

分页内存管理方案允许进程的物理地址空间可以是**非连续**的。**分页的基本方法是将程序分成等长的小块，称为页（page），同样内存也被分成了和页面同样大小的页框/帧（frame），这样一个页就可以装到一个页框中。**

分页机制把进程散布在不同的页框中，因此在执行程序的时候，需要根据**页表（page table）查找某个页面在内存的某个页框中**，由此完成逻辑地址到物理地址的映射。

每当进程的某一条指令要访问进程内某地址时，其产生的地址都会被分割成**页号**和**偏移量**的形式。CPU根据页号查找页表，找到物理内存中该页对应的页框号用来替换页号，重新拼接成真正的物理地址。



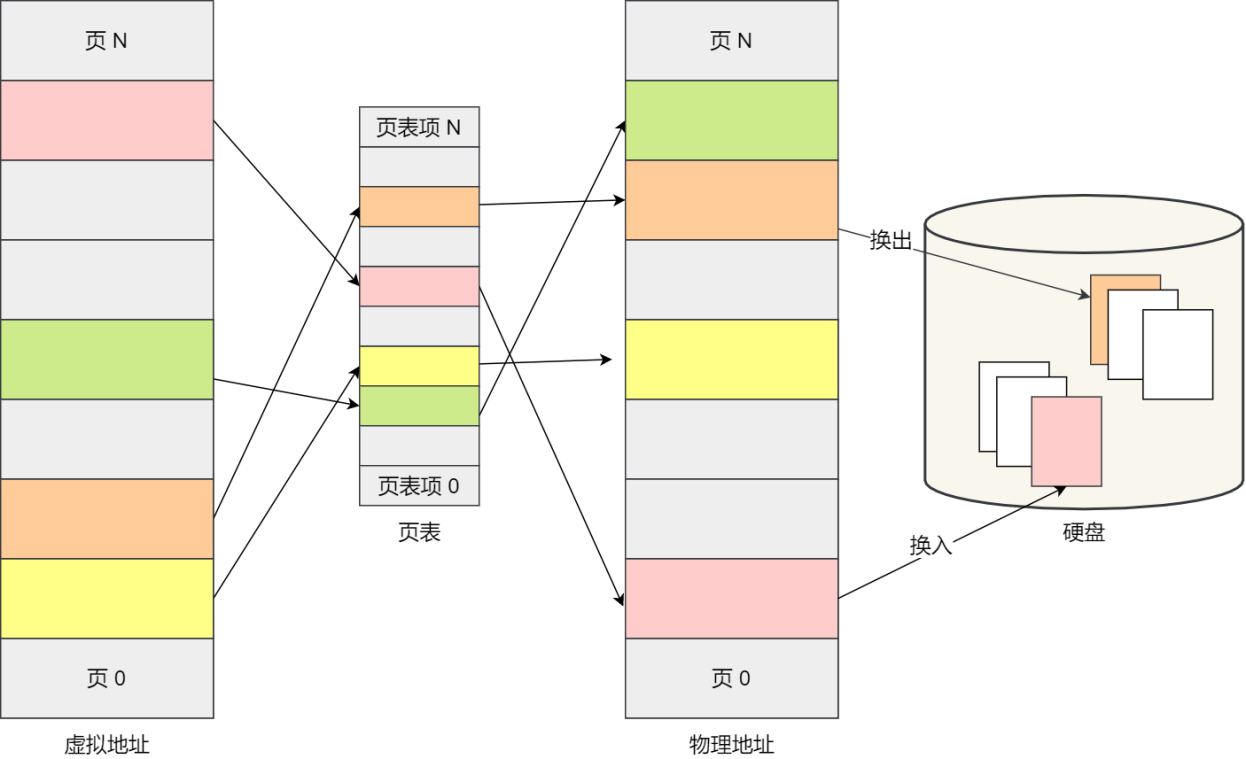


采用分页技术不会产生外部碎片，每个页框都可以分配给需要它的进程。不过，分页会产生**内部碎片**，如果进程所要求的内存并不是页的整数倍，那么最后一个页框就可能用不完。

分页的一个重要特点是**用户视角的内存和实际的物理内存是分离的**，逻辑地址需要通过映射转变成物理地址，这种映射由操作系统所控制，用户是不知道的。

由于内存空间都是预先划分好的，也就不会像分段会产生间隙非常小的内存。同时内存也都是以页为单位释放的，也就不会产生无法给进程使用的小内存。

**如果内存空间不够，操作系统会把其它正在运行的进程中的“最近没被使用”的内存页面暂时换出（swap out）到硬盘上。一旦需要的时候，再换入（swap in）加载进来。因此，一次性写入磁盘的也只有少数的一个页或者几个页，不会花太多时间，内存交换的效率就相对比较高。**

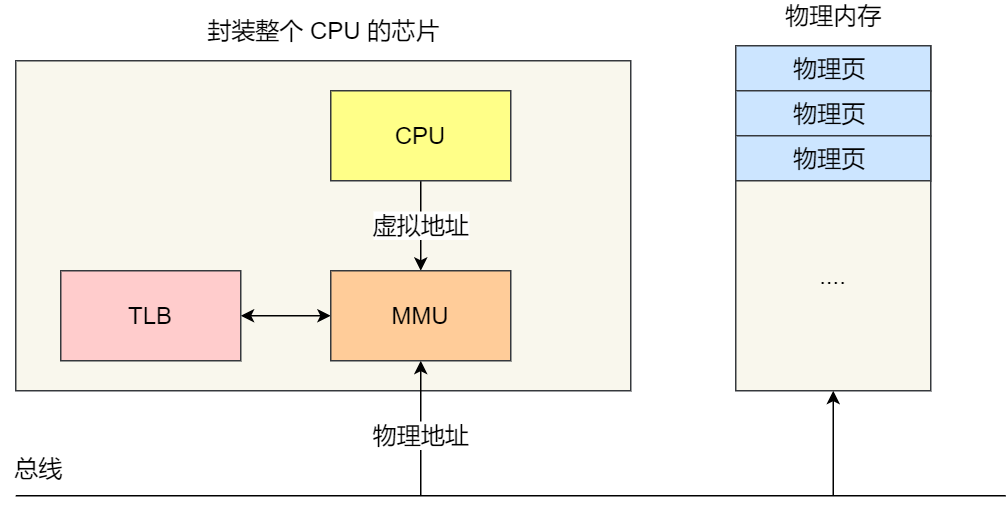


**分页的方式使得在加载程序的时候，不再需要一次性都把程序加载到物理内存中。而是只有在程序运行中，需要用到对应虚拟内存页里面的指令和数据时，再加载到物理内存里面去。**

**TLB (Translation Lookaside Buffer)**

程序是有局部性的，**局部性原理是指CPU访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中**。即在一段时间内，整个程序的执行仅限于程序中的某一部分。相应地，执行所访问的存储空间也局限于某个内存区域。

利用这一特性，可以把最常访问的几个页表项存储到访问速度更快的硬件，于是计算机科学家们就在CPU芯片中加入了一个专门存放程序最常访问的页表项的Cache，通常称为**TLB、页表缓存、快表**等。有了TLB后，那么**CPU在寻址时会先查TLB，如果没找到，才会继续查常规的页表**。



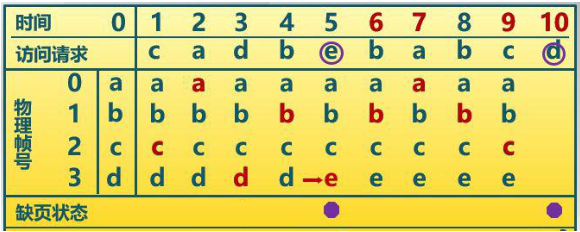
**3.6 页面置换算法**

**页面置换算法**

当出现缺页异常，需调入新页面而内存已满时，置换算法选择被置换的物理页面。页面置换算法的设计目标是尽可能减少页面的调入调出次数，把未来不再访问或短期内不访问的页面调出。

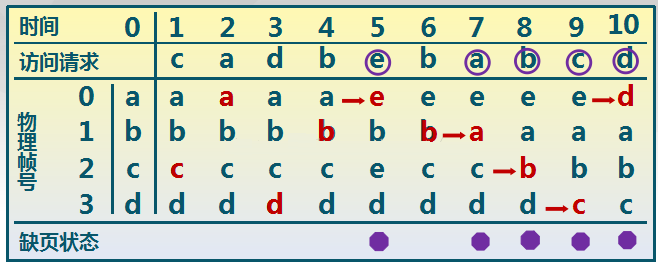
**最优页面置换算法（OPT）**

当一个缺页中断发生时，**把将来最长时间不需要的页面置换**。这只是一种想的情况，在实际的系统中是无法实现的，因为操作系统无法知道每一个页面要等待多长时间以后才会再次被访问。该算法**无法实现**，但是会利用该算法作为参考，评价其它算法。



**先进先出算法（FIFO）**

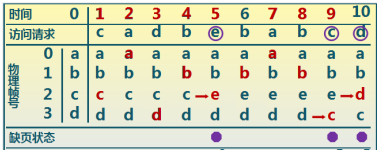
FIFO算法**选择在内存驻留时间最长的页面进行置换**。算法实现需要维护一个**队列**，出现缺页时，选择队头页面进行置换，新页面加到队尾。FIFO算法实现简单，但是性能较差，调出的页面可能是经常访问的。



**最近最久未使用算法（LRU, Least Recently Used）**

**LRU算法是对最优置换算法的近似，以过去推未来。根据程序的局部性原理，在最近一小段时间，如果某些页面被频繁访问，那么在将来的一小段时间内，它们还可能再一次被频繁访问。反之，如果在过去某些页面长时间未被访问，那么将来它们还可能会长时间地得不到访问。因此，LRU算法选择最长时间没有被引用的页面进行置换。**

LRU算法的实现需要计算内存中每个逻辑页面的上一次访问时间，选择上一次使用到当前时间最长的页面。该算法可能达到最优的效果，但是维护这样的访问链表开销比较大。

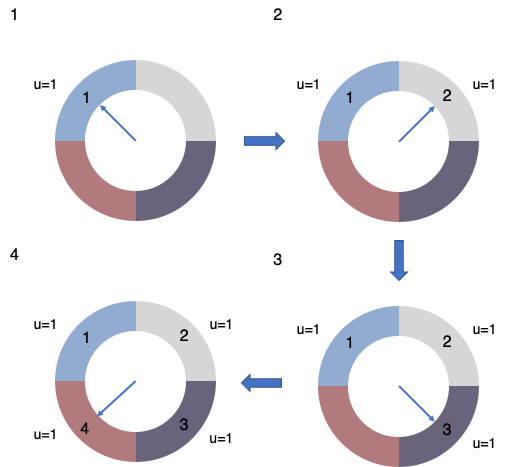


**时钟页面置换算法（Clock）**

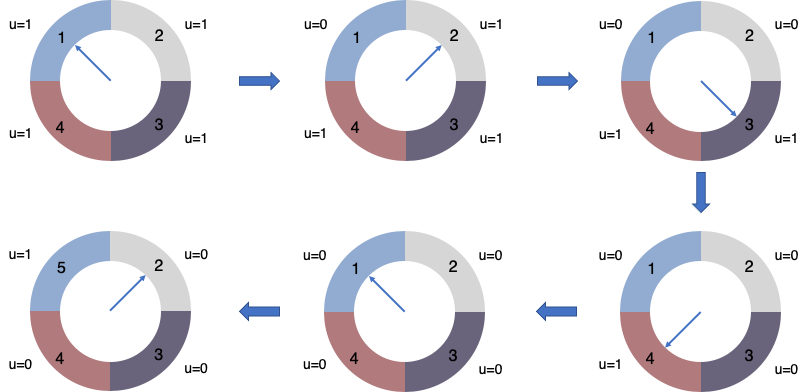
**Clock页面置换算法是LRU的近似和对FIFO的改进。算法需要用到页表项的访问位（access bit），当一个页面被装入内存时，把该位初始化为0，如果这个页被访问时把它置为1。把各个页面组织成环形链表，把指针指向最老的页面（最先进来）。**

**当发生一个缺页中断，如果指针所指向的最老的页面的访问位为0，则立即淘汰。若访问位为1，则把该位置置为0，然后指针往下移动一格。如此下去，直到找到被淘汰的页面，然后把指针移动到它的下一格。**

例如按照{1, 2, 3, 4}的顺序访问页面，则缓冲池会以这样的一种顺序被填满：



访问结束后，缓冲池已经被填满了。此时如果按照{1, 2, 3, 4, 5}的顺序访问，那么在访问{1, 2, 3, 4}的时候是可以直接命中缓存返回的，但是访问5的时候，因为缓冲池已经满了，所以要进行一次逐出操作。



**第4章 设备管理**

**4.1 I/O重定向**

**标准I/O**

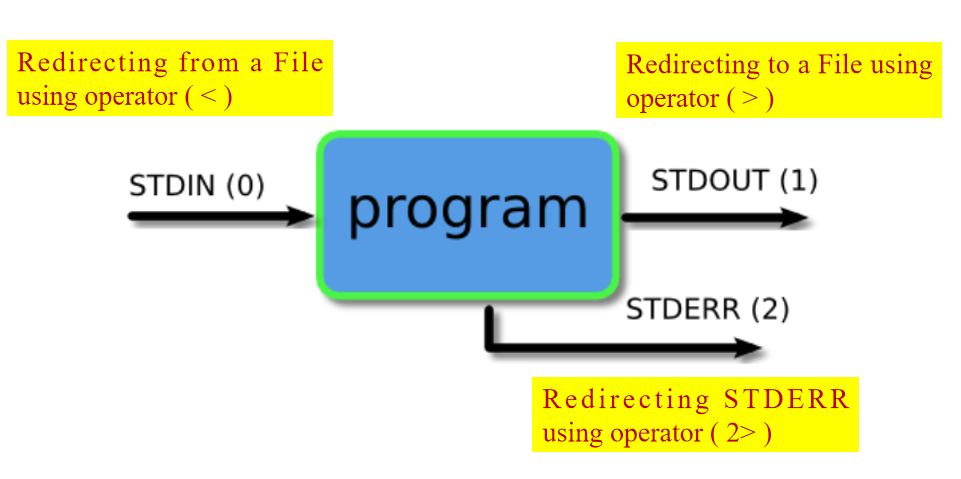
程序对读入的数据进行处理，再输出数据。**数据的输入(Input)和输出(Output)简称为I/O，在没有指定输入输出的情况下，默认为标准输入和标准输出。**

打开的文件都有一个**文件描述符（fd: file descriptor）**，表现为一个数字：

**1. 标准输入stdin（键盘）：fd = 0。**

**2. 标准输出stdout（显示器）：fd = 1。**

**3. 标准错误输出stderr（显示器）：fd = 2。**



|  |  |
| --- | --- |
| **范例：标准I/O** | |
| #include <stdio.h>  *int* main(*int* *argc*, *char* \**argv*[]) {  *int* num;      fscanf(stdin, "%d", &num);      fprintf(stdout, "[stdout] num = %d\n", num);      fprintf(stderr, "[stderr] This is an error message.\n");      return 0;  } | |
| **运行结果** | 123  [STDOUT] num = 123  [STDERR] This is an error message. |

|  |  |
| --- | --- |
| **范例：文件I/O** | |
| #include <stdio.h>  #include <stdlib.h>  *void* writeFile(const *char* \**filename*) {      FILE \*fp = fopen(*filename*, "w");      if(!fp) {          fprintf(stderr, "File open failed.\n");          exit(1);      }  *char* \*name = "小灰";  *int* age = 17;  *double* height = 182.3;      fprintf(fp, "姓名：%s\n年龄：%d\n身高：%.2f\n",                  name, age, height);      fclose(fp);  }  *void* readFile(const *char* \**filename*) {      FILE \*fp = fopen(*filename*, "r");      if(!fp) {          fprintf(stderr, "File open failed.\n");          exit(1);      }  *char* name[32];  *int* age;  *double* height;      fscanf(fp, "姓名：%s\n年龄：%d\n身高：%lf\n",                  name, &age, &height);      printf("name: %s\n", name);      printf("age: %d\n", age);      printf("height: %.2f\n", height);      fclose(fp);  }  *int* main(*int* *argc*, *char* \**argv*[]) {      const *char* \*filename = "info.txt";      writeFile(filename);      readFile(filename);      return 0;  } | |
| **运行结果** | name: 小灰  age: 17  height: 182.30 |

**I/O重定向(I/O Redirection)**

**I/O重定向就是改变标准输入与输出的默认位置**。标准输入默认是键盘，通过改成其它输入，就是输入重定向，例如从文本文件里输入。标准输出默认是显示器，通过改成其它输出，就是输出重定向，例如输出到文件。

**输出重定向用“>”表示，若文件不存在，则创建；若文件已存在，则覆盖。使用“>>”时若文件不存在，则创建，若文件已存在，则追加。**

**错误输出重定向用“2>”和“2>>”表示。**

**输入重定向用“<”表示，但是在输入重定向中“<<”可不是表示输入追加。**

|  |  |
| --- | --- |
| **范例：I/O重定向** | |
| #include <stdio.h>  *int* main(*int* *argc*, *char* \**argv*[]) {  *int* data;      scanf("%d", &data);      printf("data = %d\n", data);      return 0;  } | |
| **编译** | gcc -Wall io\_redirection.c -o io\_redirection |
| **命令行** | Windows cmd: io\_redirection < info.txt > output.txt  Linux terminal: ./io\_redirection < input.txt > output.txt |
| **input.txt** | 12345 |
| **output.txt** | num = 12345 |

**4.2 I/O控制方式**

**I/O设备**

现代计算机系统总是配有各种类型的外部设备，除了显示器、键盘、打印机、磁带、磁盘外，又出现了光盘、绘图仪、图形数字化仪、鼠标器、激光打印机、声音输入输出设备等，**种类繁多**。不同外设之间的**差异较大**，因此**I/O性能经常成为系统的瓶颈**。

操作系统设备管理的目标包括：

1. **向用户提供使用外部设备的方便、统一的接口**，按照用户的要求和设备的类型，控制设备工作，完成用户的输入输出请求。方便是指用户能独立于具体设备的复杂物理特性而方便地使用设备；统一是指对不同设备尽量能统一操作方式。**方便和统一要求对用户屏蔽实现具体设备I/O操作的细节**，呈现给用户的是一种性能理想化的、操作简便的逻辑设备。系统的这种性能亦称为设备的**独立性（设备无关性）**。

2. 充分利用**中断技术**、**通道技术**和缓冲技术，**提高CPU与设备、设备与设备间的并行工作能力**，充分利用设备资源，提高外部设备的使用效率。

3. 设备管理就是要保证在**多道程序环境**下，当多个进程竞争使用设备时，按照一定的**策略分配和管理设备**，以使系统能有条不紊地工作。

**I/O控制方式**

早期，计算机设计者没有将CPU的执行与I/O操作分开，甚至大多数人认为输入输出与计算的时间应该是同一数量级。后来，他们意识到，**CPU要比I/O操作速度高几个数量级**。于是，硬件和软件设计师开始寻找一种技术，**使CPU计算可以不用等待I/O操作而持续执行**。

I/O控制方式包括：

1. **程序控制I/O（Programmed I/O）**：处理器代表一个进程给I/O模块发送一个I/O命令，该进程进入忙等待（busy waiting），直到操作完成。

2. **中断驱动I/O（Interrupt I/O）**：处理器代表进程向I/O发送命令，如果来自进程的I/O指令是非阻塞的，那么处理器继续执行进程的后续指令。如果I/O指令时阻塞的，那么处理器执行来自操作系统的指令，将当前进程设置为阻塞态并且调度其它进程。

3. **直接存储器访问（DMA）**：DMA模块控制内存和I/O模块之间的数据交换。为传送一块数据，处理器给DMA模块发送请求，只有当整个数据块传送结束后，它才能被中断。

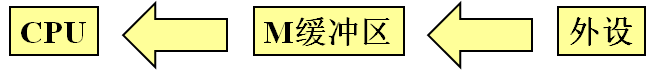
**4.3 I/O缓冲**

**I/O缓冲(I/O Buffering)**

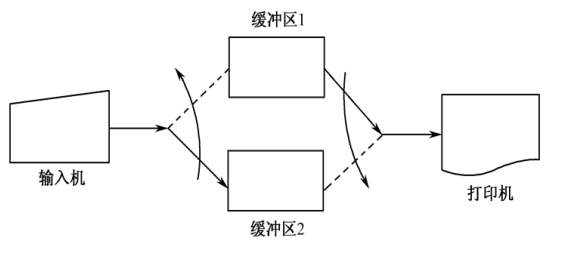
在设备管理中，为了缓和CPU与I/O设备速度不匹配的矛盾，**提高CPU与I/O设备的并行性、减少对CPU的中断频率**，在I/O设备与处理机交换数据时都用到了缓冲区。

缓冲的种类包括：

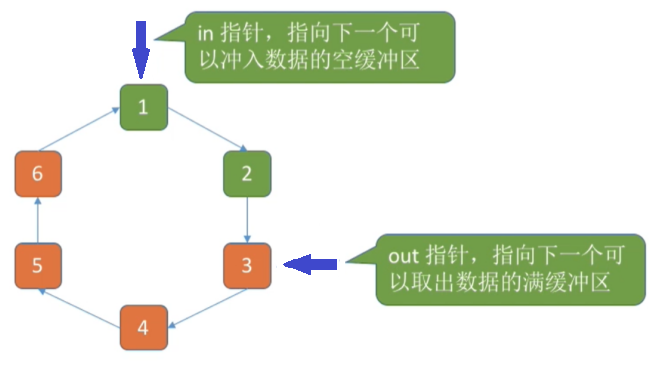
1. **单缓冲**：输入时通道先将数据送入缓冲区，CPU从缓冲区取数据处理。通道再送入后续数据，如此反复直到输入完成。输出情形正好相反。由于缓冲区属于互斥区，所以单缓冲并不能明显改善CPU与外部设备的并行性。



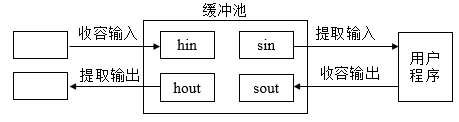
2. **双缓冲**：分别设置输入缓冲区和输出缓冲区，CPU和通道可以分别访问两个缓冲区，即在CPU访问一个缓冲区的同时，通道可以访问另一个缓冲区。双缓冲只是一种说明设备和设备、CPU和设备并行操作的简单模型，并不能用于实际系统中的操作。因为计算机系统中的外围设备较多，另外双缓冲也很难匹配设备和处理机的处理速度。现代计算机系统一般使用多缓冲或缓冲池结构。



3. **多缓冲**：把多个缓冲区连接起来组成两部分，一部分专门用于输入，另一部分专门用于输出的缓冲结构。常组织成循环队列的结构，也称为循环缓冲。



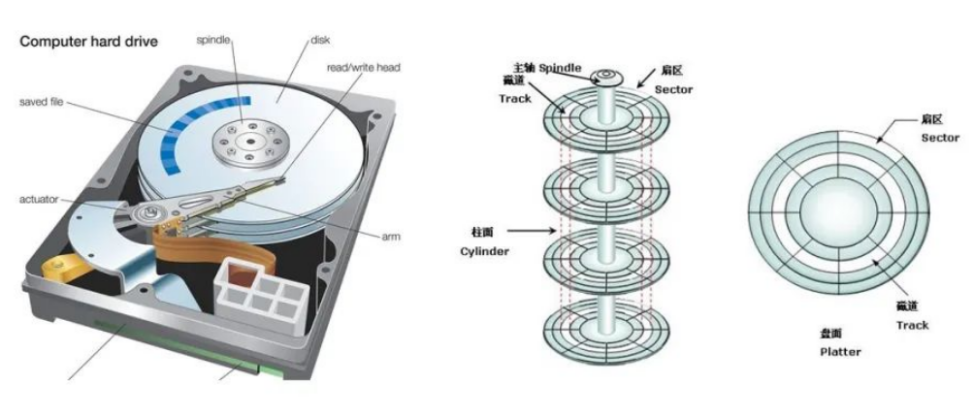
4. **缓冲池**：把多个缓冲区连接起来统一管理，既可用于输入又可用于输出的缓冲结构。



**4.4 磁盘调度**

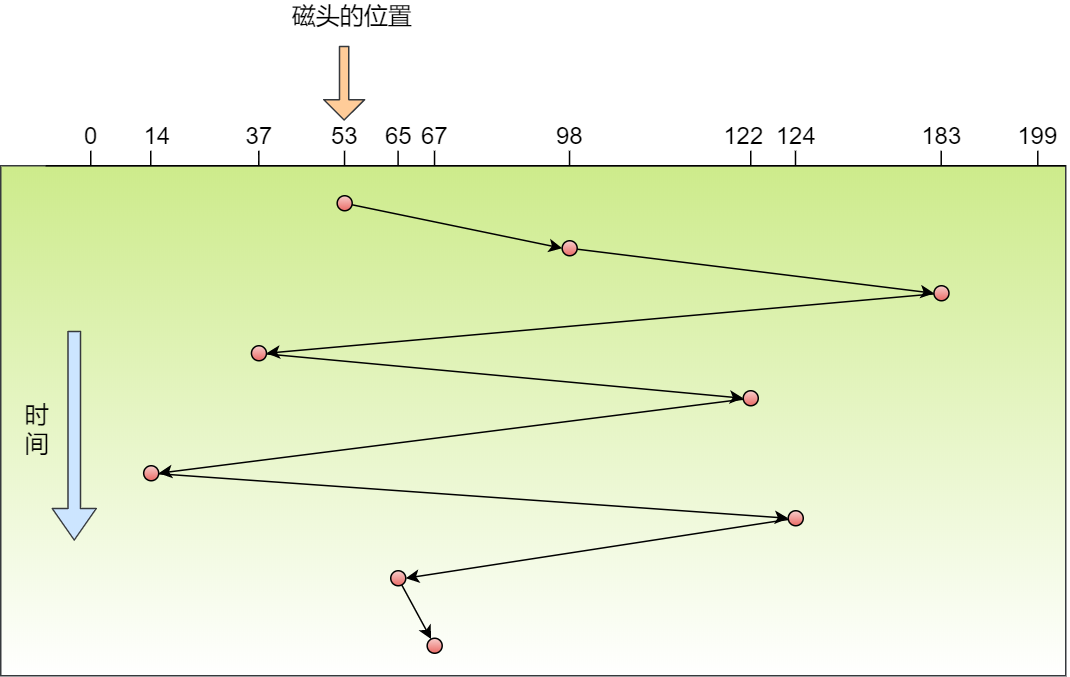
**磁盘调度算法**

在多道程序系统中，各个进程可能会不断提出不同的对磁盘进行读/写操作的请求。由于有时候这些进程的发送请求的速度比磁盘响应的还要快，因此我们有必要为每个磁盘设备建立一个等待队列。**磁盘调度算法的目的就是为了提高磁盘的访问性能，一般是通过优化磁盘的访问请求顺序来做到的。**



假设有下面一个请求序列，每个数字代表磁道的位置：98, 183, 37, 122, 14, 124, 65, 67。初始磁头当前的位置是在第53磁道。

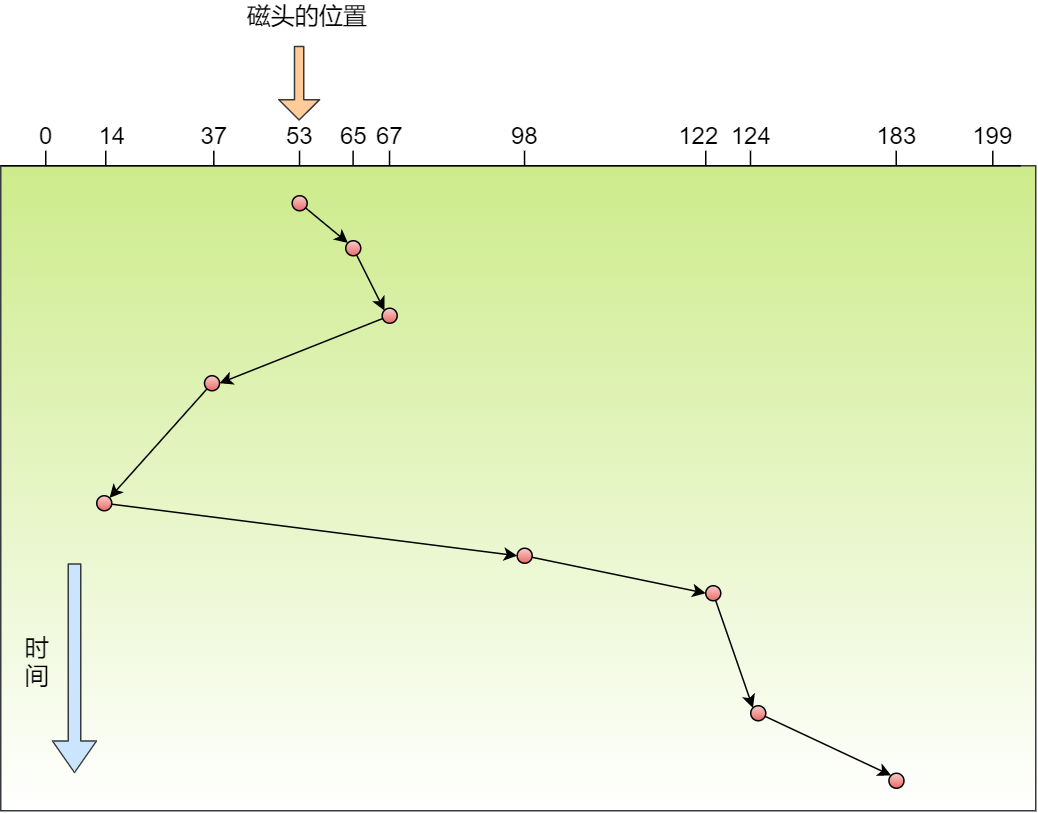
**先来先服务(FCFS, First Come First Served)**



**FCFS算法比较简单粗暴，但是如果大量进程竞争使用磁盘，请求访问的磁道可能会很分散，那FCFS算法因为寻道时间过长，在性能上就会显得很差。**

**最短寻道时间优先(SSTF, Shortest Seek Time First)**

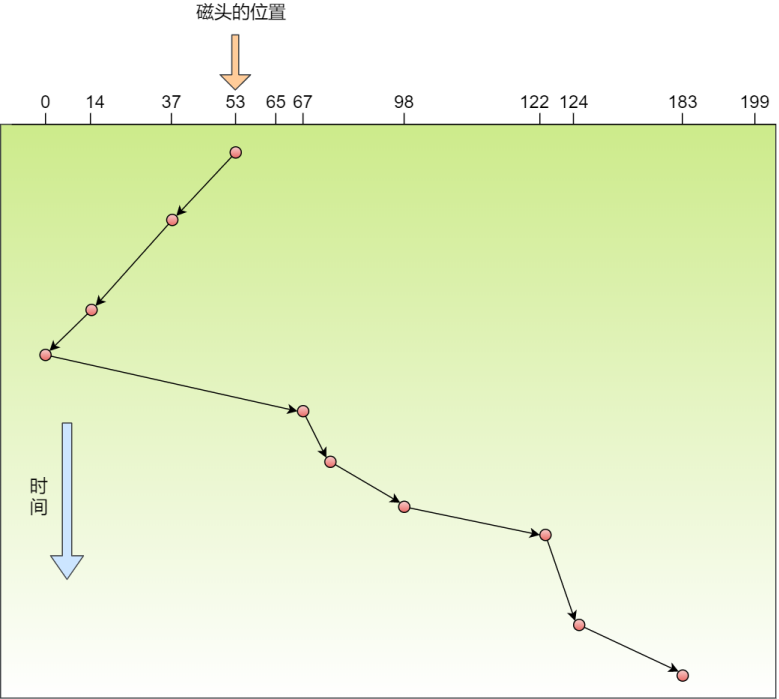
**SSTF算法优先选择从当前磁头位置所需寻道时间最短的请求。**



**但SSTF算法可能存在某些请求的饥饿，对用户的服务请求的响应机会不是均等的。磁头在一小块区域来回移动，因而导致响应时间的变化幅度很大，有些请求的响应时间将不可预期。**

**扫描算法(SCAN)**

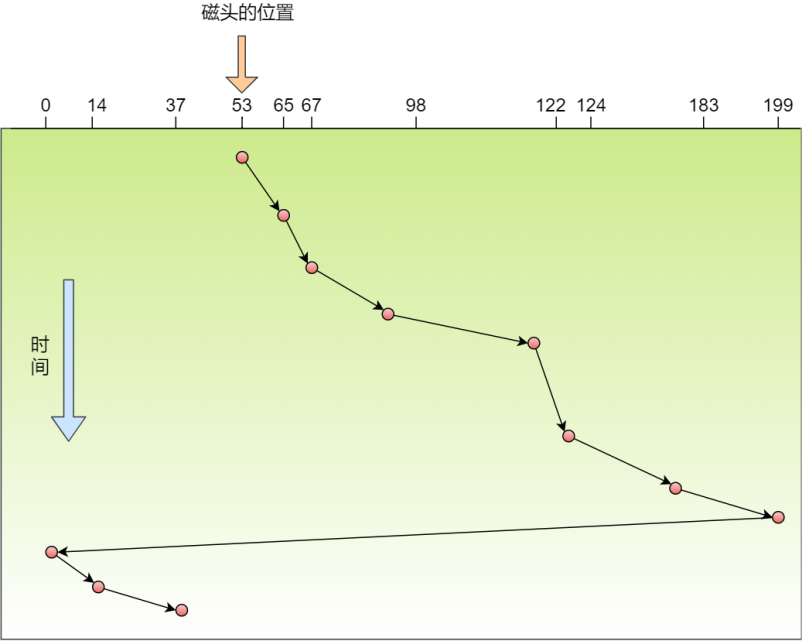
**为了防止SSTF算法产生饥饿的问题，可以规定磁头在一个方向上移动，访问所有未完成的请求，直到磁头到达该方向上的最后的磁道，才调换方向。这种算法也叫做电梯调度算法，比如电梯保持按一个方向移动，直到在那个方向上没有请求为止，然后改变方向。**



**SCAN算法性能较好，不会产生饥饿现象，但是存在这样的问题：中间部分的磁道会比较占便宜，中间部分相比其它部分响应的频率会比较多，也就是说每个磁道的响应频率存在差异。**

**循环扫描算法(CSCAN, Circular Scan)**

**CSCAN算法规定只有磁头朝某个特定方向移动时，才处理磁道访问请求，而返回时直接快速移动至最靠边缘的磁道，也就是复位磁头，这个过程是很快的，并且返回中途不处理任何请求。**



**CSCAN算法的特点就是磁道只响应一个方向上的请求，相比于SCAN算法，对于各个位置磁道响应频率相对比较平均。**

**4.5 RAID**

**RAID (Redundant Array of Independent Disks)**

在单机时代，采用单块磁盘进行数据存储和读写的方式，由于寻址和读写的时间消耗，导致I/O性能非常低，且存储容量还会受到限制。另外，单块磁盘极其容易出现物理故障，经常导致数据的丢失。

1988年美国加州大学伯克利分校的D. A. Patterson教授等首次在论文“A Case of Redundant Array of Inexpensive Disks”中提出了RAID概念，即**廉价冗余磁盘阵列（Redundant Array of Inexpensive Disks）**。**由于当时大容量磁盘比较昂贵，RAID的基本思想是将多个容量较小、相对廉价的磁盘进行有机组合，从而以较低的成本获得与昂贵大容量磁盘相当的容量、性能、可靠性**。随着磁盘成本和价格的不断降低，“廉价”已经毫无意义。因此决定用“Independent”替代“Inexpensive”。于时RAID变成了**独立磁盘冗余阵列（Redundant Array of Independent Disks）**，但这仅仅是名称的变化，实质内容没有改变。

RAID思想从提出后就广泛被业界所接纳，存储工业界投入了大量的时间和财力来研究和开发相关产品。随着处理器、内存、计算机接口等技术的不断发展，RAID不断地发展和革新，在计算机存储领域得到了广泛的应用，从高端系统逐渐延伸到普通的中低端系统。RAID主要优势包括：

1. **大容量**：扩大了磁盘的容量，由多个磁盘组成的RAID系统具有海量的存储空间。现在单个磁盘的容量就可以到1TB以上，这样RAID的存储容量就可以达到PB级，大多数的存储需求都可以满足。

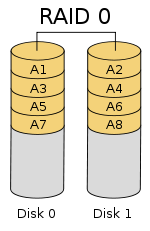
2. **高性能**：单个磁盘的I/O性能受到接口、带宽等计算机技术的限制，性能往往很有限，容易成为系统性能的瓶颈。通过数据条带化，RAID将数据I/O分散到各个成员磁盘上，从而获得比单个磁盘成倍增长的聚合I/O性能。

3. **可靠性**：RAID冗余技术大幅提升数据可用性和可靠性，保证了若干磁盘出错时，不会导致数据的丢失，不影响系统的连续运行。

**RAID0**

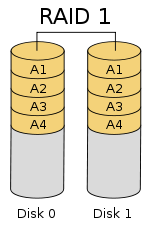
RAID0是一种简单的、无数据校验的数据条带化技术，将所在磁盘条带化后组成大容量的存储空间，将数据分散存储在所有磁盘中，以独立访问方式实现多块磁盘的并读访问。

RAID0具有低成本、高读写性能，但是它不提供数据冗余保护，一旦数据损坏，将无法恢复。因此，RAID0一般适用于对性能要求严格但对数据安全性和可靠性不高的应用，如视频、音频存储、临时数据缓存空间等。



**RAID1**

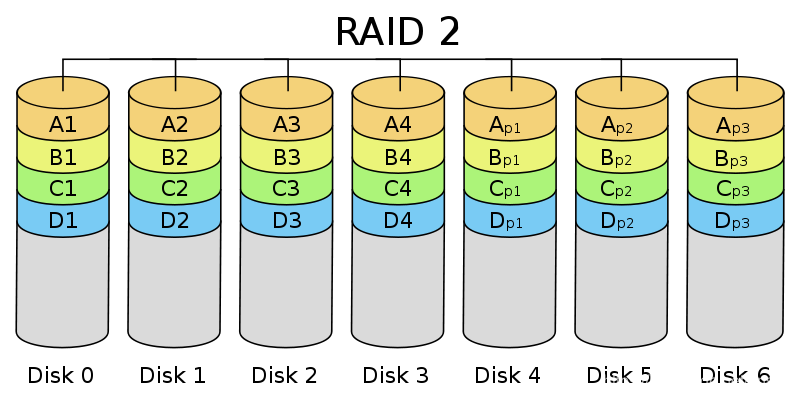
RAID1称为镜像，它将数据完全一致地分别写到工作磁盘和镜像磁盘，它的磁盘空间利用率为50%。RAID1在数据写入时，响应时间会有所影响，但是读数据的时候没有影响。RAID1提供了最佳的数据保护，一旦工作磁盘发生故障，系统自动从镜像磁盘读取数据，不会影响用户工作。RAID1拥有完全容错的能力，但实现成本高。



**RAID2**

RAID2称为纠错海明码磁盘阵列，其设计思想是利用海明码实现数据校验冗余。海明码是一种在原始数据中加入若干校验码来进行错误检测和纠正的编码技术。海明码自身具备纠错能力，因此RAID2可以在数据发生错误的情况下对纠正错误，保证数据的安全性。

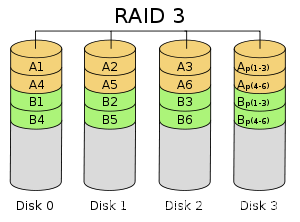
但是海明码的数据冗余开销太大，而且RAID2的数据输出性能受阵列中最慢磁盘驱动器的限制。并且海明码是按位运算，RAID2数据重建非常耗时。因此RAID2在实际中很少应用，没有形成商业产品。



**RAID3**

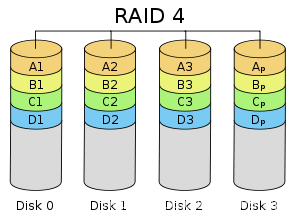
RAID3是使用专用校验盘的并行访问阵列，它采用一个专用的磁盘作为校验盘，其余磁盘作为数据盘。不同磁盘上同一带区的数据作XOR校验，校验值写入校验盘中。RAID3完好时读性能与RAID0完全一致，并行从多个磁盘条带读取数据，性能非常高，同时还提供了数据容错能力。如果RAID3中某一磁盘出现故障，不会影响数据读取，可以借助校验数据和其他完好数据来重建数据。

RAID3只需要一个校验盘，阵列的存储空间利用率高，再加上并行访问的特征，能够为高带宽的大量读写提供高性能，适用大容量数据的顺序访问应用，如影像处理、流媒体服务等。



**RAID4**

RAID4与RAID3的原理大致相同，区别在于条带化的方式不同。RAID4按照块的方式来组织数据，保证单块的完整性，可以避免受到其他磁盘上同条带产生的不利影响。

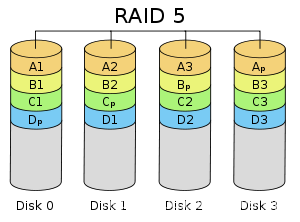


**RAID5**

RAID5应该是目前最常见的RAID等级，它的原理与RAID4相似，区别在于校验数据分布在阵列中的所有磁盘上，而没有采用专门的校验磁盘。因此，RAID5不存在RAID4中的并发写操作时的校验盘性能瓶颈问题。

RAID5的磁盘上同时存储数据和校验数据，数据块和对应的校验信息存保存在不同的磁盘上，当一个数据盘损坏时，系统可以根据同一条带的其他数据块和对应的校验数据来重建损坏的数据。

RAID5 兼顾存储性能、数据安全和存储成本等各方面因素，是目前综合性能最佳的数据保护解决方案。RAID5基本上可以满足大部分的存储应用需求，数据中心大多采用它作为应用数据的保护方案。



**第5章 文件系统**

**5.1 文件操作**

**打开文件**

计算机除了拥有计算的能力之外，一定要有数据的**存储**能力，而对于数据的存储一般就可以通过文件的形式来完成。在Python中直接提供有文件的**IO (Input/Output)处理函数操作**，利用这些函数可以方便地实现**读取**和**写入**。

**open()**函数的功能是进行**文件的打开**，在进行文件打开的时候如果不设置任何的模式类型，则**默认为“r”（只读模式）**。

|  |
| --- |
| *def* open(*file*, *mode*='r', *buffering*=None, *encoding*=None, *errors*=None, *newline*=None, *closefd*=True) |

Python对于文件的模式提供多种类型：

|  |  |
| --- | --- |
| **模式** | **描述** |
| **r** | **使用只读模式打开文件，此为默认模式** |
| **w** | **写模式，如果文件存在则覆盖，文件不存在则创建** |
| **x** | **写模式，新建一个文件，如果该文件已存在则会报错** |
| **a** | **内容追加模式** |
| **b** | **二进制模式** |
| **t** | **文本模式（默认）** |
| **+** | **打开一个文件进行更新（可读可写）** |

如果以**只读**的模式打开文件，并且**文件路径不存在**的话，就会出现**FileNotFoundError**的错误信息。

|  |  |
| --- | --- |
| **范例：打开文件** | |
| *def* main():      file = open(*file*="data.txt", *mode*="w")      print("文件名称：%s" % file.name)      print("访问模式：%s" % file.mode)      print("文件状态：%s" % file.closed)      print("关闭文件...")      file.close()      print("文件状态：%s" % file.closed)  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **运行结果** | 文件名称：data.txt  访问模式：w  文件状态：False  关闭文件...  文件状态：True |

**文件读写**

当使用open()函数打开一个文件之后，接下来可以使用创建的**文件对象**进行读写操作，**文件操作方法**如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| def close(self) | 关闭文件资源 |
| def fileno(self) | 获取文件描述符，返回内容为：0（标准输入stdin）、1（标准输出stdout）、2（标准错误stderr）、其它数字（映射打开文件的地址） |
| def flush(self) | 强制刷新缓冲区 |
| def read(self, n: int = -1) | 数据读取，默认读取全部内容，也可以设置读取个数 |
| def readlines(self, hint: int = -1) | 读取所有数据行，并以列表的形式返回 |
| def readline(self, limit: int = -1) | 读取每行数据（“\n”为结尾），也可以设置读取个数 |
| def truncate(self, size: int = None) | 文件截取 |
| def writable(self) | 判断文件是否可以写入 |
| def write(self, s: AnyStr) | 文件写入 |
| def writelines(self, lines, List[AnyStr]) | 写入一组数据 |

既然所有的文件对象最终都需要被开发者关闭，那么可以结合**with**语句实现**自动的关闭处理**。通过with实现所有**资源对象的连接和释放**是在Python中编写**资源操作**的重要技术手段，通过这样的操作可以极大地**减少和优化代码结构**。

|  |  |
| --- | --- |
| **范例：写入数据** | |
| *def* main():      with open(*file*="data.txt", *mode*="r", *encoding*="utf-8") as file:          for line in file:              print(line, *end*='')  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **文件内容**  **data.txt** | 小灰 16  小白 17  小黄 21 |

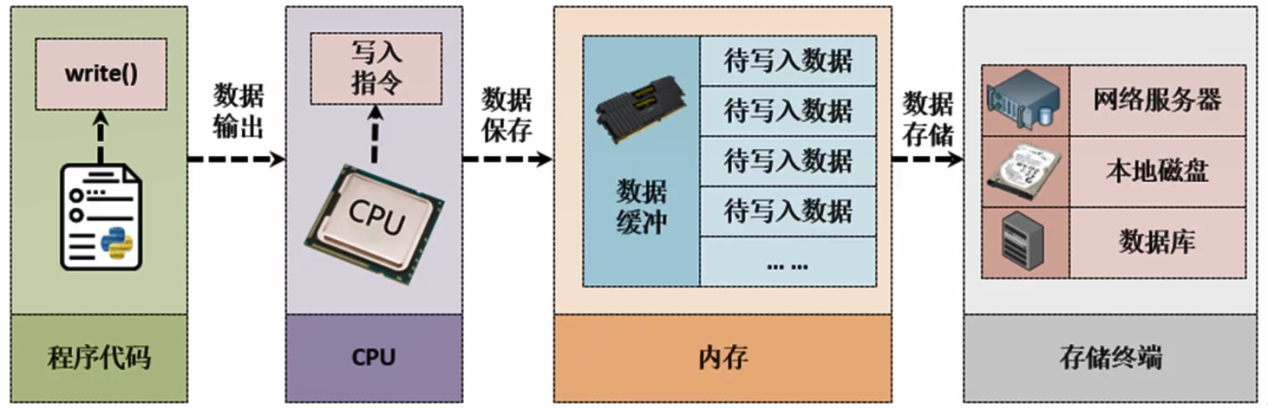
使用**读模式**打开文件后，可以使用**循环读取每一行**的数据内容。Python在进行文件读取操作的时候也可以进一步**简化**操作。**文件对象**本身是可以**迭代**的，在迭代的时候是以换行转义字符“**\n**”为主进行**分割**，每次迭代就读取到一行数据内容。

|  |  |
| --- | --- |
| **范例：读取文件** | |
| *def* main():      with open(*file*="data.txt", *mode*="w", *encoding*="utf-8") as file:          info = {"小灰": 16, "小白": 17, "小黄": 21}          for name, age in info.items():              file.write("%s\t%d\n" % (name, age))  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **运行结果** | 小灰 16  小白 17  小黄 21 |

**5.2 文件缓冲**

**文件缓冲**

在使用open()函数创建一个文件对象的时候，默认情况下是不会启用缓冲的。在**open()**函数中提供的**buffering参数**描述的就是文件处理**缓冲**的定义，在进行文件写入的时候利用缓冲可以**避免频繁的IO资源占用**。



**开启缓冲可以提高写入效率**，buffering参数设置有3种类型：

* **全缓冲（buffering内容大于1）**：当标准IO缓存被填满后才会进行真正IO操作，全缓冲的典型代表就是对磁盘文件的读写操作。
* **行缓冲（buffering设置为1）**：在IO操作中遇见换行符时才执行真正的IO操作，例如在使用网络聊天工具时所编辑的文字在没有发送前是不会进行IO操作的。
* **不缓冲（buffering设置为0）**：直接进行终端设备的IO操作，数据不经过缓冲保存。

如果想要观察到行缓冲的使用特点，就**不能**直接使用**with**语句，因为with最后会执行close()的关闭操作，而一旦**关闭**，则缓冲的内容会**全部**进行输出。

使用**flush()**方法进行缓冲区的**强制清空**，一旦强制清空之后，缓冲区的内容将全部输出。每次使用**close()**方法**关闭文件流**的时候**默认**情况下也会调用flush()方法进行缓冲区的清空处理。

|  |  |
| --- | --- |
| **范例：文件缓冲** | |
| import os  *def* main():      file = open(*file*="data.txt", *mode*="w", *encoding*="utf-8", *buffering*=1)      file.write("This is a test.")      os.system("pause")  # 程序暂停      file.flush()        # 强制清空缓冲区      os.system("pause")  # 程序暂停      file.close()  if \_\_name\_\_ == "\_\_main\_\_":      main() | |
| **文件内容**  **data.txt** | This is a test. |

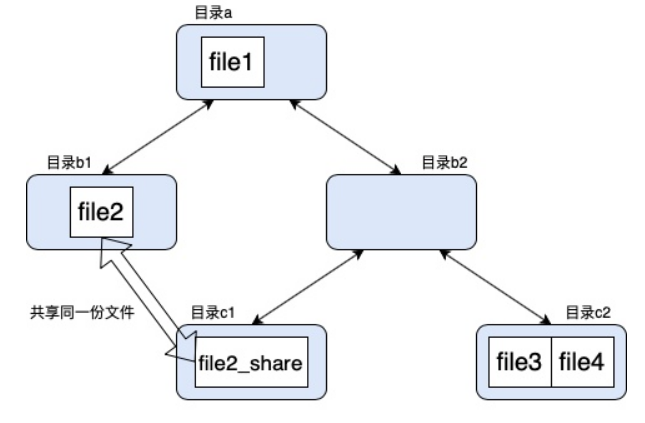
**5.3 文件系统概念**

**文件系统**

**文件系统是OS中负责管理持久数据的子系统，也就是负责把用户的文件存到磁盘硬件中，即使计算机断电了，磁盘里的数据并不会丢失，所以可以持久化地保存文件。**文件系统的基本数据单位是文件，它的目的是对磁盘上的文件进行组织管理。

**Linux最经典的一句话是“一切皆文件”，不仅普通的文件和目录，就连块设备、管道、socket等，也都是统一交给文件系统管理的。**

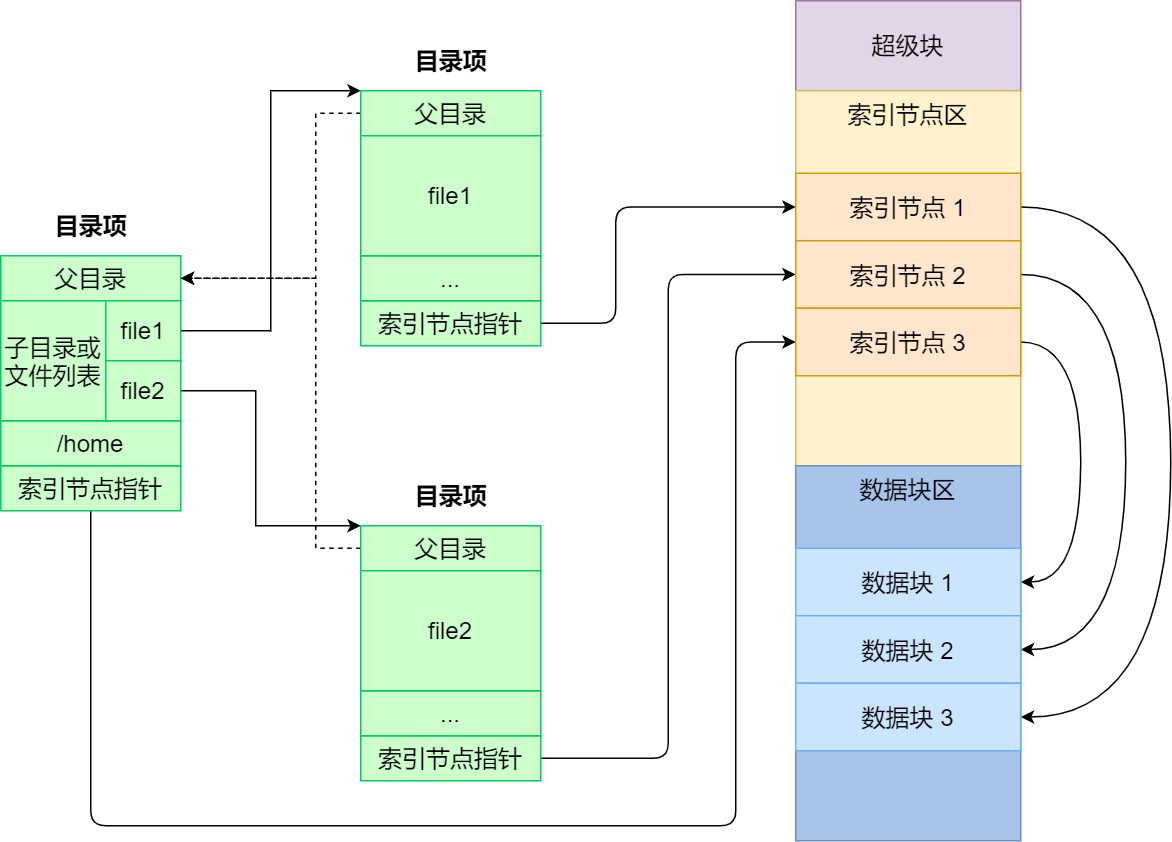
文件和目录是按照层次关系管理的，这种结构可以用树表示。



文件夹中包含了子文件夹或者文件，这样就形成了一个树的结构。不过也有一些特殊情况，例如团队在协同工作时会共享一些信息，两个人有着不同的目录，但是可以共享同一份文件。

**Linux文件系统会为每个文件分配索引结点（inode, index node）和目录项（directory entry）这两个数据结构。**索引结点用来记录文件的元信息，比如inode编号、文件大小、访问权限、创建时间、修改时间、数据在磁盘的位置等。目录项用来记录文件的名字、索引结点指针以及与其它目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，但它与索引结点不同的是，目录项是由内核维护的一个数据结构，不存放于磁盘，而是缓存在内存。

由于索引结点唯一标识一个文件，而目录项记录着文件名，所以目录项和索引结点的关系是多对一。也就是说，一个文件可以有多个别名，例如，**硬链接**的实现就是多个目录项中的索引结点指向同一个文件。

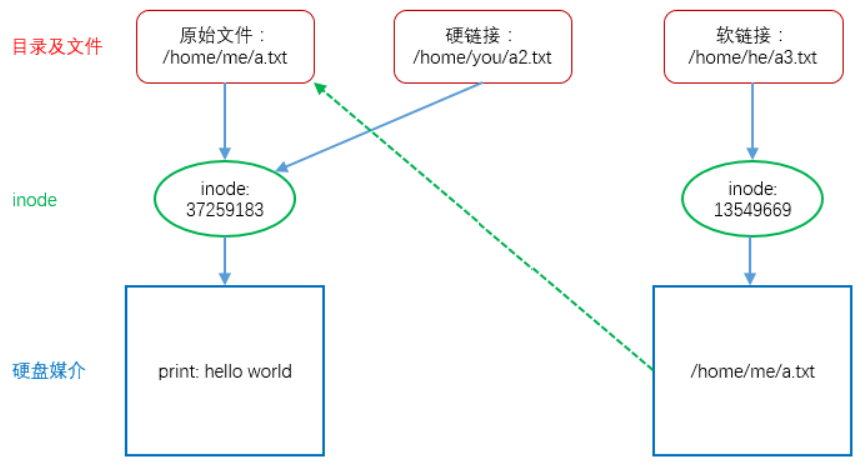


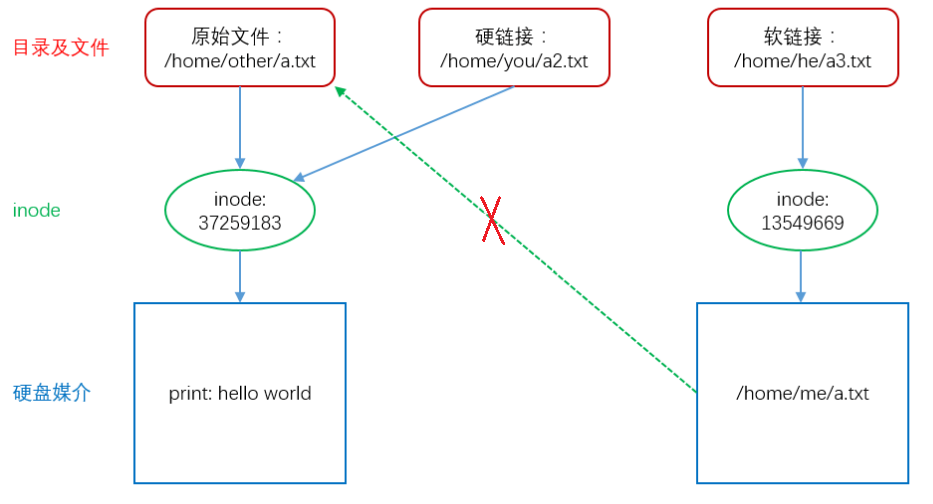
**硬链接与软链接**

链接简单说实际上是一种**文件共享**的方式，主流文件系统都支持链接文件。链接简单地理解为Windows中常见的快捷方式，Linux中常用它来解决一些库版本的问题，通常也会将一些目录层次较深的文件链接到一个更易访问的目录中。

**链接分为硬链接（hard link）和软链接（symbolic link）**。从使用的角度讲，两者没有任何区别，都与正常的文件访问方式一样，支持读写，如果是可执行文件的话也可以直接执行。

硬链接通过同一个inode指向原始文件，软链接通过硬盘媒介中的描述指向原始文件。但是，当原始文件的位置发生改变后，inode不会改变，所以硬链接还是正确的，而软链接就无法访问了。





**5.4 文件存储**

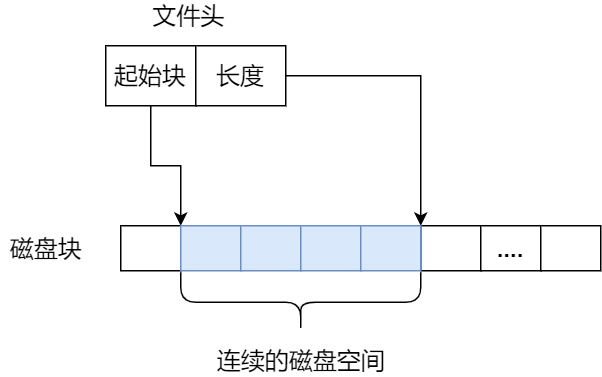
**文件存储**

文件的数据是要存储在硬盘上面的，数据在磁盘上的存放方式，就像程序在内存中存放的方式那样，分为**连续空间存放**和**非连续空间存放**两种方式。不同的存储方式有各自的特点，它们的**存储效率**和**读写性能**各不相同。

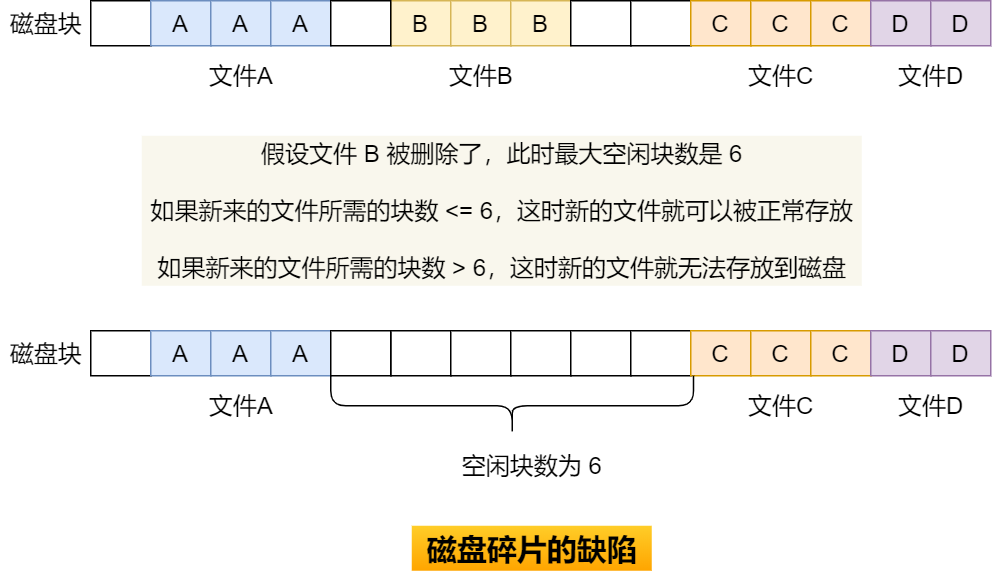
**顺序分配**

**连续空间存放方式顾名思义，文件存放在磁盘连续的物理空间中。这种模式下，文件的数据都是紧密相连，读写效率很高，因为一次磁盘寻道就可以读出整个文件。**

**使用连续存放的方式有一个前提，必须先知道一个文件的大小，这样文件系统才会根据文件的大小在磁盘上找到一块连续的空间分配给文件。所以文件头里需要指定起始块的位置和长度，有了这两个信息就可以很好地表示文件存放方式是一块连续的磁盘空间。**



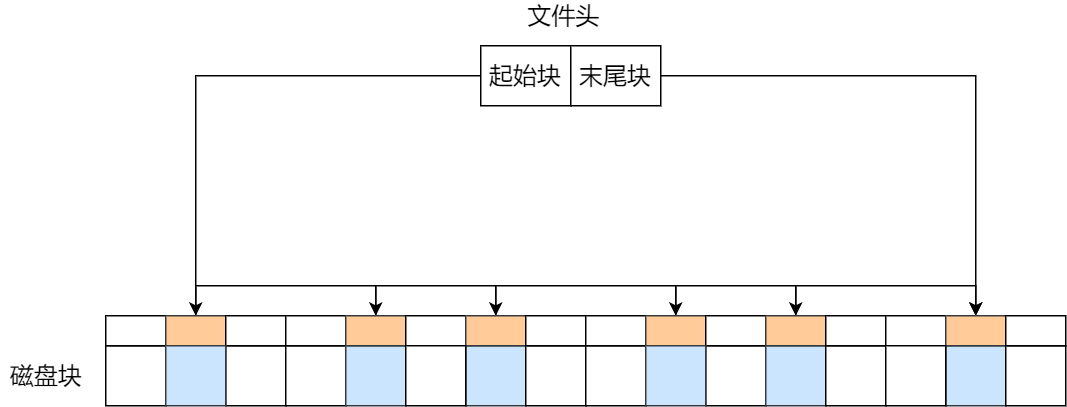
**连续空间存放的方式虽然读写效率高，但是有磁盘空间碎片和文件长度不易扩展的缺陷。**



**链表分配**

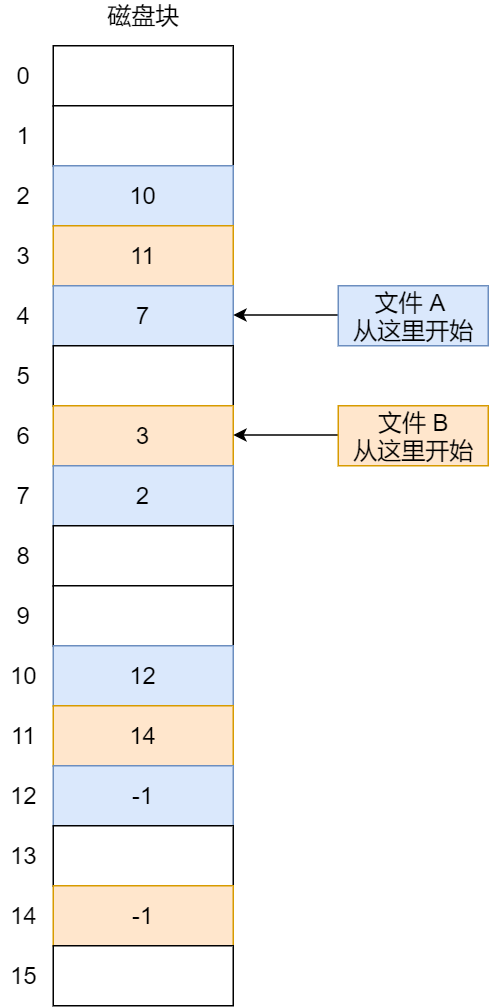
**链表的方式存放是离散的，于是就可以消除磁盘碎片，可大大提高磁盘空间的利用率，同时文件的长度可以动态扩展。根据实现的方式的不同，链表可分为隐式链表和显式链接两种形式。**

**隐式链表实现的方式是文件头要包含第一块和最后一块的位置，并且每个数据块里面留出一个指针空间，用来存放下一个数据块的位置，这样从链头开始就可以顺着指针找到所有的数据块，所以存放是不连续的。**



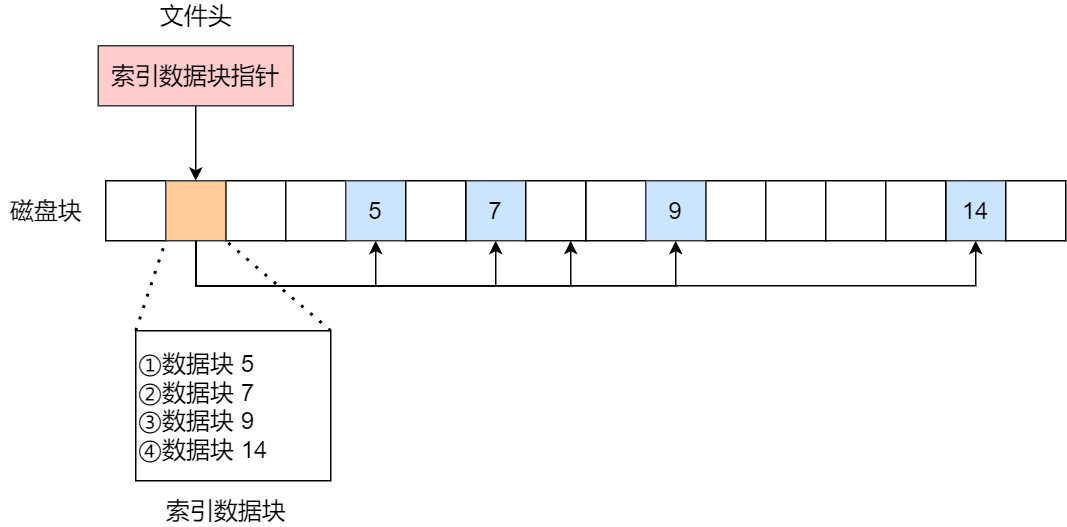
**隐式链表存放方式的缺点在于无法直接访问数据块，只能通过指针顺序访问，以及数据块指针消耗了一定的存储空间。隐式链表稳定性较差，系统在运行过程中由于软件或者硬件错误导致指针丢失或损坏，会导致文件数据的丢失。**

**如果取出每个磁盘块的指针，把它放在内存的一个表中，就可以解决上述隐式链表的两个不足。显式链接指把用于链接文件各数据块的指针，显式地存放在内存的一张链接表中，该表在整个磁盘仅设置一张，每个表项中存放链接指针，指向下一个数据块号。这个表格称为文件分配表（FAT, File Allocation Table）。**



**索引分配**

**链表的方式解决了连续分配的磁盘碎片和文件动态扩展的问题，但是不能有效支持直接访问，索引的方式可以解决这个问题。索引的实现是为每个文件创建一个索引数据块，里面存放的是指向文件数据块的指针列表，类似于书的目录，通过目录就可以找到对应章节。文件头需要包含指向索引数据块的指针，这样就可以通过文件头知道索引数据块的位置，再通过索引数据块里的索引信息找到对应的数据块。**



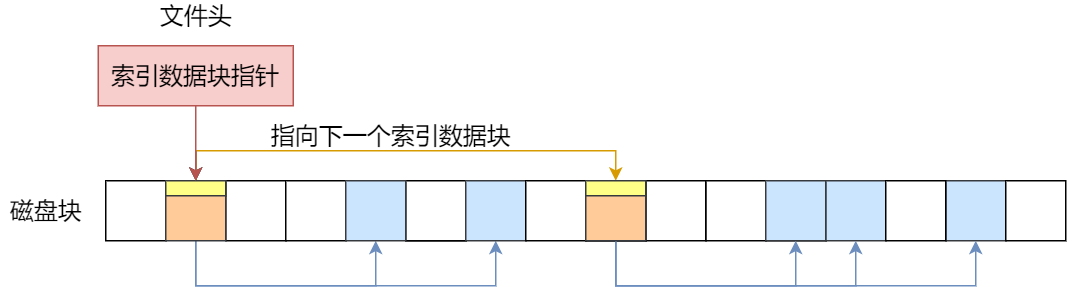
**索引的方式优点在于：**

**1. 文件的创建、增大、缩小很方便。**

**2. 不会有碎片的问题。**

**3. 支持顺序读写和随机读写。**

如果文件很大，大到一个索引数据块放不下索引信息，这时可以通过链表与索引的组合，这种组合称为**链式索引块**，它的实现方式是在索引数据块留出一个存放下一个索引数据块的指针，于是当一个索引数据块的索引信息用完了，就可以通过指针的方式，找到下一个索引数据块的信息。



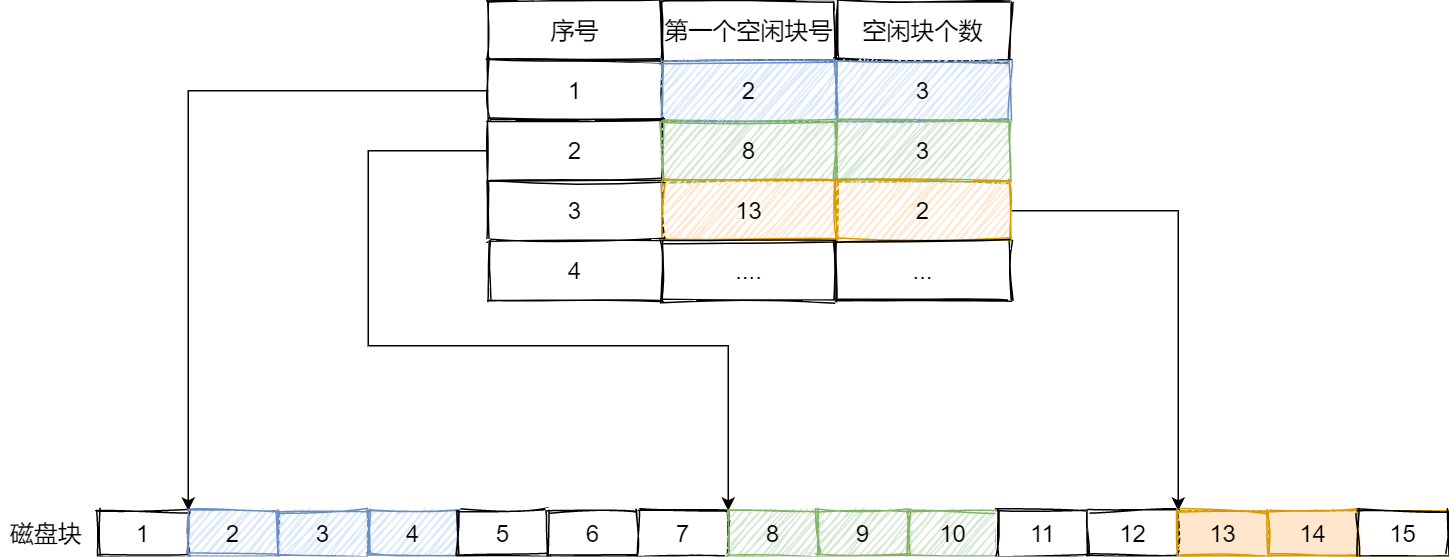
**5.5 空闲空间管理**

**空闲空间管理**

文件的存储是针对已经被占用的数据块的组织和管理，如果要保存一个数据块，应该放在硬盘上的哪个位置呢？如果将所有的块扫描一遍寻找一个空闲空间，这种方式效率太低了，所以针对磁盘的空闲空间也要引入管理的机制。

**空闲表法**

**空闲表法就是为所有空闲空间建立一张表，表内容包括空闲区的第一个块号和该空闲区的块个数。注意，这个方式是连续分配的。**

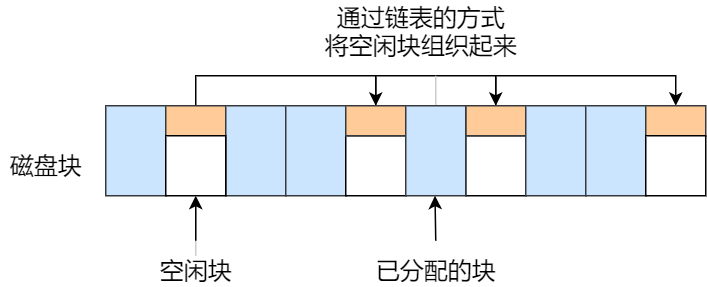


当请求分配磁盘空间时，系统依次扫描空闲表里的内容，直到找到一个合适的空闲区域为止。当用户撤销一个文件时，系统回收文件空间。这时，也需顺序扫描空闲表，寻找一个空闲表条目并将释放空间的第一个物理块号及它占用的块数填到这个条目中。

这种方法仅当有少量的空闲区时才有较好的效果，因为如果存储空间中有着大量的小的空闲区，则空闲表变得很大，这样查询效率会很低。另外，这种分配技术适用于建立连续文件。

**空闲链表法**

**除了空闲表，也可以使用链表的方式来管理空闲空间，每一个空闲块里有一个指针指向下一个空闲块，这样也能很方便的找到空闲块并管理起来。当创建文件需要一块或几块时，就从链头上依次取下一块或几块。反之，当回收空间时，把这些空闲块依次接到链头上。**



这种技术只要在主存中保存一个指针，令它指向第一个空闲块。其特点是简单，但不能随机访问，工作效率低，因为每当在链上增加或移动空闲块时需要做很多I/O操作，同时数据块的指针消耗了一定的存储空间。

空闲表法和空闲链表法都不适合用于大型文件系统，因为这会使空闲表或空闲链表太大。

**位图法(Bit Table)**

**位图是利用二进制的一位来表示磁盘中一个盘块的使用情况，磁盘上所有的盘块都有一个二进制位与之对应。当值为0时，表示对应的盘块空闲，值为1时，表示对应的盘块已分配。**Linux文件系统就采用了位图的方式来管理空闲空间。