# Linux 中进程和线程是如何创建的，区别在哪里

在 Linux 中进程和线程实际上都是用一个结构体 task\_struct来表示一个执行任务的实体。进程创建调用fork 系统调用，而线程创建则是 pthread\_create 方法，但是这两个方法最终都会调用到 do\_fork 来做具体的创建操作 ，区别就在于传入的参数不同。

深究下去，你会发现 Linux 实现线程的方式简直太巧妙了，实际上根本没有**线程**，它创建的就是进程，只不过通过参数指定多个进程之间共享某些资源（如虚拟内存、页表、文件描述符等），函数调用栈、寄存器等线程私有数据则独立。

# 阻塞和挂起的区别和联系

阻塞：正在执行的进程由于发生某时间（如I/O请求、申请缓冲区失败等）暂时无法继续执行。此时引起进程调度，OS把处理机分配给另一个就绪进程，而让受阻进程处于暂停状态，一般将这种状态称为阻塞状态。

挂起：由于系统和用户的需要引入了挂起的操作，进程被挂起意味着该进程处于静止状态。如果进程正在执行，它将暂停执行，若原本处于就绪状态，则该进程此时暂不接受调度。

共同点：

1. 进程都暂停执行。
2. 进程都释放CPU，即两个过程都会涉及上下文切换。

不同点：

1. 对系统资源占用不同：虽然都释放了CPU，但阻塞的进程仍处于内存中，而挂起的进程通过“对换”技术被换出到外存（磁盘）中。
2. 发生时机不同：阻塞一般在进程等待资源（IO资源、信号量等）时发生；而挂起是由于用户和系统的需要，例如，终端用户需要暂停程序研究其执行情况或对其进行修改、OS为了提高内存利用率需要将暂时不能运行的进程（处于就绪或阻塞队列的进程）调出到磁盘
3. 恢复时机不同：阻塞要在等待的资源得到满足（例如获得了锁）后，才会进入就绪状态，等待被调度而执行；被挂起的进程由将其挂起的对象（如用户、系统）在时机符合时（调试结束、被调度进程选中需要重新执行）将其主动激活。

# 进程的上下文切换

进程是由内核管理和调度的，所以进程的切换只能发生在内核态。所以，**进程的上下文切换不仅包含了虚拟内存、栈、全局变量等用户空间的资源，还包括了内核堆栈、寄存器等内核空间的资源。**通常会把CPU的寄存器和程序计数器的交换的信息保存在进程的 PCB，当要运行另外一个进程的时候，我们需要从这个进程的 PCB 取出上下文，然后恢复到 CPU 中，这使得这个进程可以继续执行。所以进程的上下文开销是很关键的，我们希望它的开销越小越好，这样可以使得进程可以把更多时间花费在执行程序上，而不是耗费在上下文切换。

发生进程上下文切换有哪些场景？

* 为了保证所有进程可以得到公平调度，CPU 时间被划分为一段段的时间片，这些时间片再被轮流分配给各个进程。这样，当某个进程的时间片耗尽了，就会被系统挂起，切换到其它正在等待 CPU 的进程运行；
* 进程在系统资源不足（比如内存不足）时，要等到资源满足后才可以运行，这个时候进程也会被挂起，并由系统调度其他进程运行；
* 当进程通过睡眠函数 sleep 这样的方法将自己主动挂起时，自然也会重新调度；
* 当有优先级更高的进程运行时，为了保证高优先级进程的运行，当前进程会被挂起，由高优先级进程来运行；
* 发生硬件中断时，CPU 上的进程会被中断挂起，转而执行内核中的中断服务程序；

# 进程的控制

### 创建进程

操作系统允许一个进程创建另一个进程，而且允许子进程继承父进程所拥有的资源，当子进程被终止时，其在父进程处继承的资源应当还给父进程。同时，终止父进程时同时也会终止其所有的子进程。

创建进程的过程如下：

* 为新进程分配一个唯一的进程标识号，并申请一个空白的 PCB，PCB 是有限的，若申请失败则创建失败；
* 为进程分配资源，此处如果资源不足，进程就会进入等待状态，以等待资源；
* 初始化 PCB；
* 如果进程的调度队列能够接纳新进程，那就将进程插入到就绪队列，等待被调度运行；

### 终止进程

进程可以有 3 种终止方式：正常结束、异常结束以及外界干预（信号 kill 掉）。

终止进程的过程如下：

* 查找需要终止的进程的 PCB；
* 如果处于执行状态，则立即终止该进程的执行，然后将 CPU 资源分配给其他进程；
* 如果其还有子进程，则应将其所有子进程终止；
* 将该进程所拥有的全部资源都归还给父进程或操作系统；
* 将其从 PCB 所在队列中删除；

### 阻塞进程

当进程需要等待某一事件完成时，它可以调用阻塞语句把自己阻塞等待。而一旦被阻塞等待，它只能由另一个进程唤醒。

阻塞进程的过程如下：

* 找到将要被阻塞进程标识号对应的 PCB；
* 如果该进程为运行状态，则保护其现场，将其状态转为阻塞状态，停止运行；
* 将该 PCB 插入的阻塞队列中去；

### 唤醒进程

进程由「运行」转变为「阻塞」状态是由于进程必须等待某一事件的完成，所以处于阻塞状态的进程是绝对不可能叫醒自己的。

如果某进程正在等待 I/O 事件，需由别的进程发消息给它，则只有当该进程所期待的事件出现时，才由发现者进程用唤醒语句叫醒它。

唤醒进程的过程如下：

* 在该事件的阻塞队列中找到相应进程的 PCB；
* 将其从阻塞队列中移出，并置其状态为就绪状态；
* 把该 PCB 插入到就绪队列中，等待调度程序调度；

进程的阻塞和唤醒是一对功能相反的语句，如果某个进程调用了阻塞语句，则必有一个与之对应的唤醒语句。

# 进程的组织

有三部分组成：PCB，程序段和数据段

**PCB 是进程存在的唯一标识**，这意味着一个进程的存在，必然会有一个 PCB，如果进程消失了，那么 PCB 也会随之消失。

PCB 具体包含什么信息呢？

**进程描述信息：**

* 进程标识符：标识各个进程，每个进程都有一个并且唯一的标识符；
* 用户标识符：进程归属的用户，用户标识符主要为共享和保护服务；

**进程控制和管理信息：**

* 进程当前状态，如 new、ready、running、waiting 或 blocked 等；
* 进程优先级：进程抢占 CPU 时的优先级；

**资源分配清单：**

* 有关内存地址空间或虚拟地址空间的信息，所打开文件的列表和所使用的 I/O 设备信息。

**CPU 相关信息：**

* CPU 中各个寄存器的值，当进程被切换时，CPU 的状态信息都会被保存在相应的 PCB 中，以便进程重新执行时，能从断点处继续执行。

通常是通过**链表**的方式进行组织，把具有**相同状态的进程链在一起，组成各种队列**。比如：

* 将所有处于就绪状态的进程链在一起，称为**就绪队列**；
* 把所有因等待某事件而处于等待状态的进程链在一起就组成各种**阻塞队列**；

# 什么是线程

**线程是进程当中的一条执行流程。**同一个进程内多个线程之间可以共享代码段、数据段、打开的文件等资源，但每个线程都有独立一套的寄存器和栈，这样可以确保线程的控制流是相对独立的。

# 线程的优缺点

线程的优点：

* 一个进程中可以同时存在多个线程；
* 各个线程之间可以并发执行；
* 各个线程之间可以共享地址空间和文件等资源；

线程的缺点：

* 当进程中的一个线程奔溃时，会导致其所属进程的所有线程奔溃。

# 进程和线程的区别

* 调度 ：进程是操作系统分配资源的一个基本单位。线程是 CPU调度的基本单位。**（最大的区别）**
* 并发性：引入线程之后，不仅进程之间是可以并发执行的，而且在一个进程之中的多个线程也是可以并发执行的。同样，不同的进程中的线程也是可以并发执行的。使得OS有更好的并发性，提高了资源的利用率和系统吞吐量。
* 拥有资源：进程拥有一个完整的资源平台，并且是系统拥有资源的基本单位 。线程本身并不拥有系统资源，仅有一些能保证独立运行的资源，如寄存器和栈；
* 系统开销：线程切换的开销低于进程切换的开销，
* 支持多处理机系统：对于传统的进程，也就是单线程进程 ，不管有多少个处理机，进程只能运行在同一个处理机上面，但对于多线程进程，就可以将一个进程中的多个线程分配到多个处理机上面，使其并发执行，加速了进程的完成。

# 为什么线程能减少开销

* 线程的创建时间比进程快，因为进程在创建的过程中，还需要资源管理信息，比如内存管理信息、文件管理信息，而线程在创建的过程中，不会涉及这些资源管理信息，而是共享它们；
* 线程的终止时间比进程快，因为线程释放的资源相比进程少很多；
* 同一个进程内的线程切换比进程切换快，因为线程具有相同的地址空间（虚拟内存共享），这意味着同一个进程的线程都具有同一个页表，那么在切换的时候不需要切换页表。而对于进程之间的切换，切换的时候要把页表给切换掉，而页表的切换过程开销是比较大的；
* 由于同一进程的各线程间共享内存和文件资源，那么在线程之间数据传递的时候，就不需要经过内核了，这就使得线程之间的数据交互效率更高了；

所以，线程比进程不管是时间效率，还是空间效率都要高。

# 线程的上下文切换

这还得看线程是不是属于同一个进程：

* 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；
* **当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据**；

所以，线程的上下文切换相比进程，开销要小很多。

# 进程调度原则

* **CPU 利用率**：调度程序应确保 CPU 是始终匆忙的状态，这可提高 CPU 的利用率；
* **系统吞吐量**：吞吐量表示的是单位时间内 CPU 完成进程的数量，长作业的进程会占用较长的 CPU 资源，因此会降低吞吐量，相反，短作业的进程会提升系统吞吐量；
* **周转时间**：周转时间是进程运行和阻塞时间总和，一个进程的周转时间越小越好；
* **等待时间**：这个等待时间不是阻塞状态的时间，而是进程处于就绪队列的时间，等待的时间越长，用户越不满意；
* **响应时间**：用户提交请求到系统第一次产生响应所花费的时间，在交互式系统中，响应时间是衡量调度算法好坏的主要标准。

说白了，这么多调度原则，目的就是要使得进程要「快」。

# 进程调度算法

接下来，说说在**单核 CPU 系统**中常见的调度算法。

### 先来先服务调度算法

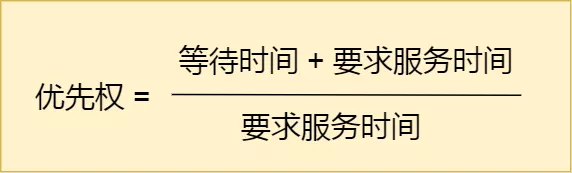
这似乎很公平，但是当一个长作业先运行了，那么后面的短作业等待的时间就会很长，不利于短作业。FCFS 对长作业有利，适用于 CPU 繁忙型作业的系统，而不适用于 I/O 繁忙型作业的系统。

### 最短作业优先调度算法

这显然对长作业不利，很容易造成一种极端现象。比如，一个长作业在就绪队列等待运行，而这个就绪队列有非常多的短作业，那么就会使得长作业不断的往后推，周转时间变长，致使长作业长期不会被运行。

### 高响应比优先调度算法

高响应比优先调度算法主要是权衡了短作业和长作业。「响应比优先级」的计算公式：



如果两个进程的「等待时间」相同时，「要求的服务时间」越短，「响应比」就越高，这样短作业的进程容易被选中运行；

如果两个进程「要求的服务时间」相同时，「等待时间」越长，「响应比」就越高，这就兼顾到了长作业进程，因为进程的响应比可以随时间等待的增加而提高，当其等待时间足够长时，其响应比便可以升到很高，从而获得运行的机会；

### 时间片轮转调度算法

每个进程被分配一个时间段，称为时间片，即允许该进程在该时间段中运行。如果时间片用完，进程还在运行，那么将会把此进程从 CPU 释放出来，并把 CPU 分配另外一个进程；如果该进程在时间片结束前阻塞或结束，则 CPU 立即进行切换；

另外，时间片的长度就是一个很关键的点：如果时间片设得太短会导致过多的进程上下文切换，降低了 CPU 效率；如果设得太长又可能引起对短作业进程的响应时间变长。通常时间片设为 20ms~50ms 通常是一个比较合理的折中值。

### 最高优先级调度算法

调度程序能从就绪队列中选择最高优先级的进程进行运行，这称为最高优先级调度算法。

进程的优先级可以分为，**静态优先级或动态优先级**

静态优先级：创建进程时候，就已经确定了优先级了，然后整个运行时间优先级都不会变化；

动态优先级：根据进程的动态变化调整优先级，比如如果进程运行时间增加，则降低其优先级，如果进程等待时间（就绪队列的等待时间）增加，则升高其优先级，也就是**随着时间的推移增加等待进程的优先级**。

该算法也有两种处理优先级高的方法，**非抢占式和抢占式**：

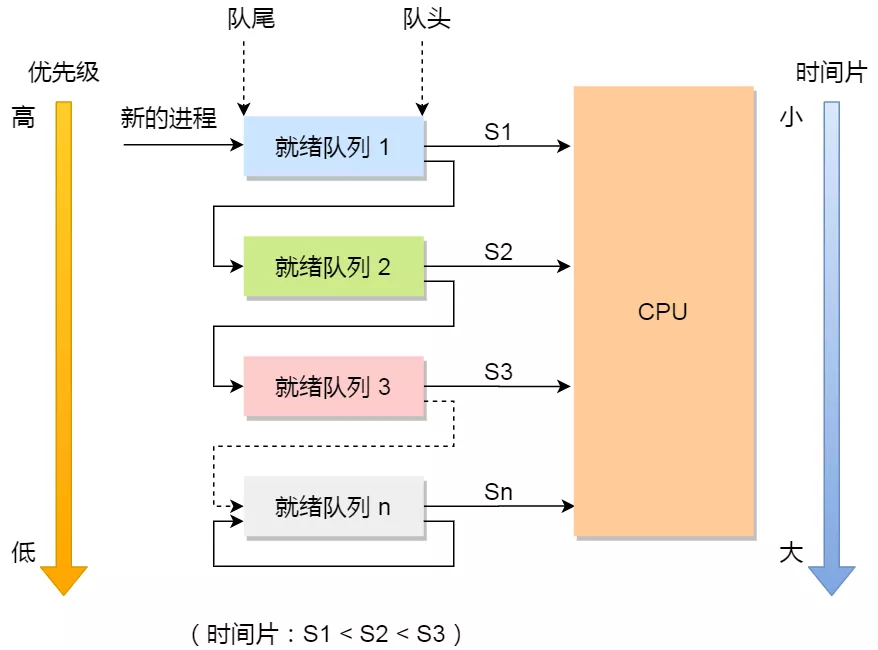
非抢占式：当就绪队列中出现优先级高的进程，运行完当前进程，再选择优先级高的进程。

抢占式：当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。

### 多级反馈队列调度算法

算法是「时间片轮转算法」和「最高优先级算法」的综合和发展。顾名思义：

* 「多级」表示有多个队列，每个队列优先级从高到低，同时优先级越高时间片越短。
* 「反馈」表示如果有新的进程加入优先级高的队列时，立刻停止当前正在运行的进程，转而去运行优先级高的队列；



来看看，它是如何工作的：

* 设置了多个队列，赋予每个队列不同的优先级，每个**队列优先级从高到低**，同时**优先级越高时间片越短**；
* 新的进程会被放入到第一级队列的末尾，按先来先服务的原则排队等待被调度，如果在第一级队列规定的时间片没运行完成，则将其转入到第二级队列的末尾，以此类推，直至完成；
* 当较高优先级的队列为空，才调度较低优先级的队列中的进程运行。如果进程运行时，有新进程进入较高优先级的队列，则停止当前运行的进程并将其移入到原队列末尾，接着让较高优先级的进程运行；

可以发现，对于短作业可能可以在第一级队列很快被处理完。对于长作业，如果在第一级队列处理不完，可以移入下次队列等待被执行，虽然等待的时间变长了，但是运行时间也会更长了，所以该算法很好的**兼顾了长短作业，同时有较好的响应时间。**

# 互斥和同步是什么，作用

在多线程的环境中为了避免发生竞态条件的发生，通常使用互斥和同步两种手段减少程序的在并发环境下资源竞争造成的资源破坏。也就是说使用互斥和同步可以以线程安全的方式去访问某种资源，使其资源的能够正常使用，不被破坏。

互斥是当一个线程在访问一个临界区时，其他的线程不能够访问，被阻塞在临界区外，只有当前线程执行出临界区时，另一个线程才能访问临界区资源。

同步是多个线程之间相互协调互斥访问的一直方式，当一个线程被阻塞的时候，另一个线程可以通知这个被阻塞的线程，从而达到多个线程相互协调这样一个作用。

# 虚拟内存

虚拟内存是操作系统提供的一种内存管理技术。它为每个进程分配独立的一套「虚拟地址」。程序在运行期间可以相互隔离，各自拥有自己的数据。

# 指令的执行过程

指令的执行需要完成取指令、分析指令和执行指令的操作，过程分为：取指令、指令译码、按指令操作码执行、形成下一条指令地址等。

（1）在程序执行之前，会将程序的起始地址送入程序计数器中，该地址在程序加载到内存时确定，所以程序计数器中首先存的是程序第一条指令的地址。将该地址送往地址总线，完成取指操作。

（2）取来的指令暂存到指令寄存器中。

（3）指令译码器从指令寄存器中得到指令，分析指令的操作码和地址码。然后 CPU 根据分析的操作码知道该条指令要进行的操作，根据地址码找到需要的数据，完成指令的执行。

（4）程序计数器加1或根据转移指令得到下一条指令的地址，接下来再进行下一条指令的执行，直到整个程序执行完成。

# 内存分段

## 什么是分段

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。不同的段是有不同的属性的，所以就用分段（Segmentation）的形式把这些段分离出来。

## 虚拟地址和物理地址的映射

分段机制下的虚拟地址由两部分组成，段选择子和段内偏移量。

* 段选择子就保存在段寄存器里面。段选择子里面最重要的是段号，用作段表的索引。段表里面保存的是这个段的基地址、段的界限和特权等级等。
* 虚拟地址中的段内偏移量应该位于 0 和段界限之间，如果段内偏移量是合法的，就将段基地址加上段内偏移量得到物理内存地址。

## 缺点

### 内存碎片

### 内存交换的效率低

* 内存碎片是很容易产生的，产生了内存碎片，那不得不重新 Swap 内存区域，这个过程会产生性能瓶颈。
* 因为硬盘的访问速度要比内存慢太多了，每一次内存交换，我们都需要把一大段连续的内存数据写到硬盘上。
* 所以，如果内存交换的时候，交换的是一个占内存空间很大的程序，这样整个机器都会显得卡顿。

# 内存分页

## 什么是分页

分页是把整个虚拟和物理内存空间切成一段段固定尺寸的大小。这样一个连续并且尺寸固定的内存空间，我们叫页（Page）。在 Linux 下，每一页的大小为 4KB。

页表实际上存储在 内存中， CPU通过 MMU，找出要实际要访问的物理内存地址。

而当进程访问的虚拟地址在页表中查不到时，系统会产生一个缺页异常，进入系统内核空间分配物理内存、更新进程页表，最后再返回用户空间，恢复进程的运行。

## 虚拟地址和物理地址的映射

在分页机制下，虚拟地址分为两部分，页号和页内偏移。页号作为页表的索引，页表包含物理页每页所在物理内存的基地址，这个基地址与页内偏移的组合就形成了物理内存地址，见下图。

## 简单的分页有什么缺陷？

有空间上的缺陷，页表会非常的庞大

解决方法：多级页表

## 多级页表

每个进程都有 4GB 的虚拟地址空间，而显然对于大多数程序来说，其使用到的空间远未达到 4GB，因为会存在部分对应的页表项都是空的，根本没有分配，对于已分配的页表项，如果存在最近一定时间未访问的页表，在物理内存紧张的情况下，操作系统会将页面换出到硬盘，也就是说不会占用物理内存。

如果使用了二级分页，一级页表就可以覆盖整个 4GB 虚拟地址空间，但如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。

我们把二级分页再推广到多级页表，就会发现页表占用的内存空间更少了，这一切都要归功于对局部性原理的充分应用。

对于 64 位的系统，两级分页肯定不够了，就变成了四级目录，分别是：

* 全局页目录项 PGD（Page Global Directory）；
* 上层页目录项 PUD（Page Upper Directory）；
* 中间页目录项 PMD（Page Middle Directory）；
* 页表项 PTE（Page Table Entry）；

## 为什么不分级的页表就做不到这样节约内存呢？

页表一定要覆盖全部虚拟地址空间，不分级的页表就需要有 100 多万个页表项来映射，而二级分页则只需要 1024 个页表项（此时一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建）。

## 快表TLB

在CPU芯片中加入了快表

在 CPU 芯片里面，封装了内存管理单元（Memory Management Unit）芯片，它用来完成地址转换和 TLB 的访问与交互。

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。

TLB 的命中率其实是很高的，因为程序最常访问的页就那么几个。

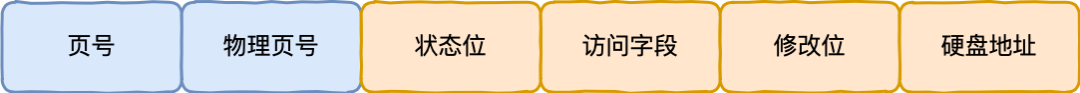
# 页面置换算法

## 缺页异常

当 CPU 访问的页面不在物理内存时，便会产生一个缺页中断，请求操作系统将所缺页调入到物理内存。那它与一般中断的主要区别在于：

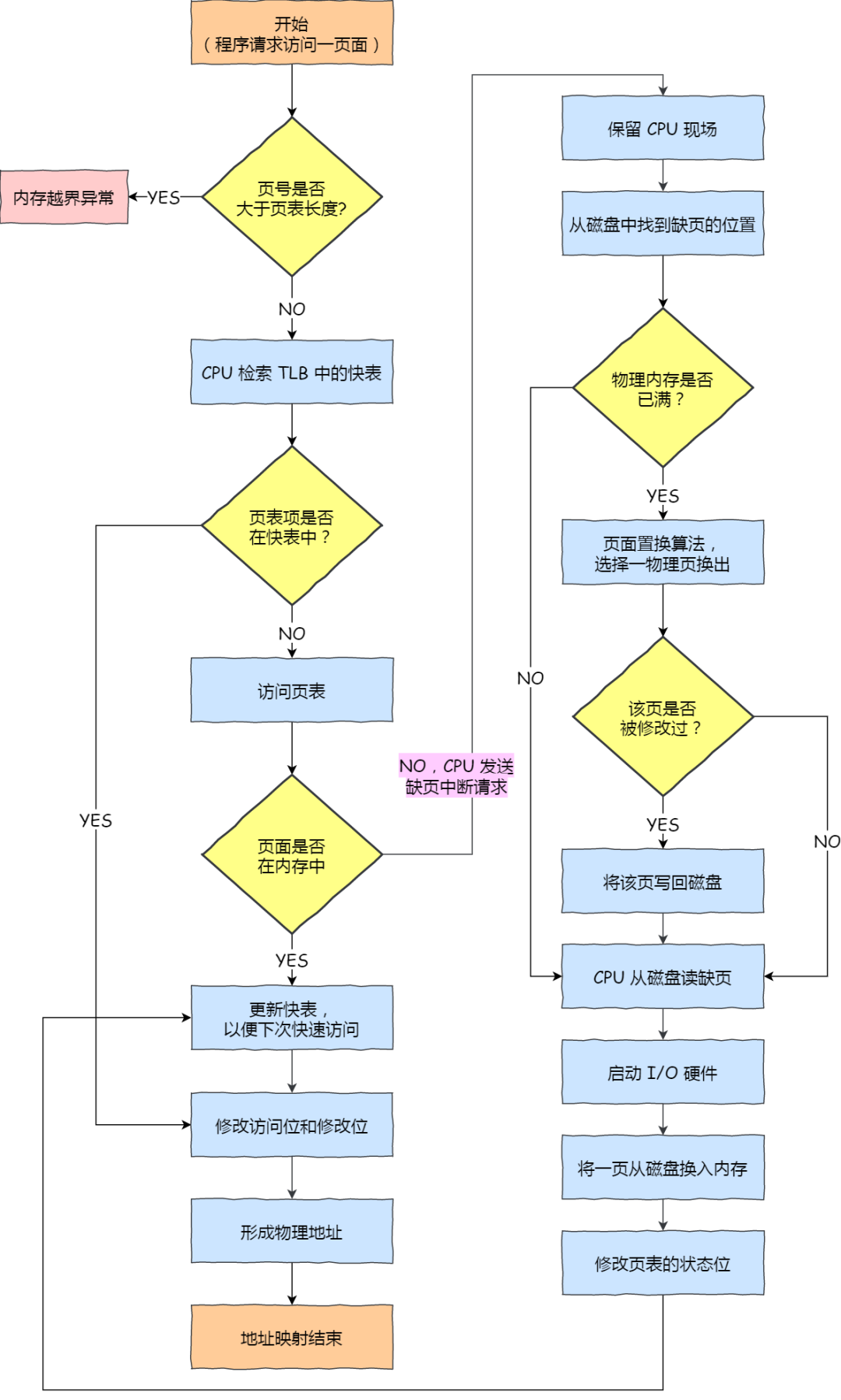
* 缺页中断在指令执行「期间」产生和处理中断信号，而一般中断在一条指令执行「完成」后检查和处理中断信号。
* 缺页中断返回到该指令的开始重新执行「该指令」，而一般中断返回回到该指令的「下一个指令」执行。

页表项结构



* 状态位：用于表示该页是否有效，也就是说是否在物理内存中，供程序访问时参考。
* 访问字段：用于记录该页在一段时间被访问的次数，供页面置换算法选择出页面时参考。
* 修改位：表示该页在调入内存后是否有被修改过，由于内存中的每一页都在磁盘上保留一份副本，因此，如果没有修改，在置换该页时就不需要将该页写回到磁盘上，以减少系统的开销；如果已经被修改，则将该页重写到磁盘上，以保证磁盘中所保留的始终是最新的副本。
* 硬盘地址：用于指出该页在硬盘上的地址，通常是物理块号，供调入该页时使用。

处理流程



## 作用

页面置换算法的功能是，当出现缺页异常，需调入新页面而内存已满时，选择被置换的物理页面，也就是说选择一个物理页面换出到磁盘，然后把需要访问的页面换入到物理页。

## 分类

### 最佳置换算法

* 置换在「未来」最长时间不访问的页面。

### FIFO算法

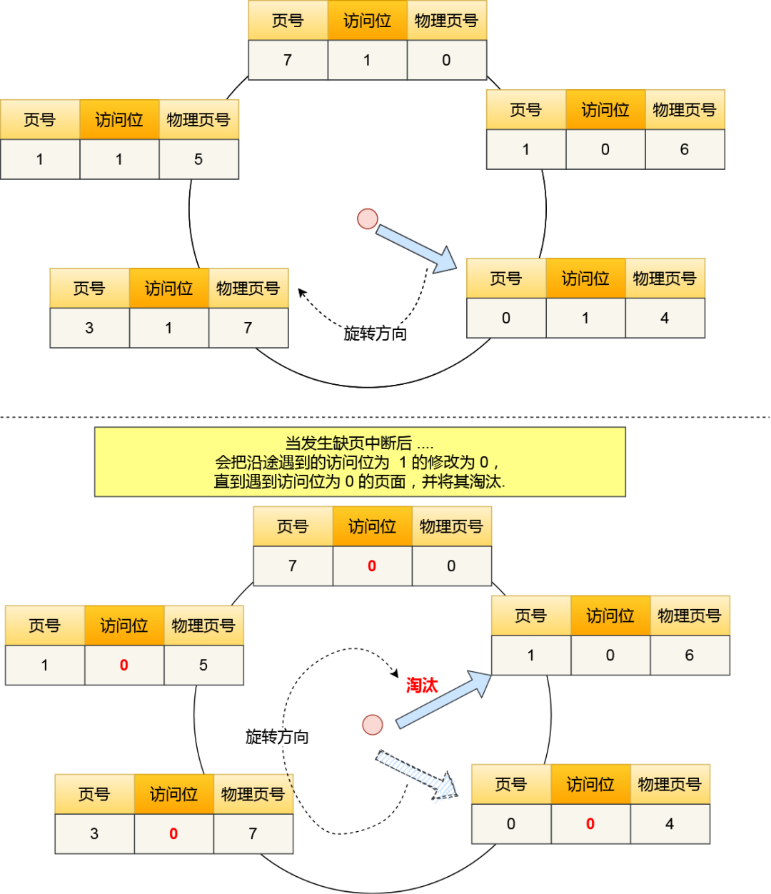
* 选择在内存驻留时间很长的页面进行中置换

### LRU算法

* 选择最长时间没有被访问的页面进行置换
* 虽然 LRU 在理论上是可以实现的，但代价很高。为了完全实现 LRU，需要在内存中维护一个所有页面的链表，最近最多使用的页面在表头，最近最少使用的页面在表尾。
* 困难的是，在每次访问内存时都必须要更新「整个链表」。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作。

### Clock算法

* 把所有的页面都保存在一个类似钟面的「环形链表」中，一个表针指向最老的页面。
* 发生缺页中断时，算法首先检查表针指向的页面：
  + 如果它的访问位位是 0 就淘汰该页面，并把新的页面插入这个位置，然后把表针前移一个位置；
  + 如果访问位是 1 就清除访问位，并把表针前移一个位置，重复这个过程直到找到了一个访问位为 0 的页面为止；



### LFU算法

* 当发生缺页中断时，选择「访问次数」最少的那个页面，并将其淘汰。
* 对每个页面设置一个「访问计数器」，每当一个页面被访问时，该页面的访问计数器就累加 1。在发生缺页中断时，淘汰计数器值最小的那个页面。

# 段页式内存管理

## 实现方式

先将程序划分为多个有逻辑意义的段，也就是前面提到的分段机制；

接着再把每个段划分为多个页，也就是对分段划分出来的连续空间，再划分固定大小的页；

## 虚拟地址和物理地址的映射

地址结构就由段号、段内页号和页内位移三部分组成。

用于段页式地址变换的数据结构是每一个程序一张段表，每个段又建立一张页表，段表中的地址是页表的起始地址，而页表中的地址则为某页的物理页号

# Linux内存管理

## Intel X86 逻辑地址解析

Intel 80386 的页式内存管理设计时，没有绕开段式内存管理，而是建立在段式内存管理的基础上，这就意味着，页式内存管理的作用是在由段式内存管理所映射而成的的地址上再加上一层地址映射。

* 程序所使用的地址，通常是没被段式内存管理映射的地址，称为逻辑地址；
* 通过段式内存管理映射的地址，称为线性地址，也叫虚拟地址；

## Linux 采用了什么方式管理内存？

主要采用的是页式内存管理，但同时也不可避免地涉及了段机制。

这主要是上面 Intel 处理器发展历史导致的，因为 Intel X86 CPU 一律对程序中使用的地址先进行段式映射，然后才能进行页式映射。既然 CPU 的硬件结构是这样，Linux 内核也只好服从 Intel 的选择。

Linux 系统中的每个段都是从 0 地址开始的整个 4GB 虚拟空间（32 位环境下），也就是所有的段的起始地址都是一样的。这意味着，Linux 系统中的代码，包括操作系统本身的代码和应用程序代码，所面对的地址空间都是线性地址空间（虚拟地址），这种做法相当于屏蔽了处理器中的逻辑地址概念，段只被用于访问控制和内存保护。

## Linux 的虚拟地址空间是如何分布的？

虚拟地址空间的内部又被分为内核空间和用户空间两部分，不同位数的系统，地址空间的范围也不同。比如最常见的 32 位和 64 位系统

虽然每个进程都各自有独立的虚拟内存，但是每个虚拟内存中的内核地址，其实关联的都是相同的物理内存。这样，进程切换到内核态后，就可以很方便地访问内核空间内存。

## Linux用户空间的划分情况？

* 程序文件段，包括二进制可执行代码；
* 已初始化数据段，包括静态常量；
* 未初始化数据段，包括未初始化的静态变量；
* 堆段，包括动态分配的内存，从低地址开始向上增长；
* 文件映射段，包括动态库、共享内存等，从低地址开始向上增长（跟硬件和内核版本有关）
* 栈段，包括局部变量和函数调用的上下文等。栈的大小是固定的，一般是 8 MB。当然系统也提供了参数，以便我们自定义大小；

# 文件系统的基本组成

文件系统是操作系统中负责管理持久数据的子系统，说简单点，就是负责把用户的文件存到磁盘硬件中。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，那组织的方式不同，就会形成不同的文件系统。

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。

Linux 文件系统会为每个文件分配两个数据结构：索引节点（index node）和目录项（directory entry）

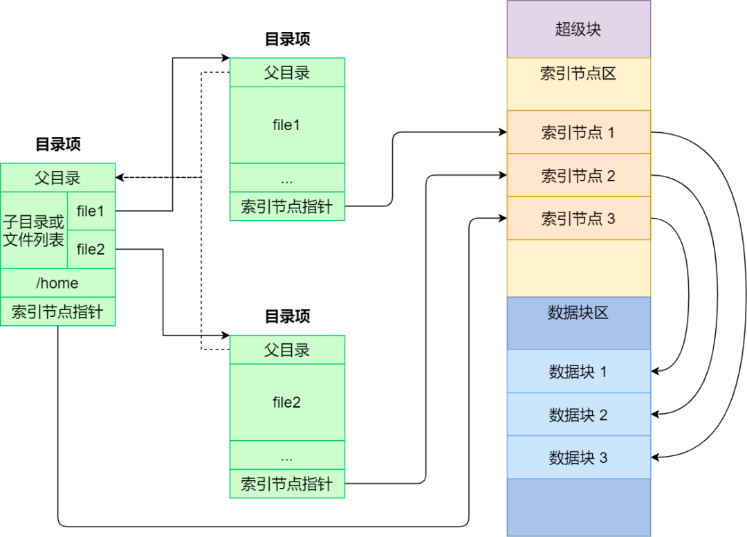
目录项，也就是 dentry，用来记录文件的名字、索引节点指针以及与其他目录项的层级关联关系。多个目录项关联起来，就会形成目录结构，但它与索引节点不同的是，目录项是由内核维护的一个数据结构，不存放于磁盘，而是缓存在内存。

索引节点，也就是 inode，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等。索引节点是文件的唯一标识，它们之间一一对应，也同样都会被存储在硬盘中，所以索引节点同样占用磁盘空间。

# 文件是如何存储的

磁盘读写的最小单位是扇区，扇区的大小只有 512B 大小。文件系统把多个扇区组成了一个逻辑块，每次读写按照块来读，Linux 中的逻辑块大小为 4KB

索引节点、目录项以及文件数据之间的关系



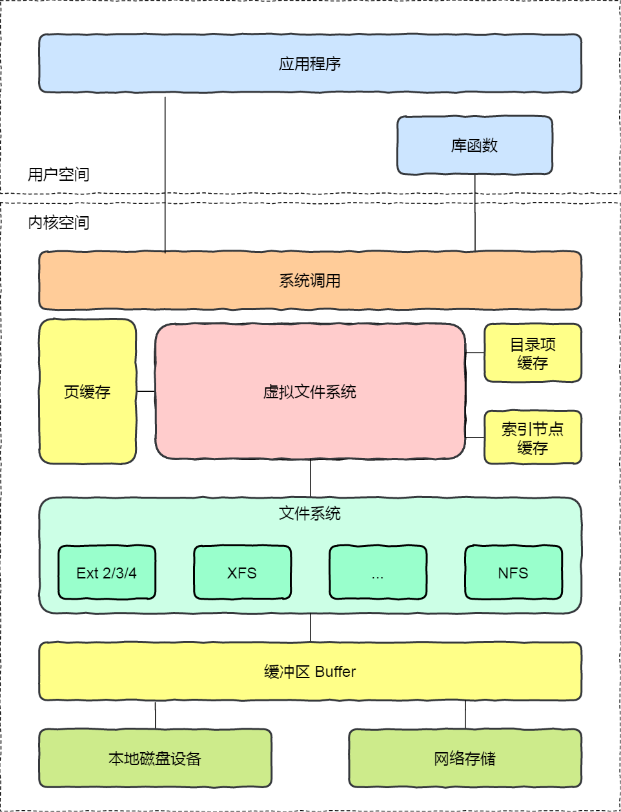
磁盘进行格式化的时候，会被分成三个存储区域

* 超级块，用来存储文件系统的详细信息，比如块个数、块大小、空闲块等等。当文件系统挂载时进入内存；
* 索引节点区，用来存储索引节点；当文件被访问时进入内存；
* 数据块区，用来存储文件或目录数据；

# 虚拟文件系统

文件系统的种类众多，而操作系统希望对用户提供一个统一的接口，于是在用户层与文件系统层引入了中间层，这个中间层就称为虚拟文件系统（Virtual File System，VFS）。

在 Linux 文件系统中，用户空间、系统调用、虚拟机文件系统、缓存、文件系统以及存储之间的关系如上图

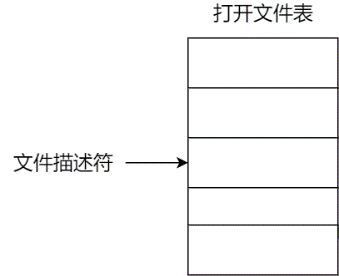


文件系统首先要先挂载到某个目录才可以正常使用，比如 Linux 系统在启动时，会把文件系统挂载到根目录。

# 文件的使用

1. 首先用 open 系统调用打开文件，open 的参数中包含文件的路径名和文件名。fd = open(name, flag); # 打开文件
2. 使用 write 写数据，其中 write 使用 open 所返回的文件描述符，并不使用文件名作为参数。write(fd,...); # 写数据
3. 使用完文件后，要用 close 系统调用关闭文件，避免资源的泄露。

打开了一个文件后，操作系统会记录进程打开的所有文件，会为每个进程维护一个打开文件表，文件表里的每一项代表「文件描述符」，所以说文件描述符是打开文件的标识。在打开文件表中维护着打开文件的状态和信息如下：



文件指针：系统跟踪上次读写位置作为当前文件位置指针，这种指针对打开文件的某个进程来说是唯一的；

文件打开计数器：文件关闭时，操作系统必须重用其打开文件表条目，否则表内空间不够用。因为多个进程可能打开同一个文件，所以系统在删除打开文件条目之前，必须等待最后一个进程关闭文件，该计数器跟踪打开和关闭的数量，当该计数为 0 时，系统关闭文件，删除该条目；

文件磁盘位置：绝大多数文件操作都要求系统修改文件数据，该信息保存在内存中，以免每个操作都从磁盘中读取；

访问权限：每个进程打开文件都需要有一个访问模式（创建、只读、读写、添加等），该信息保存在进程的打开文件表中，以便操作系统能允许或拒绝之后的 I/O 请求；

# 文件系统的基本操作单位是数据块

当用户进程从文件读取 1 个字节大小的数据时，文件系统则需要获取字节所在的数据块，再返回数据块对应的用户进程所需的数据部分。

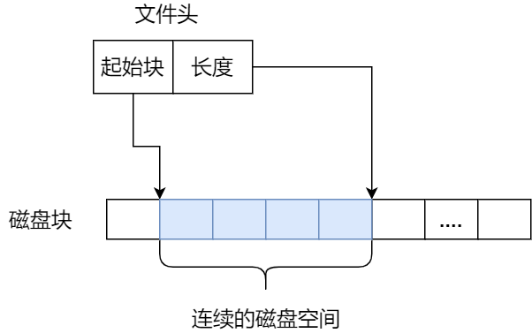
当用户进程把 1 个字节大小的数据写进文件时，文件系统则找到需要写入数据的数据块的位置，然后修改数据块中对应的部分，最后再把数据块写回磁盘。

# 文件的存储

## 连续空间存存储

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。

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」



缺点

* 磁盘空间碎片
* 文件长度不易扩展
* 挪位置解决

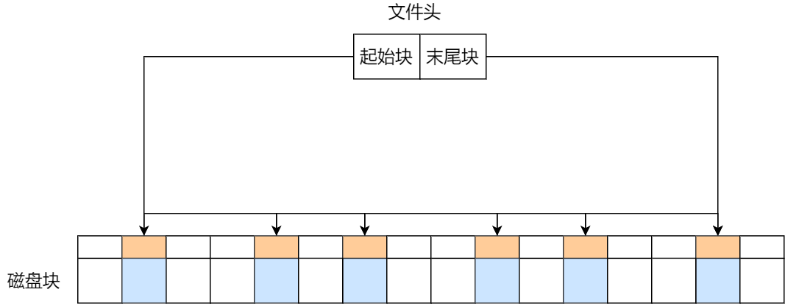
## 非连续空间存储

### 链表方式

链表的方式存放是离散的，解决了磁盘碎片，同时文件的长度可以动态扩展。

#### 隐式链接

文件要以「隐式链表」的方式存放的话，它的文件头要包含「第一块」和「最后一块」的位置，并且每个数据块都有一个指针空间存放下一个数据块的位置，这样从链头开是就可以顺着指针找到所有的数据块。

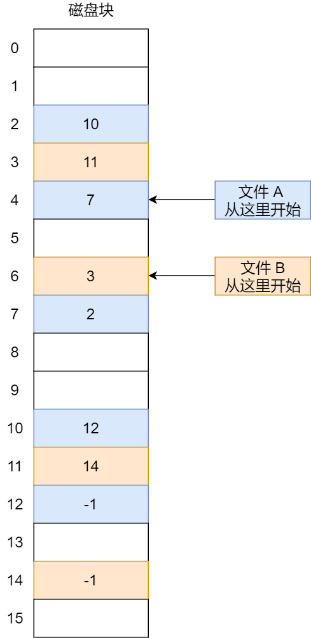


缺点

* 无法直接访问数据块
* 数据块指针消耗了一定的存储空间
* 隐式链接分配的稳定性较差，系统在运行过程中由于软件或者硬件错误导致链表中的指针丢失或损坏，会导致文件数据的丢失。

#### 显式链接

它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。



我们举个例子，文件 A 依次使用了磁盘块 4、7、2、10 和 12 ，文件 B 依次使用了磁盘块 6、3、11 和 14 。利用下图中的表，可以从第 4 块开始，顺着链走到最后，找到文件 A 的全部磁盘块。同样，从第 6 块开始，顺着链走到最后，也能够找出文件 B 的全部磁盘块。最后，这两个链都以一个不属于有效磁盘编号的特殊标记（如 -1 ）结束。

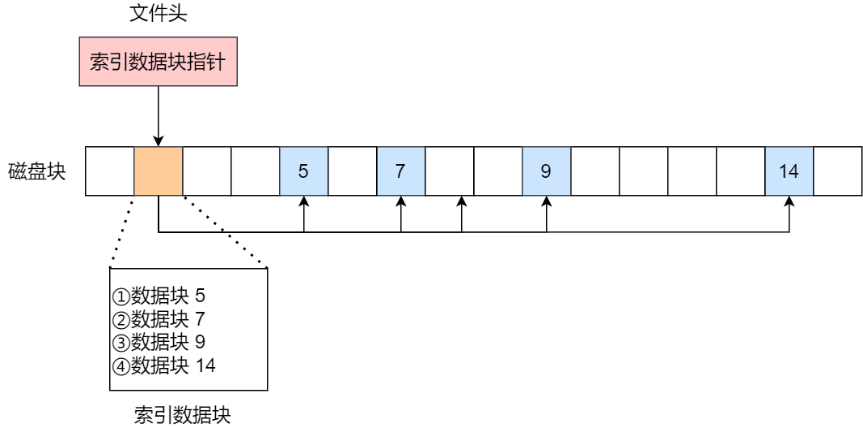
由于查找记录的过程是在内存中进行的，因而不仅显著地提高了检索速度，而且大大减少了访问磁盘的次数。

缺点

* 不适用于大磁盘

### 索引方式

* 解决了直接访问的问题（FAT除外）
* 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，说白了就像书的目录一样，要找哪个章节的内容，看目录查就可以。
* 另外，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。
* 创建文件时，索引块的所有指针都设为空。当首次写入第 i 块时，先从空闲空间中取得一个块，再将其地址写到索引块的第 i 个条目。



优点

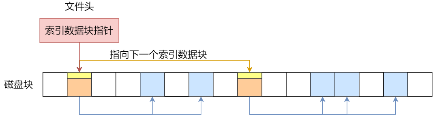
* 文件的创建、增大、缩小很方便；
* 不会有碎片的问题；
* 支持顺序读写和随机读写；

缺点

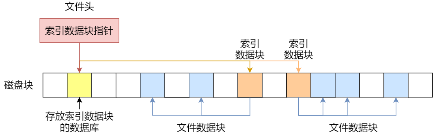
* 需要额外分配一块来存放索引数据块

文件很大，大到一个索引数据块放不下索引信息，如何存储？

「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。那这种方式也会出现前面提到的链表方式的问题，万一某个指针损坏了，后面的数据也就会无法读取了。



「多级索引块」，实现方式是通过一个索引块来存放多个索引数据块，一层套一层索引，像极了俄罗斯套娃是吧。

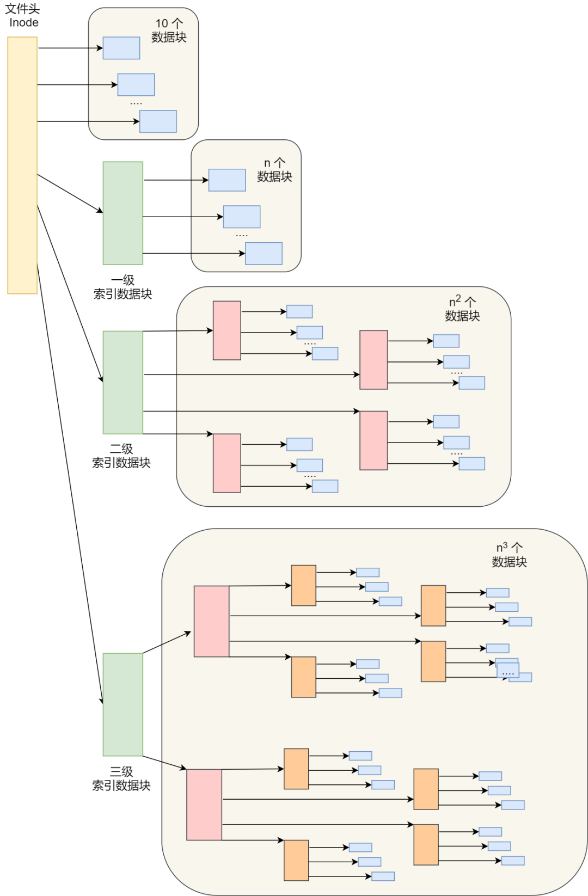


## 对比



# Unix 文件的实现方式

早期 Unix 文件系统是组合了前面的文件存放方式的优点



它是根据文件的大小，存放的方式会有所变化

* 如果存放文件所需的数据块小于 10 块，则采用直接查找的方式；
* 如果存放文件所需的数据块超过 10 块，则采用一级间接索引方式；
* 如果前面两种方式都不够存放大文件，则采用二级间接索引方式；
* 如果二级间接索引也不够存放大文件，这采用三级间接索引方式；

文件头（inode）就需要包含 13 个指针

* 10 个指向数据块的指针；
* 第 11 个指向索引块的指针；
* 第 12 个指向二级索引块的指针；
* 第 13 个指向三级索引块的指针；

这种方式能很灵活地支持小文件和大文件的存放

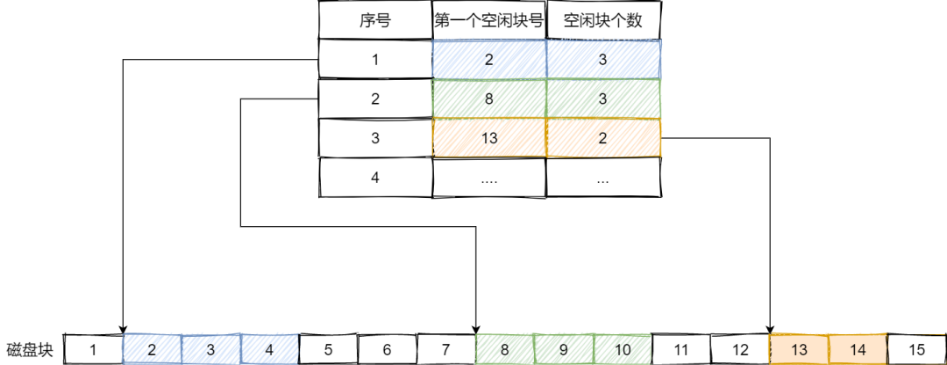
对于小文件使用直接查找的方式可减少索引数据块的开销；

对于大文件则以多级索引的方式来支持，所以大文件在访问数据块时需要大量查询；

# 空闲空间管理

## 空闲表法

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

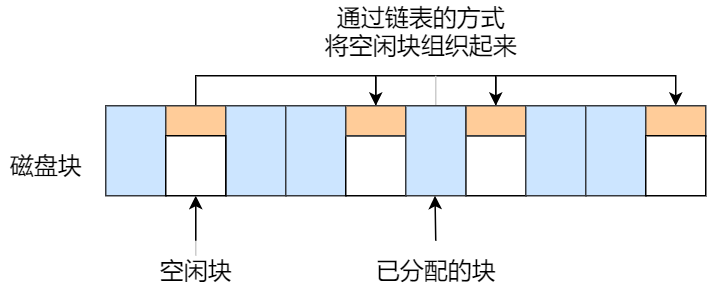


当请求分配磁盘空间时，系统依次扫描空闲表里的内容，直到找到一个合适的空闲区域为止。

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。

## 空闲链表法

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。



只要在主存中保存一个指针，令它指向第一个空闲块。

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。

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。

## 位图法

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。

1111110011111110001110110111111100111 ...

不仅用于数据空闲块的管理，还用于 inode 空闲块的管理，因为 inode 也是存储在磁盘的，自然也要有对其管理。

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

# 文件系统的结构

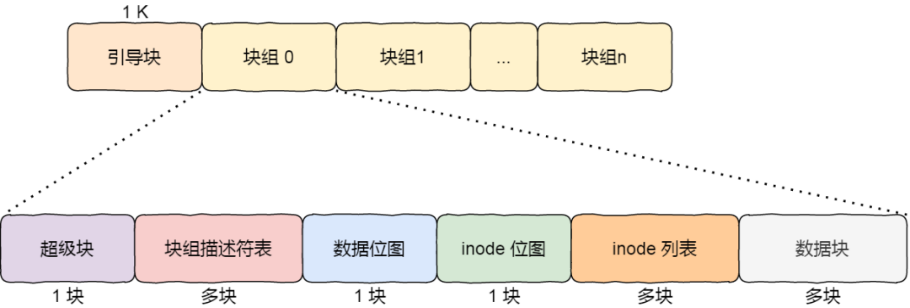
Linux 是用位图的方式管理空闲空间，用户在创建一个新文件时，Linux 内核会通过 inode 的位图找到空闲可用的 inode，并进行分配。要存储数据时，会通过块的位图找到空闲的块，并分配，但仔细计算一下还是有问题的。

数据块的位图是放在磁盘块里的，假设是放在一个块里，一个块 4K，每位表示一个数据块，共可以表示 4 \* 1024 \* 8 = 2^15 个空闲块，由于 1 个数据块是 4K 大小，那么最大可以表示的空间为 2^15 \* 4 \* 1024 = 2^27 个 byte，也就是 128M。

也就是说按照上面的结构，如果采用「一个块的位图 + 一系列的块」，外加「一个块的 inode 的位图 + 一系列的 inode 的结构」能表示的最大空间也就 128M，这太少了，现在很多文件都比这个大。

在 Linux 文件系统，把这个结构称为一个块组，那么有 N 多的块组，就能够表示 N 大的文件。

下图给出了 Linux Ext2 整个文件系统的结构和块组的内容，文件系统都由大量块组组成，在硬盘上相继排布



最前面的第一个块是引导块，在系统启动时用于启用引导，接着后面就是一个一个连续的块组了，块组的内容如下：

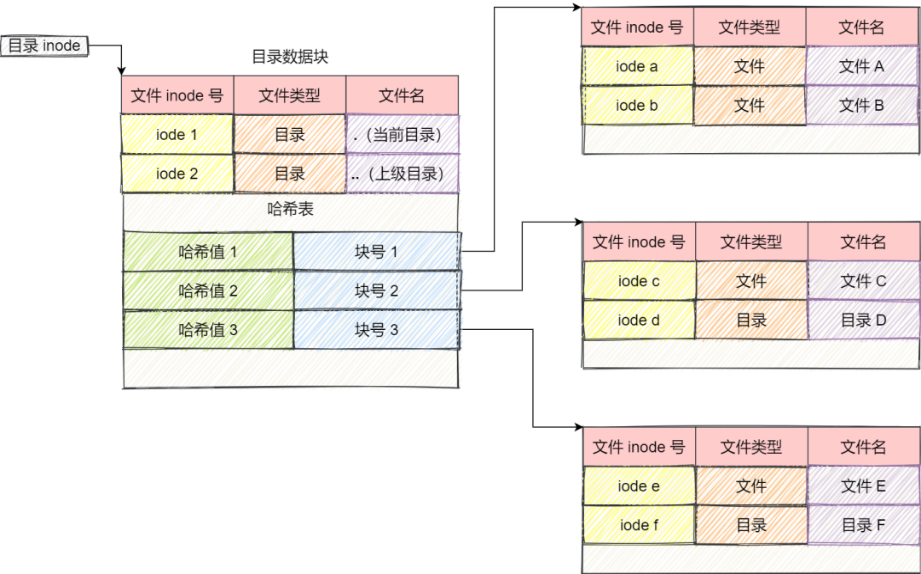
* 超级块，包含的是文件系统的重要信息，比如 inode 总个数、块总个数、每个块组的 inode 个数、每个块组的块个数等等。
* 块组描述符，包含文件系统中各个块组的状态，比如块组中空闲块和 inode 的数目等，每个块组都包含了文件系统中「所有块组的组描述符信息」。
* 数据位图和 inode 位图， 用于表示对应的数据块或 inode 是空闲的，还是被使用中。
* inode 列表，包含了块组中所有的 inode，inode 用于保存文件系统中与各个文件和目录相关的所有元数据。
* 数据块，包含文件的有用数据。

# 目录的存储

和普通文件不同的是，普通文件的块里面保存的是文件数据，而目录文件的块里面保存的是目录里面一项一项的文件信息。

在目录文件的块中，最简单的保存格式就是列表，就是一项一项地将目录下的文件信息（如文件名、文件 inode、文件类型等）列在表里。

列表中每一项就代表该目录下的文件的文件名和对应的 inode，通过这个 inode，就可以找到真正的文件。



通常，第一项是「.」，表示当前目录，第二项是「..」，表示上一级目录，接下来就是一项一项的文件名和 inode。

如果一个目录有超级多的文件，我们要想在这个目录下找文件，按照列表一项一项的找，效率就不高了。

于是，保存目录的格式改成哈希表，对文件名进行哈希计算，把哈希值保存起来，如果我们要查找一个目录下面的文件名，可以通过名称取哈希。如果哈希能够匹配上，就说明这个文件的信息在相应的块里面。

Linux 系统的 ext 文件系统就是采用了哈希表，来保存目录的内容，这种方法的优点是查找非常迅速，插入和删除也较简单，不过需要一些预备措施来避免哈希冲突。

目录查询是通过在磁盘上反复搜索完成，需要不断地进行 I/O 操作，开销较大。所以，为了减少 I/O 操作，把当前使用的文件目录缓存在内存，以后要使用该文件时只要在内存中操作，从而降低了磁盘操作次数，提高了文件系统的访问速度。

# 软链接和硬链接

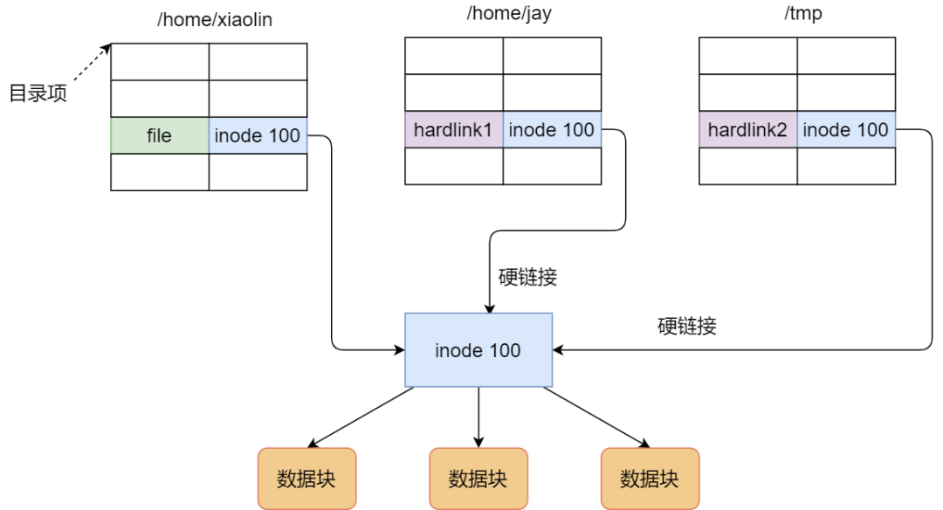
有时候我们希望给某个文件取个别名，那么在 Linux 中可以通过硬链接（Hard Link） 和软链接（Symbolic Link） 的方式来实现

## 硬链接

硬链接是多个目录项中的「索引节点」指向一个文件

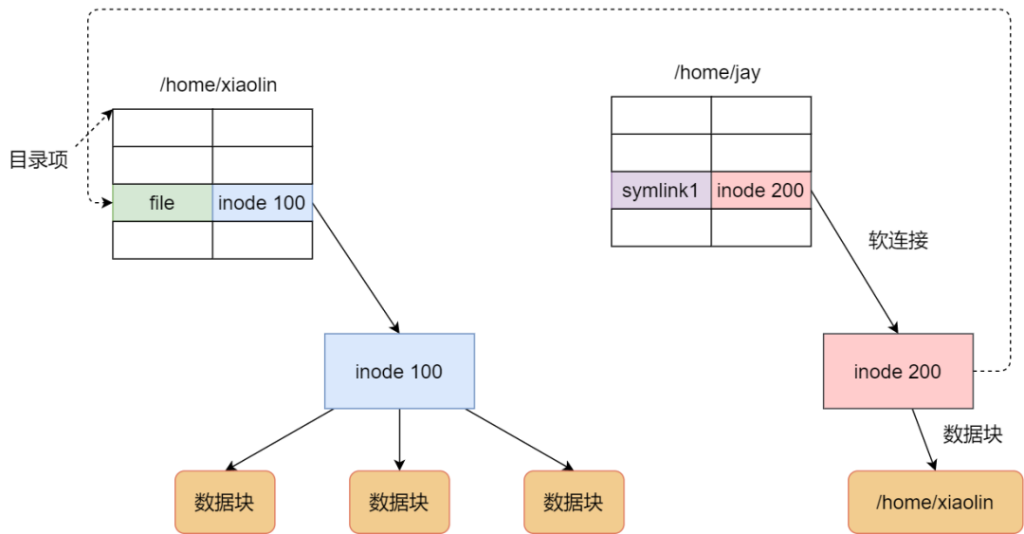
也就是指向同一个 inode，但是 inode 是不可能跨越文件系统的，所以硬链接是不可用于跨文件系统的。

由于多个目录项都是指向一个 inode，那么只有删除文件的所有硬链接以及源文件时，系统才会彻底删除该文件。



## 软链接

软链接相当于重新创建一个文件，这个文件有独立的 inode，但是这个文件的内容是另外一个文件的路径，所以访问软链接的时候，实际上相当于访问到了另外一个文件，所以软链接是可以跨文件系统的，甚至目标文件被删除了，链接文件还是在的，只不过指向的文件找不到了而已。



# 文件 I/O

## 缓冲与非缓冲 I/O

文件操作的标准库是可以实现数据的缓存，那么根据「是否利用标准库缓冲」，可以把文件 I/O 分为缓冲 I/O 和非缓冲 I/O：

缓冲 I/O，利用的是标准库的缓存实现文件的加速访问，而标准库再通过系统调用访问文件。

非缓冲 I/O，直接通过系统调用访问文件，不经过标准库缓存。

这里所说的「缓冲」特指标准库内部实现的缓冲。

比方说，很多程序遇到换行时才真正输出，而换行前的内容，其实就是被标准库暂时缓存了起来，这样做的目的是，减少系统调用的次数，毕竟系统调用是有 CPU 上下文切换的开销的。

## 直接与非直接 I/O

Linux 内核为了减少磁盘 I/O 次数，在系统调用后，会把用户数据拷贝到内核中缓存起来，这个内核缓存空间就是「页缓存 PageCache」，只有当缓存满足某些条件的时候，才发起磁盘 I/O 的请求。

根据是「否利用操作系统的缓存」，可以把文件 I/O 分为直接 I/O 与非直接 I/O：

直接 I/O，不会发生内核缓存和用户程序之间数据复制，而是直接经过文件系统访问磁盘。

* 应用程序已经实现了磁盘数据的缓存，那么可以不需要 PageCache 再次缓存，减少额外的性能损耗。在 MySQL 数据库中，可以通过参数设置开启直接 I/O，默认是不开启；
* 传输大文件的时候，由于大文件难以命中 PageCache 缓存，而且会占满 PageCache 导致「热点」文件无法充分利用缓存，从而增大了性能开销，因此，这时应该使用直接 I/O。

非直接 I/O，读操作时，数据从内核缓存中拷贝给用户程序，写操作时，数据从用户程序拷贝给内核缓存，再由内核决定什么时候写入数据到磁盘。

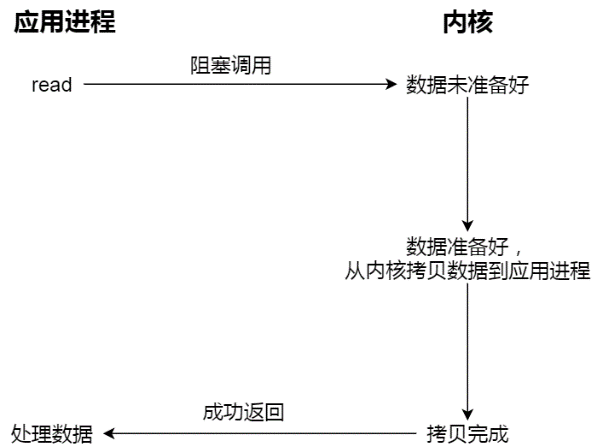
如果用了非直接 I/O 进行写数据操作，内核什么情况下才会把缓存数据写入到磁盘？

* 在调用 write 的最后，当发现内核缓存的数据太多的时候，内核会把数据写到磁盘上；
* 用户主动调用 sync，内核缓存会刷到磁盘上；
* 当内存十分紧张，无法再分配页面时，也会把内核缓存的数据刷到磁盘上；
* 内核缓存的数据的缓存时间超过某个时间时，也会把数据刷到磁盘上；

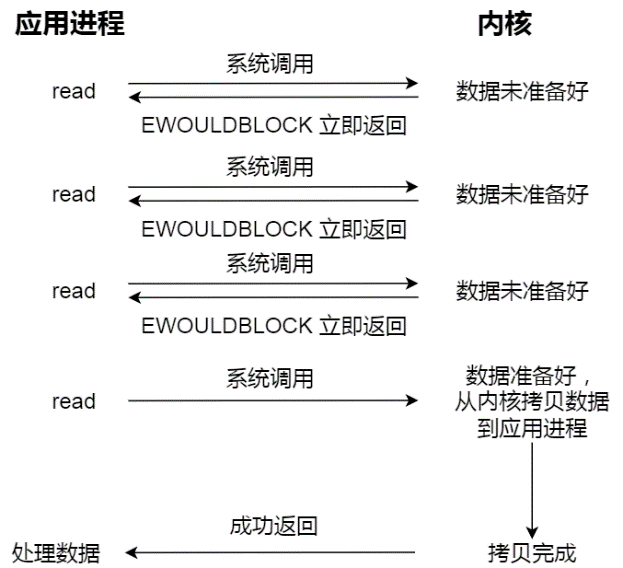
## 阻塞与非阻塞 I/O

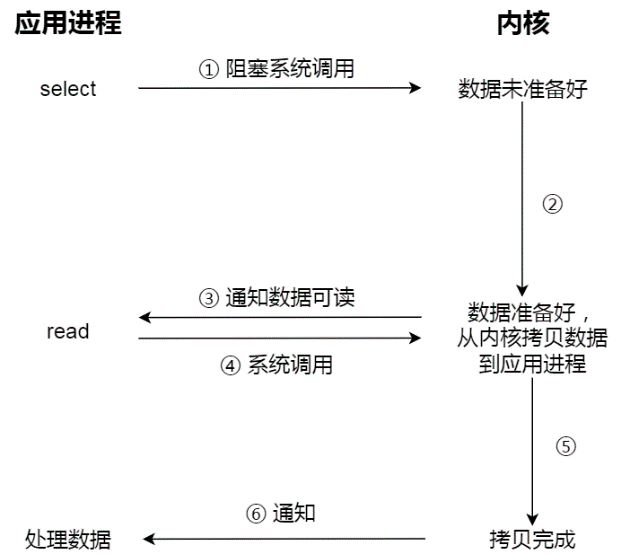
### 阻塞I/O

* 阻塞 I/O，当用户程序执行 read ，线程会被阻塞，一直等到内核数据准备好，并把数据从内核缓冲区拷贝到应用程序的缓冲区中，当拷贝过程完成，read 才会返回。
* 阻塞等待的是「内核数据准备好」和「数据从内核态拷贝到用户态」这两个过程。



### 非阻塞I/O

* 非阻塞的 read 请求在数据未准备好的情况下立即返回，可以继续往下执行，此时应用程序不断轮询内核，直到数据准备好，内核将数据拷贝到应用程序缓冲区，read 调用才可以获取到结果。
* 
* 这里最后一次 read 调用，获取数据的过程，是一个同步的过程，是需要等待的过程。这里的同步指的是内核态的数据拷贝到用户程序的缓存区这个过程。
* 应用程序每次轮询内核的 I/O 是否准备好，感觉有点傻乎乎，因为轮询的过程中，应用程序啥也做不了，只是在循环。
* 为了解决这种傻乎乎轮询方式，于是 I/O 多路复用技术就出来了，如 select、poll，它是通过 I/O 事件分发，当内核数据准备好时，再以事件通知应用程序进行操作。
* 这个做法大大改善了应用进程对 CPU 的利用率，在没有被通知的情况下，应用进程可以使用 CPU 做其他的事情。
* 上图是使用 select I/O 多路复用过程。注意，read 获取数据的过程（数据从内核态拷贝到用户态的过程），也是一个同步的过程，需要等待：

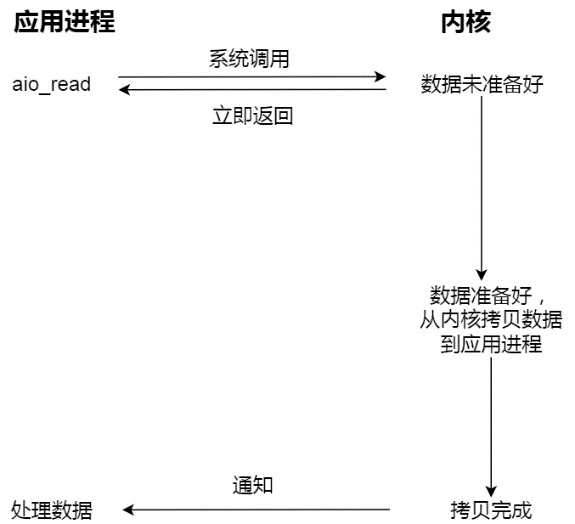


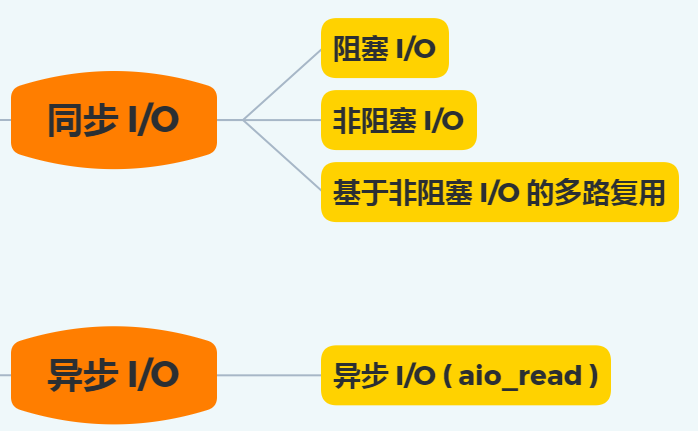
* 实际上，无论是阻塞 I/O、非阻塞 I/O，还是基于非阻塞 I/O 的多路复用都是同步调用。因为它们在 read 调用时，内核将数据从内核空间拷贝到应用程序空间，过程都是需要等待的，也就是说这个过程是同步的，如果内核实现的拷贝效率不高，read 调用就会在这个同步过程中等待比较长的时间。

## 同步与异步 I/O

真正的异步 I/O 是「内核数据准备好」和「数据从内核态拷贝到用户态」这两个过程都不用等待。

当我们发起 aio\_read 之后，就立即返回，内核自动将数据从内核空间拷贝到应用程序空间，这个拷贝过程同样是异步的，内核自动完成的，和前面的同步操作不一样，应用程序并不需要主动发起拷贝动作。过程如下图：





I/O 是分为两个过程的：

* + - 数据准备的过程
    - 数据从内核空间拷贝到用户进程缓冲区的过程

阻塞 I/O 会阻塞在「过程 1 」和「过程 2」，而非阻塞 I/O 和基于非阻塞 I/O 的多路复用只会阻塞在「过程 2」，所以这三个都可以认为是同步 I/O。

异步 I/O 则不同，「过程 1 」和「过程 2 」都不会阻塞。

# Socket通信模型

## 服务端创建过程

1. 首先调用 socket() 函数，创建网络协议为 IPv4，以及传输协议为 TCP 的 Socket接着调用 bind() 函数，给这个 Socket 绑定一个 IP 地址和端口

* 绑定端口的目的：当内核收到 TCP 报文，通过 TCP 头里面的端口号，来找到我们的应用程序
* 绑定 IP 地址的目的：一台机器是可以有多个网卡的，每个网卡都有对应的 IP 地址，当绑定一个网卡时，内核在收到该网卡上的包，才会发给应用程序

2. 然后就可以调用 listen() 函数进行监听

服务端进入了监听状态后，通过调用 accept() 函数，来从内核获取客户端的连接，如果没有客户端连接，则会阻塞等待客户端连接的到来。

## 客户端创建过程

在创建好 Socket 后，调用 connect() 函数发起连接，该函数的参数要指明服务端的 IP 地址和端口号，然后TCP 三次握手就开始了。

## 缺点

### 基本只能一对一通信

### 使用的是同步阻塞的方式

当服务端在还没处理完一个客户端的网络 I/O 时，或者 读写操作发生阻塞时，其他客户端是无法与服务端连接的

## 服务器单机理论最大能连接多少个客户端？

服务器的连接数主要受文件描述符和系统内存两个限制

### 文件描述符

Socket 实际上是一个文件，也就会对应一个文件描述符。在 Linux 下，单个进程打开的文件描述符数是有限制的，没有经过修改的值一般都是 1024，不过我们可以通过 ulimit 增大文件描述符的数目；

### 系统内存

每个 TCP 连接在内核中都有对应的数据结构，意味着每个连接都是会占用一定内存的；

尽管Socket通信也可以在一个进程中打开多个Socket文件，但是还是会受到以上两点的一个限制。想要真正实现能支持并发万级以上的请求，要考虑的地方还是服务器的网络I/O模型，效率低的模型，会加重系统开销，这会成为高并发的一个瓶颈。

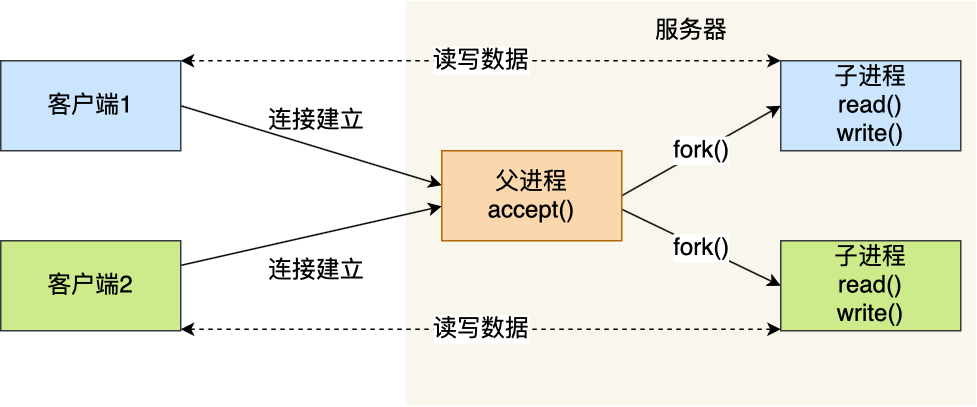
# 多进程模型

比较传统的方式，就是使用多进程模型，也就是为每个客户端分配一个进程来处理请求。

服务器端一旦监听到有客户端的连接，就会accept()并返回一个已连接的用于传输数据的Socket文件描述符，这时，就通过fork() 函数创建一个子进程，实际上就把父进程所有相关的东西都复制一份，包括文件描述符、内存地址空间、程序计数器、执行的代码等。

正因为子进程会复制父进程的文件描述符，于是就可以直接使用「已连接 Socket 」和客户端通信了，

可以发现，子进程不需要关心「监听 Socket」，只需要关心「已连接 Socket」；父进程则相反，将客户服务交给子进程来处理，因此父进程不需要关心「已连接 Socket」，只需要关心「监听 Socket」。



另外，当「子进程」退出时，实际上内核里还会保留该进程的一些信息，也是会占用内存的，如果不做好“回收”工作，就会变成僵尸进程，随着僵尸进程越多，会慢慢耗尽我们的系统资源。

这种用多个进程来应付多个客户端的方式，在应对 少数客户端还是可行的，但是当客户端数量高达一万时，肯定扛不住的，因为每产生一个进程，必会占据一定的系统资源，而且进程间上下文切换比较耗性能。

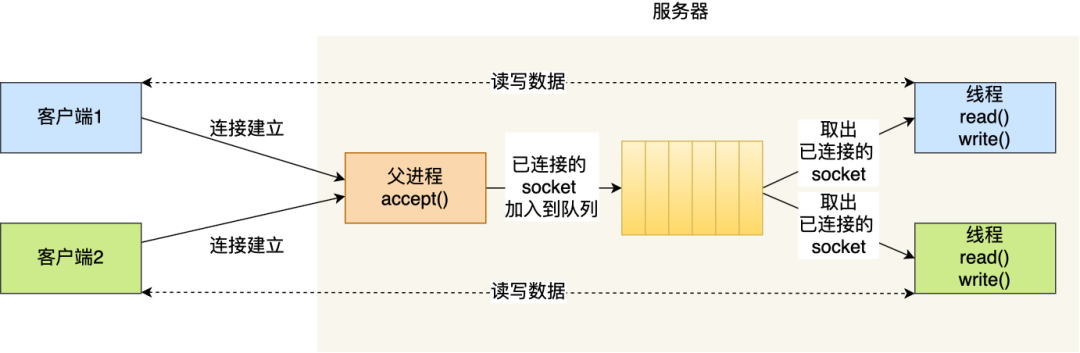
# 多线程模型

线程是运行在进程中的一个“逻辑流”，单进程中可以运行多个线程，同进程里的线程可以共享进程的部分资源的，比如文件描述符列表、进程空间、代码、全局数据、堆、共享库等，这些共享些资源在上下文切换时是不需要切换，而只需要切换线程的私有数据、寄存器等不共享的数据，因此同一个进程下的线程上下文切换的开销要比进程小得多。

当服务器与客户端 TCP 完成连接后，通过 pthread\_create() 函数创建线程，然后将「已连接 Socket」的文件描述符传递给线程函数，接着在线程里和客户端进行通信，从而达到并发处理的目的。

如果每来一个连接就创建一个线程，线程运行完后，还得操作系统还得销毁线程，虽说线程切换的上写文开销不大，但是如果频繁创建和销毁线程，系统开销也是不小的。

我们可以使用线程池的方式来避免线程的频繁创建和销毁，所谓的线程池，就是提前创建若干个线程，这样当由新连接建立时，将这个已连接的 Socket 放入到一个队列里，然后线程池里的线程负责从队列中取出已连接 Socket 进程处理。



需要注意的是，这个队列是全局的，每个线程都会操作，为了避免多线程竞争，线程在操作这个队列前要加锁。

如果要达到 C10K，意味着要一台机器维护 1 万个连接，相当于要维护 1 万个进程/线程，操作系统就算死扛也是扛不住的。

# I/O 多路复用

能不能使用一个进程来维护多个 Socket 呢？那就是 I/O 多路复用技术。

一个进程虽然任一时刻只能处理一个请求，但是处理每个请求的事件时，耗时控制在 1 毫秒以内，这样 1 秒内就可以处理上千个请求，把时间拉长来看，多个请求复用了一个进程，这就是多路复用，这种思想很类似一个 CPU 并发多个进程，所以也叫做时分多路复用。

select/poll/epoll 就是内核提供给用户态的多路复用系统调用，进程可以通过一个系统调用函数从内核中获取多个网络事件。

在获取事件时，先把所有连接（文件描述符）传给内核，再由内核返回产生了事件的连接，然后在用户态中再处理这些连接对应的请求即可。

# select/poll

## select实现方式

将已连接的 Socket 都放到一个文件描述符集合

然后调用 select 函数将文件描述符集合拷贝到内核里，让内核来检查是否有网络事件产生，检查的方式很粗暴，就是通过遍历文件描述符集合的方式

当检查到有事件产生后，将此 Socket 标记为可读或可写， 接着再把整个文件描述符集合拷贝回用户态里

然后用户态还需要再通过遍历的方法找到可读或可写的 Socket，然后再对其处理。

对于 select 这种方式，需要进行 2 次「遍历」文件描述符集合，一次是在内核态里，一个次是在用户态里 ，而且还会发生 2 次「拷贝」文件描述符集合，先从用户空间传入内核空间，由内核修改后，再传出到用户空间中。

select 使用固定长度的 BitsMap，表示文件描述符集合，而且所支持的文件描述符的个数是有限制的，在 Linux 系统中，由内核中的 FD\_SETSIZE 限制， 默认最大值为 1024，只能监听 0~1023 的文件描述符。

## poll实现方式

不再用 BitsMap 来存储所关注的文件描述符，取而代之用动态数组，以链表形式来组织，突破了 select 的文件描述符个数限制，当然还会受到系统文件描述符限制。

poll 和 select 并没有太大的本质区别，都是使用「线性结构」存储进程关注的 Socket 集合，因此都需要遍历文件描述符集合来找到可读或可写的 Socket，时间复杂度为 O(n)，而且也需要在用户态与内核态之间拷贝文件描述符集合，这种方式随着并发数上来，性能的损耗会呈指数级增长。

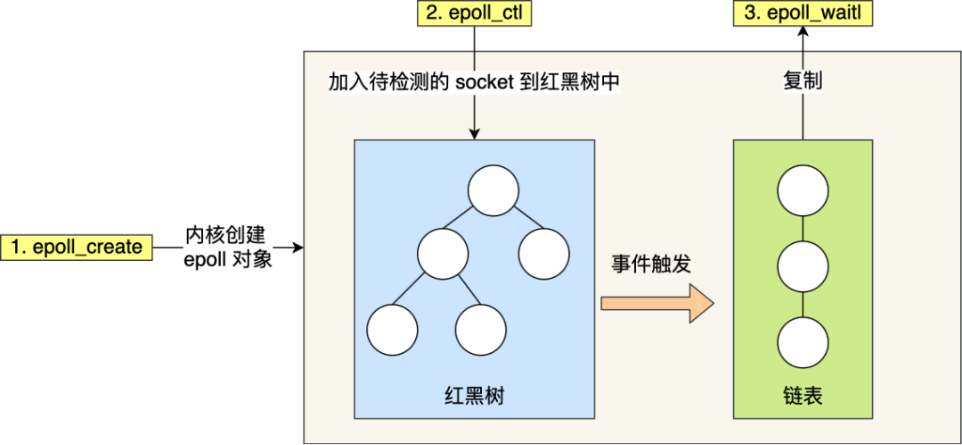
# epoll

## 优化了select/poll 的问题

第一点，epoll 在内核里使用红黑树来跟踪进程所有待检测的文件描述符，把需要监控的 socket 通过 epoll\_ctl() 函数加入内核中的红黑树里，这样增删查一般时间复杂度是 O(logn)，通过对这棵黑红树进行操作，这样就不需要像 select/poll 每次操作时都传入整个 socket 集合，只需要传入一个待检测的 socket，减少了内核和用户空间大量的数据拷贝和内存分配。

第二点， epoll 使用事件驱动的机制，内核里维护了一个链表来记录就绪事件，当某个 socket 有事件发生时，通过回调函数，内核会将其加入到这个就绪事件列表中，当用户调用 epoll\_wait() 函数时，只会返回有事件发生的文件描述符的个数，不需要像 select/poll 那样轮询扫描整个 socket 集合，大大提高了检测的效率。

epoll 的方式即使监听的 Socket 数量越多的时候，效率不会大幅度降低，能够同时监听的 Socket 的数目也非常的多了，上限就为系统定义的进程打开的最大文件描述符个数。因而，epoll 被称为解决 C10K 问题的利器。



## 边缘触发（edge-triggered，ET）

当被监控的 Socket 描述符上有可读事件发生时，服务器端只会从 epoll\_wait 中苏醒一次，即使进程没有调用 read 函数从内核读取数据，也依然只苏醒一次，因此我们程序要保证一次性将内核缓冲区的数据读取完；

## 水平触发（level-triggered，LT）

使用水平触发模式时，当被监控的 Socket 上有可读事件发生时，服务器端不断地从 epoll\_wait 中苏醒，直到内核缓冲区数据被 read 函数读完才结束，目的是告诉我们有数据需要读取；

两者的区别，水平触发的意思是只要满足事件的条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户；而边缘触发的意思是只有第一次满足条件的时候才触发，之后就不会再传递同样的事件了。

如果使用水平触发模式，当内核通知文件描述符可读写时，接下来还可以继续去检测它的状态，看它是否依然可读或可写。所以在收到通知后，没必要一次执行尽可能多的读写操作。

如果使用边缘触发模式，I/O 事件发生时只会通知一次，而且我们不知道到底能读写多少数据，所以在收到通知后应尽可能地读写数据，以免错失读写的机会。因此，我们会循环从文件描述符读写数据，那么如果文件描述符是阻塞的，没有数据可读写时，进程会阻塞在读写函数那里，程序就没办法继续往下执行。所以，边缘触发模式一般和非阻塞 I/O 搭配使用，程序会一直执行 I/O 操作，直到系统调用（如 read 和 write）返回错误，错误类型为 EAGAIN 或 EWOULDBLOCK。

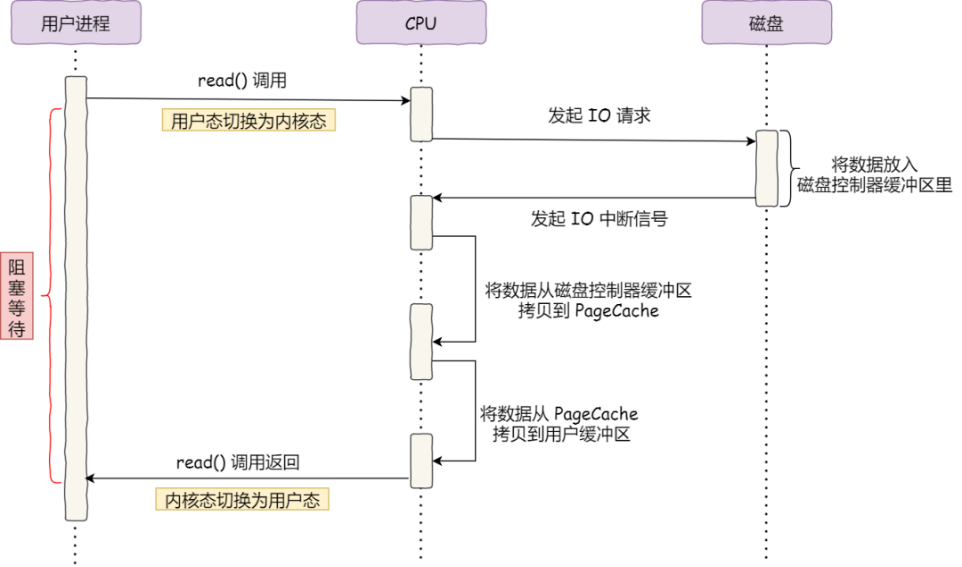
一般来说，边缘触发的效率比水平触发的效率要高，因为边缘触发可以减少 epoll\_wait 的系统调用次数，系统调用也是有一定的开销的的，毕竟也存在上下文的切换。

select/poll 只有水平触发模式，epoll 默认的触发模式是水平触发，但是可以根据应用场景设置为边缘触发模式。

使用 I/O 多路复用时，最好搭配非阻塞 I/O 一起使用

# DMA技术

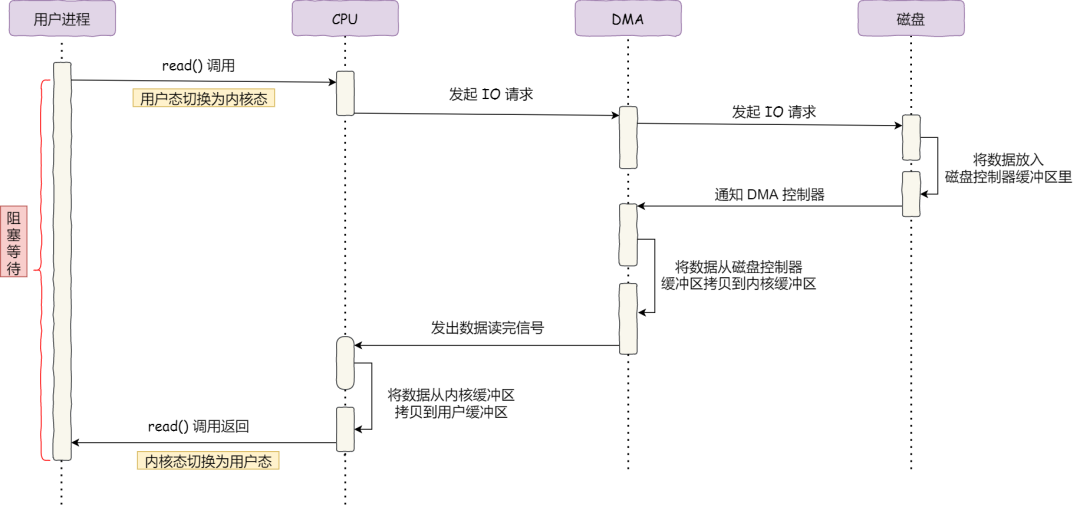
## 不使用 DMA 控制器进行数据传输的过程



整个数据的传输过程，都要需要 CPU 亲自参与搬运数据的过程，而且这个过程，CPU 是不能做其他事情的。

## 使用 DMA 控制器进行数据传输的过程

在进行 I/O 设备和内存的数据传输的时候，数据搬运的工作全部交给 DMA 控制器，而 CPU 不再参与任何与数据搬运相关的事情，这样 CPU 就可以去处理别的事务。



整个数据传输的过程，CPU 不再参与数据搬运的工作，而是全程由 DMA 完成，但是 CPU 在这个过程中也是必不可少的，因为传输什么数据，从哪里传输到哪里，都需要 CPU 来告诉 DMA 控制器。

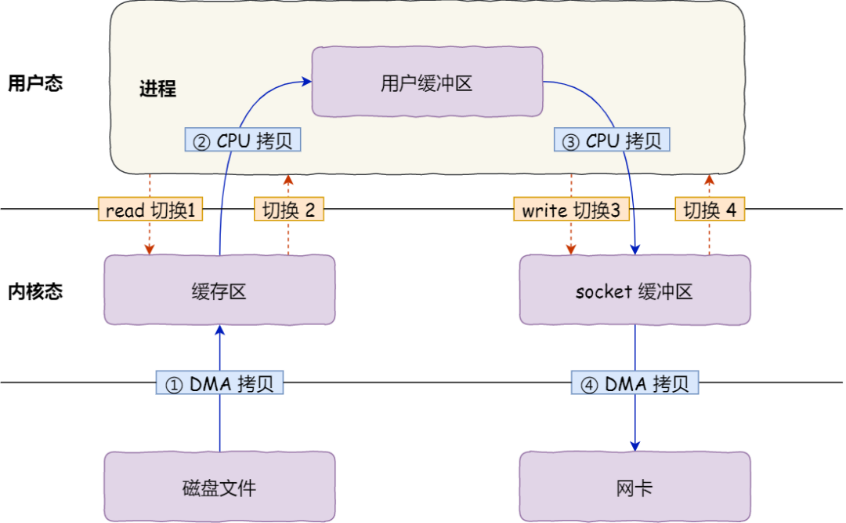
# 传统I/O传输的效率问题

## 传统 I/O 的工作方式一般会需要两个系统调用：

read(file, tmp\_buf, len);

write(socket, tmp\_buf, len);

## 从服务器传输文件的I/O流程



发生了 4 次用户态与内核态的上下文切换

因为发生了两次系统调用，一次是 read() ，一次是 write()，每次系统调用都得先从用户态切换到内核态，等内核完成任务后，再从内核态切换回用户态。

发生了 4 次数据拷贝

其中两次是 DMA 的拷贝，另外两次则是通过 CPU 拷贝的

# 如何优化文件传输的性能

减少「用户态与内核态的上下文切换」和「内存数据拷贝」的次数。

# 如何实现零拷贝

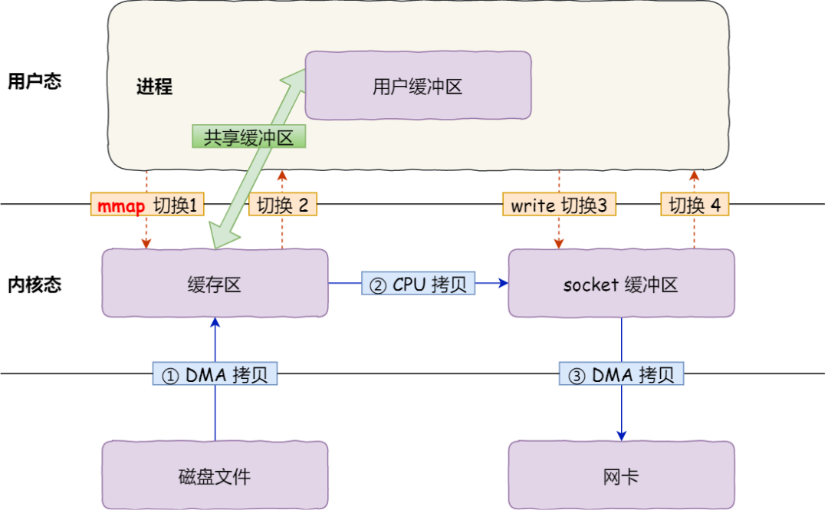
## mmap + write

read() 系统调用的过程中会把内核缓冲区的数据拷贝到用户的缓冲区里，于是为了减少这一步开销，我们可以用 mmap() 替换 read() 系统调用函数。

实现方式

* buf = mmap(file, len);
* write(sockfd, buf, len);

mmap() 系统调用函数会直接把内核缓冲区里的数据「映射」到用户空间，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。



* 应用进程调用了 mmap() 后，DMA 会把磁盘的数据拷贝到内核的缓冲区里。接着，应用进程跟操作系统内核「共享」这个缓冲区；
* 应用进程再调用 write()，操作系统直接将内核缓冲区的数据拷贝到 socket 缓冲区中，这一切都发生在内核态，由 CPU 来搬运数据；
* 最后，把内核的 socket 缓冲区里的数据，拷贝到网卡的缓冲区里，这个过程是由 DMA 搬运的。

通过使用 mmap() 来代替 read()， 可以减少一次数据拷贝的过程。仍然需要 4 次上下文切换

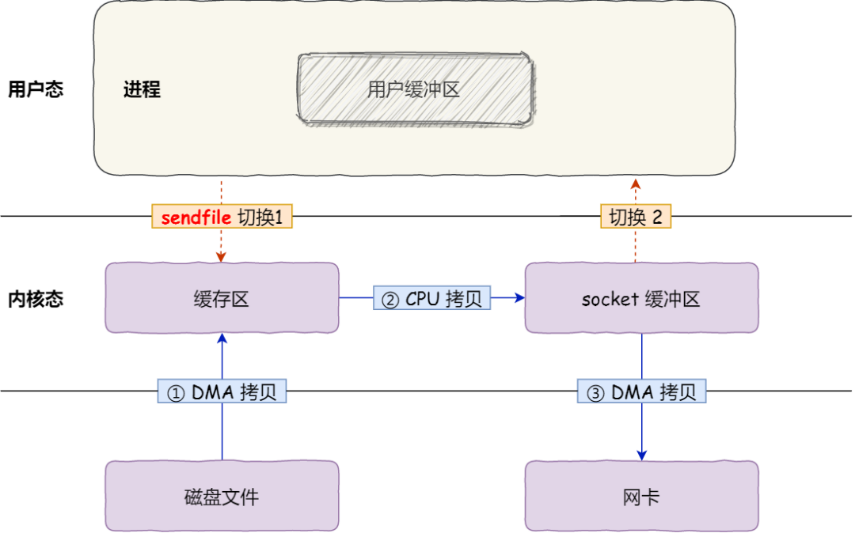
## sendfile

Linux提供了一个专门发送文件的系统调用函数 sendfile()

sendfile表示在两个文件描述符之间传输数据，它是在操作系统内核中操作的，避免了数据从内核缓冲区和用户缓冲区之间的拷贝操作，因此可以使用它来实现零拷贝。

它可以替代前面的 read() 和 write() 这两个系统调用，这样就可以减少一次系统调用，也就减少了 2 次上下文切换的开销。

该系统调用，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。

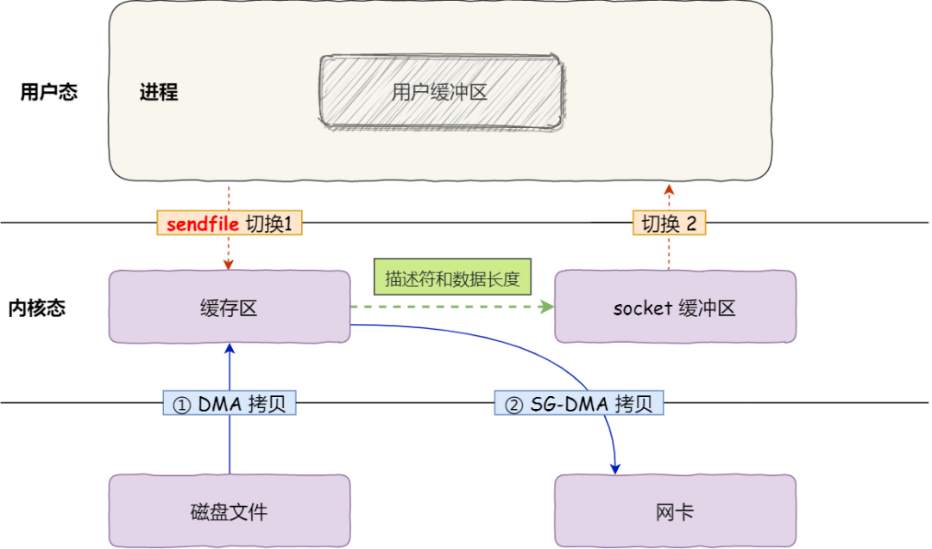


## SG-DMA实现零拷贝

对于支持网卡支持 SG-DMA 技术的情况下， sendfile() 系统调用的过程发生了点变化，具体过程如下：

* 第一步，通过 DMA 将磁盘上的数据拷贝到内核缓冲区里；
* 第二步，缓冲区描述符和数据长度传到 socket 缓冲区，这样网卡的 SG-DMA 控制器就可以直接将内核缓存中的数据拷贝到网卡的缓冲区里，此过程不需要将数据从操作系统内核缓冲区拷贝到 socket 缓冲区中，这样就减少了一次数据拷贝；

SG-DMA技术支持的sendfile()调用的I/O传输过程



所谓的零拷贝（Zero-copy）技术，因为我们没有在内存层面去拷贝数据，也就是说全程没有通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的。

只需要 2 次上下文切换和数据拷贝次数，就可以完成文件的传输，而且 2 次的数据拷贝过程，都不需要通过 CPU，2 次都是由 DMA 来搬运。

# 使用零拷贝的项目

零拷贝适合小文件的传输

Kafka

* 利用了「零拷贝」技术，从而大幅提升了 I/O 的吞吐率，这也是 Kafka 在处理海量数据为什么这么快的原因之一。
* 它调用了 Java NIO 库里的 transferTo方法
* 如果 Linux 系统支持 sendfile() 系统调用，那么 transferTo() 实际上最后就会使用到 sendfile() 系统调用函数。

Nginx

一般默认是开启零拷贝技术，这样有利于提高文件传输的效率

可以配置是否开启零拷贝sendfile选项

* 设置为 on 表示，使用零拷贝技术来传输文件：sendfile ，这样只需要 2 次上下文切换，和 2 次数据拷贝。
* 设置为 off 表示，使用传统的文件传输技术：read + write，这时就需要 4 次上下文切换，和 4 次数据拷贝。

# PageCache有什么作用

## 优点

* 缓存最近被访问的数据；
* 预读功能；

## 大文件传输不能使用

在传输大文件（GB 级别的文件）的时候，PageCache 会不起作用，那就白白浪费 DMA 多做的一次数据拷贝，造成性能的降低，即使使用了 PageCache 的零拷贝也会损失性能

由于文件太大，可能某些部分的文件数据被再次访问的概率比较低，这样就会带来 2 个问题

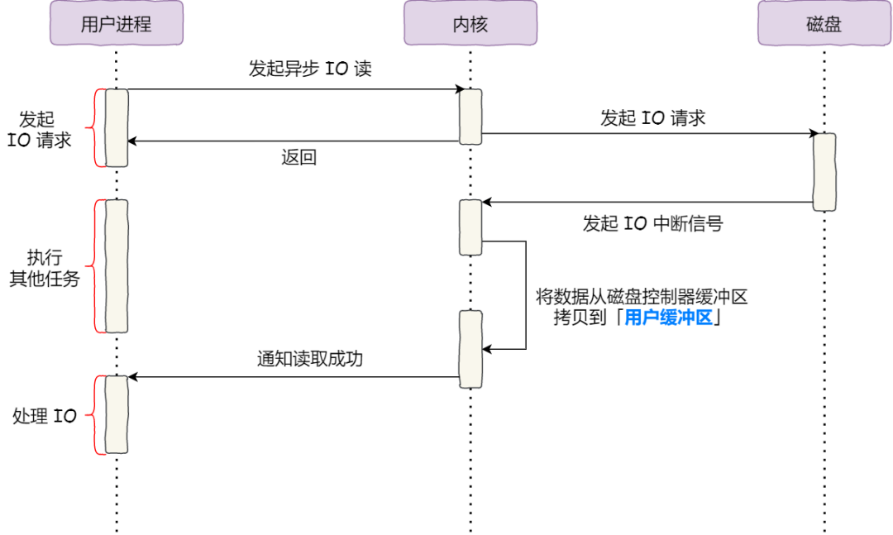
* PageCache 由于长时间被大文件占据，其他「热点」的小文件可能就无法充分使用到 PageCache，于是这样磁盘读写的性能就会下降了；
* PageCache 中的大文件数据，由于没有享受到缓存带来的好处，但却耗费 DMA 多拷贝到 PageCache 一次；

所以，针对大文件的传输，不应该使用 PageCache，也就是说不应该使用零拷贝技术，因为可能由于 PageCache 被大文件占据，而导致「热点」小文件无法利用到 PageCache，这样在高并发的环境下，会带来严重的性能问题。

# 大文件传输方式

在高并发的场景下，针对大文件的传输的方式，应该使用「异步 I/O + 直接 I/O」来替代零拷贝技术。

既然在大文件的传输过程中，用不上PageCache内存缓冲区，那我们就避开这种非直接I/O通信方式，同时为了用户进行在向内核请求I/O操作的时候不阻塞，所以同时使用异步I/O的方式，这样就可以无阻塞的直接从磁盘控制器中读取数据到用户缓冲区中了。



前半部分，内核向磁盘发起读请求，但是可以不等待数据就位就可以返回，于是进程此时可以处理其他任务；

后半部分，当内核将磁盘中的数据拷贝到进程缓冲区后，进程将接收到内核的通知，再去处理数据；

# 代码实现

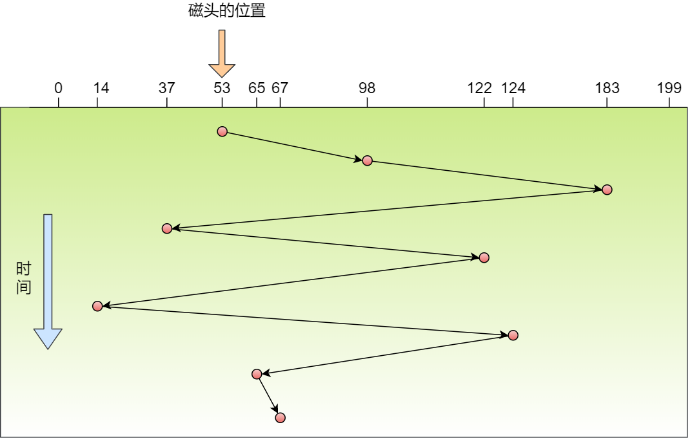


# 磁盘调度算法

## FCFS算法

先到来的请求，先被服务。

访问顺序：98，183，37，122，14，124，65，67



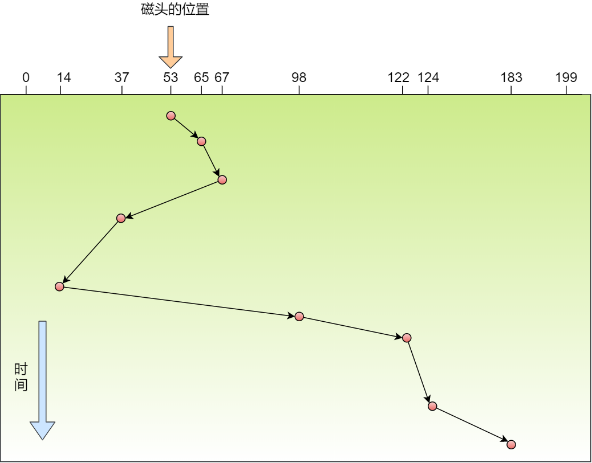
先来先服务算法总共移动了 640 个磁道的距离，这么一看这种算法，比较简单粗暴，但是如果大量进程竞争使用磁盘，请求访问的磁道可能会很分散，那先来先服务算法在性能上就会显得很差，因为寻道时间过长。

## SSTF算法

优先选择离当前磁头全部磁道位置所需寻道时间最短的请求

请求顺序：98，183，37，122，14，124，65，67

那么，那么根据距离磁头（ 53 位置）最近的请求的算法，具体的请求则会是下列从左到右的顺序：65，67，37，14，98，122，124，183

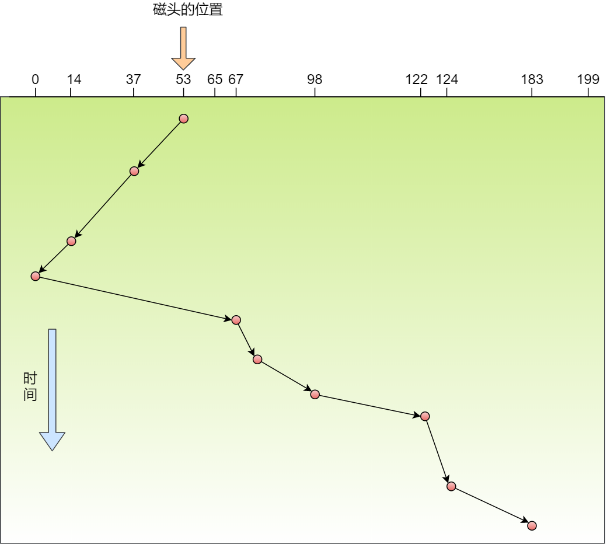


但这个算法可能存在某些请求的饥饿，因为本次例子我们是静态的序列，看不出问题，假设是一个动态的请求，如果后续来的请求都是小于 183 磁道的，那么 183 磁道可能永远不会被响应，于是就产生了饥饿现象，这里产生饥饿的原因是磁头在一小块区域来回移动。

## SCAN算法

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，这就是扫描（Scan）算法

那么，假设扫描调度算法先朝磁道号减少的方向移动，具体请求则会是下列从左到右的顺序：37，14，0，65，67，98，122，124，183

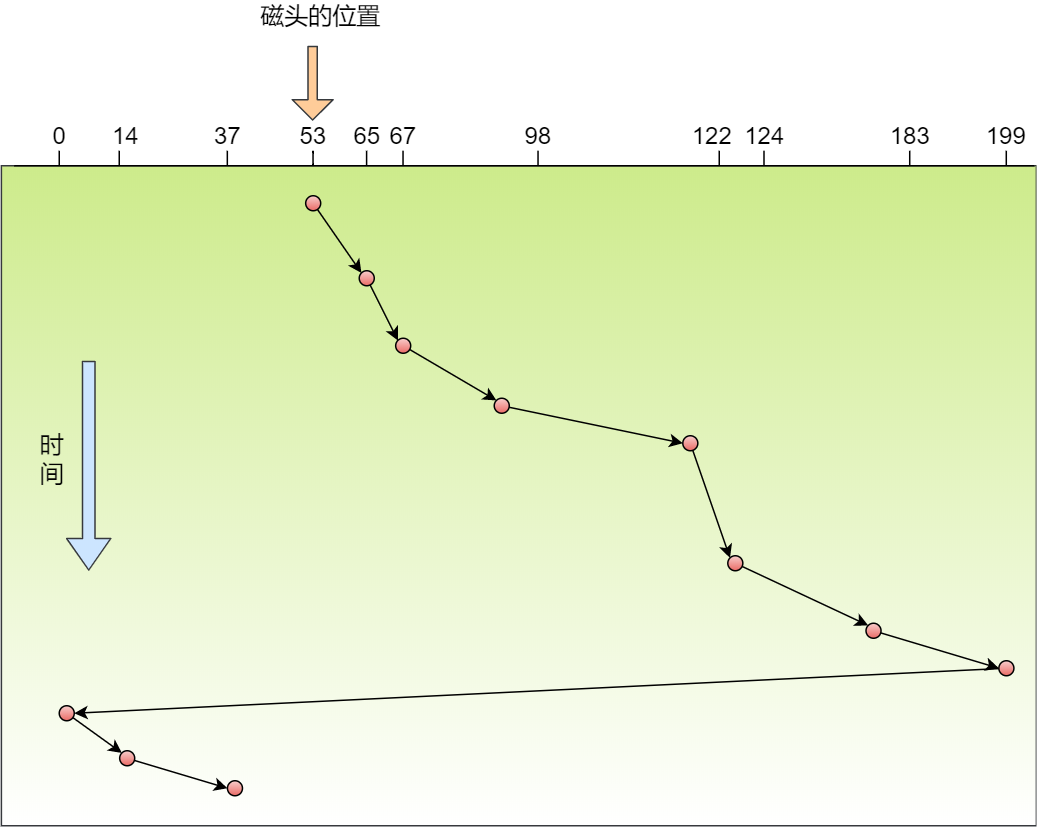


扫描调度算法性能较好，不会产生饥饿现象，但是存在这样的问题，中间部分的磁道会比较占便宜，中间部分相比其他部分响应的频率会比较多，也就是说每个磁道的响应频率存在差异。

## Circular SCAN算法

磁头总是按相同的方向进行扫描，使得每个磁道的响应频率基本一致。

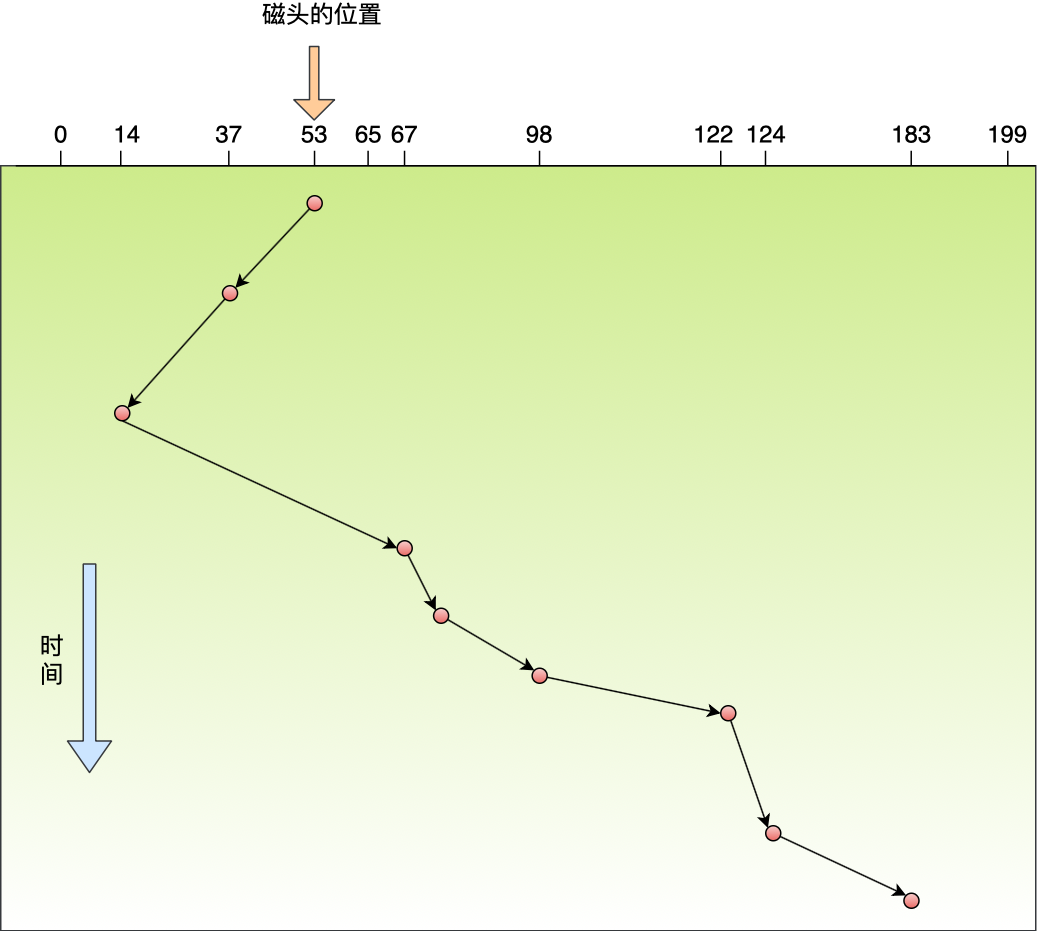
只有磁头朝某个特定方向移动时，才处理磁道访问请求，而返回时直接快速移动至最靠边缘的磁道，也就是复位磁头，这个过程是很快的，并且返回中途不处理任何请求，该算法的特点，就是磁道只响应一个方向上的请求。



循环扫描算法相比于扫描算法，对于各个位置磁道响应频率相对比较平均。

## Look算法

是磁头在移动到「最远的请求」位置，然后立即反向移动。



## C-Look算法

而针 C-SCAN 算法的优化则叫 C-LOOK，它的工作方式，磁头在每个方向上仅仅移动到最远的请求位置，然后立即反向移动，而不需要移动到磁盘的最始端或最末端，反向移动的途中不会响应请求。

