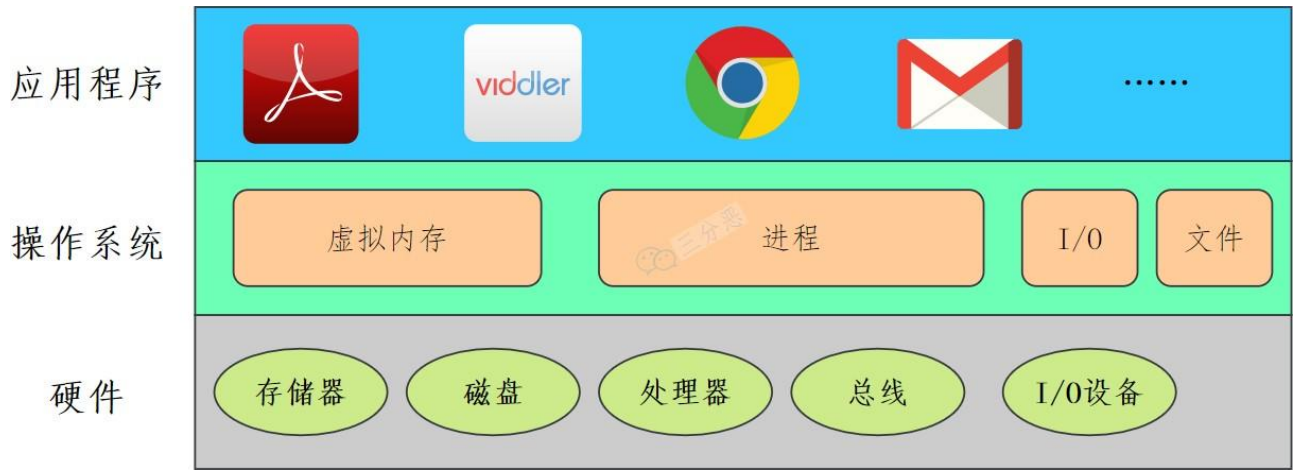
> 图文详解操作系统面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。

引论

什么是操作系统？

可以这么说，操作系统是一种运行在内核态的软件。

它是应用程序和硬件之间的媒介，向应用程序提供硬件的抽象，以及管理硬件资源。

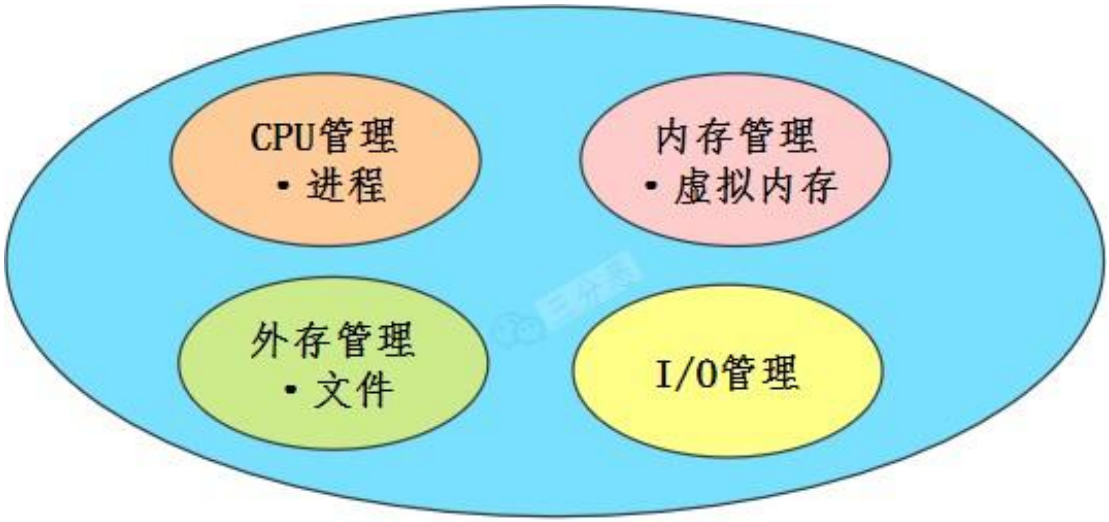


操作系统主要有哪些功能？

操作系统最主要的功能：

- 处理器（CPU）管理：CPU的管理和分配，主要指的是进程管理。内存
- 管理：内存的分配和管理，主要利用了虚拟内存的方式。
- 外存管理：外存（磁盘等）的分配和管理，将外存以文件的形式提供出去。
- I/O管理：对输入/输出设备的统一管理。

除此之外，还有保证自身正常运行的健壮性管理，防止非法操作和入侵的安全性管理。



操作系统主要功能

操作系统结构

什么是内核？

可以这么说，内核是一个计算机程序，它是操作系统的核心，提供了操作系统最核心的能力，可以控制操作系统中所有的内容。

什么是用户态和内核态？

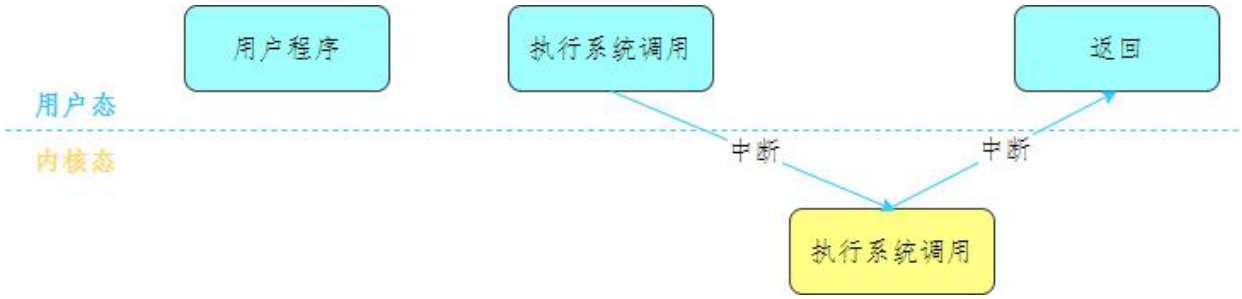
内核具有很高的权限，可以控制 cpu、内存、硬盘等硬件，出于权限控制的考虑，因此大多数操作系统，把内存分成了两个区域：

- 内核空间，这个内存空间只有内核程序可以访问；
- 用户空间，这个内存空间专门给应用程序使用，权限比较小；

用户空间的代码只能访问一个局部的内存空间，而内核空间的代码可以访问所有内存空间。因此，当程序使用用户空间时，我们常说该程序在**用户态**执行，而当程序使内核空间时，程序则在**内核态**执行。

用户态和内核态是如何切换的？

应用程序如果需要进入内核空间，就需要通过系统调用，来进入内核态：



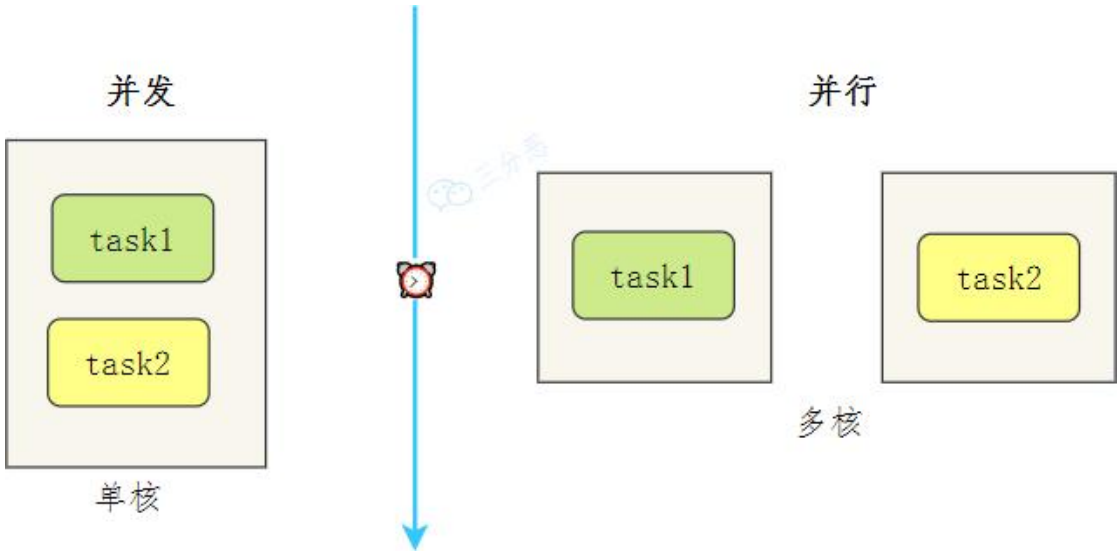
内核程序执行在内核态，用户程序执行在用户态。当应用程序使用系统调用时，会产生一个中断。发生中断后，CPU 会中断当前在执行的用户程序，转而跳转到中断处理程序，也就是开始执行内核程序。内核处理完后，主动触发中断，把 CPU 执行权限交回给用户程序，回到用户态继续工作。

进程和线程

并行和并发有什么区别？

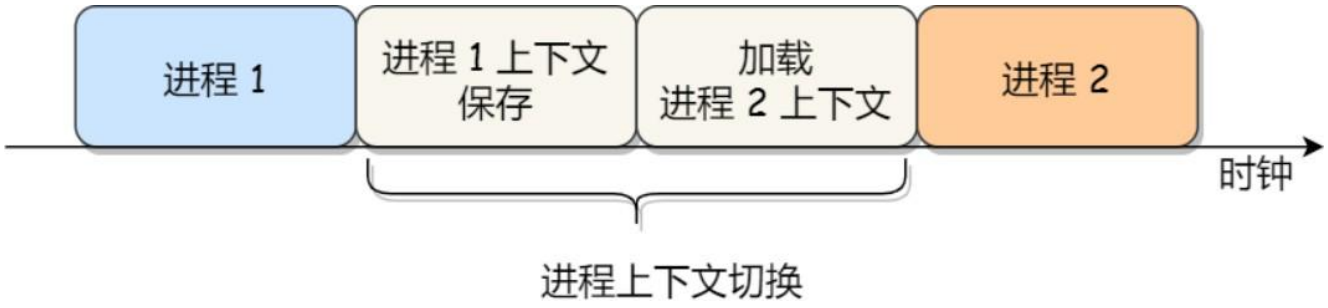
并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务在执行。单核处理器做到的并发，其实是利用时间片的轮转，例如有两个进程A和B，A运行一个时间片之后，切换到B，B运行一个时间片之后又切换到A。因为切换速度足够快，所以宏观上表现为在一段时间内能同时运行多个程序。

并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。



什么是进程上下文切换？

对于单核单线程 CPU 而言，在某一时刻只能执行一条 CPU 指令。上下文切换 (Context Switch) 是一种将 CPU 资源从一个进程分配给另一个进程的机制。从用户角度看，计算机能够并行运行多个进程，这恰恰是操作系统通过快速上下文切换造成的结果。在切换的过程中，操作系统需要先存储当前进程的状态 (包括内存空间的指针，当前执行完的指令等等)，再读入下一个进程的状态，然后执行此进程。

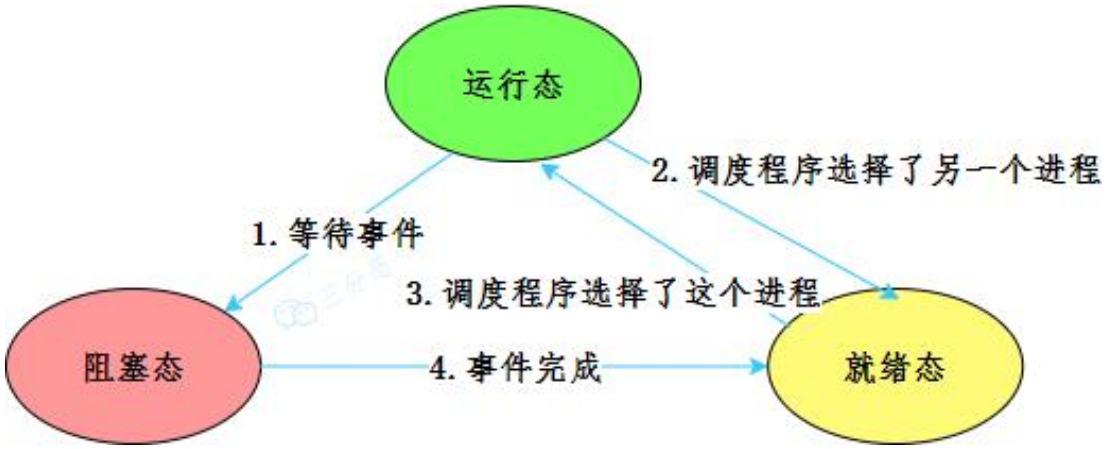


进程有哪些状态？

当一个进程开始运行时，它可能会经历下面这几种状态：

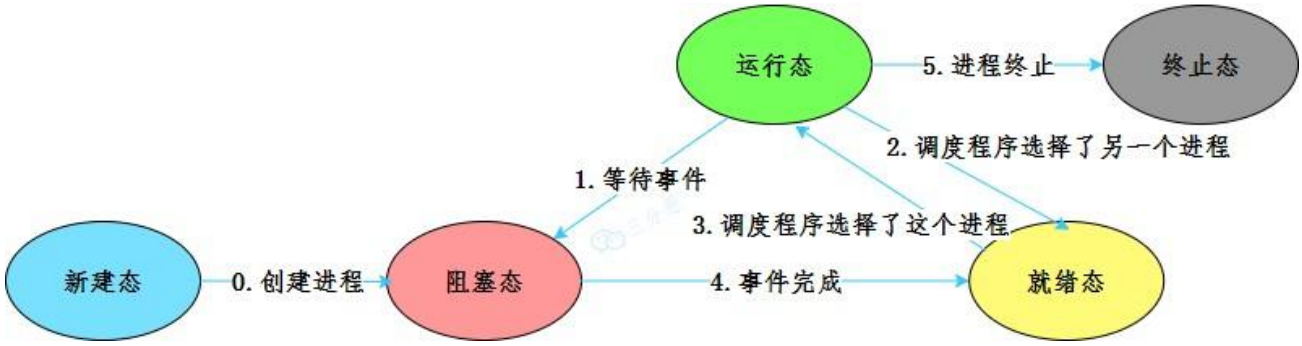
上图中各个状态的意义：

- 运行状态（*Runing*）：该时刻进程占用 CPU；
- 就绪状态（*Ready*）：可运行，由于其他进程处于运行状态而暂时停止运行；
- 阻塞状态（*Blocked*）：该进程正在等待某一事件发生（如等待输入/输出操作的完成）而暂时停止运行，这时，即使给它CPU控制权，它也无法运行；



当然，进程还有另外两个基本状态：

- 创建状态（*new*）：进程正在被创建时的状态；
- 结束状态（*Exit*）：进程正在从系统中消失时的状态；



什么是僵尸进程？

僵尸进程是已完成且处于终止状态，但在进程表中却仍然存在的进程。

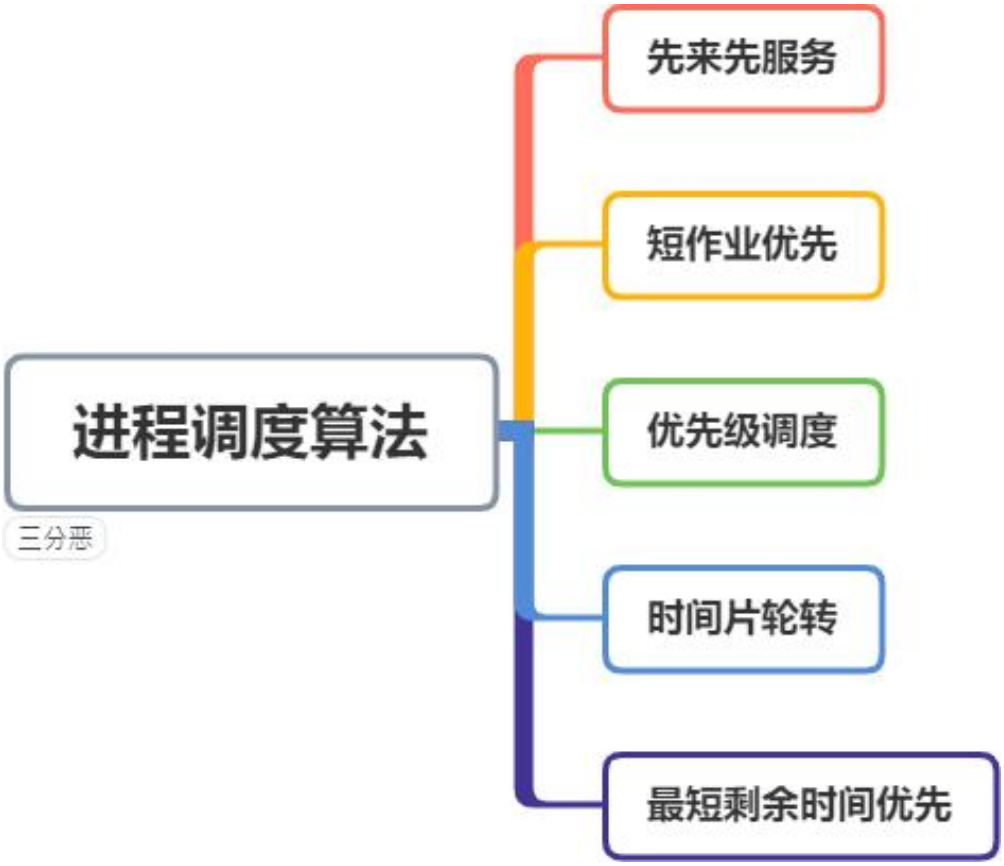
僵尸进程一般发生有父子关系的进程中，一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过 wait() 或 waitpid() 获取了子进程信息后才会释放。如果子进程退出，而父进程并没有调用 wait() 或 waitpid()，那么子进程的进程描述符仍然保存在系统中。

什么是孤儿进程？

一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。孤儿进程将被 init 进程 (进程 ID 为 1 的进程) 所收养，并由 init 进程对它们完成状态收集工作。因为孤儿进程会被 init 进程收养，所以孤儿进程不会对系统造成危害。

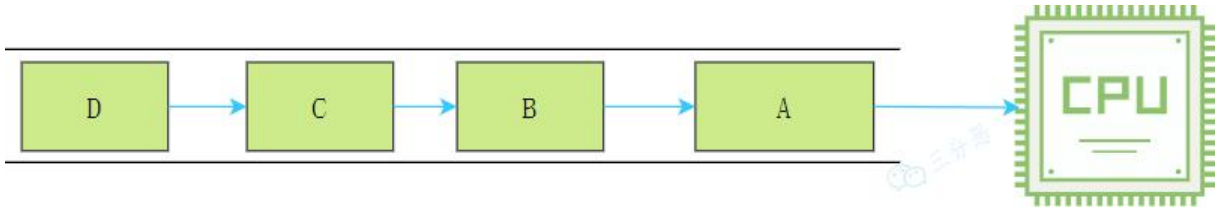
进程有哪些调度算法？

进程调度就是确定某一个时刻CPU运行哪个进程，常见的进程调度算法有：



- **先来先服务**

非抢占式的调度算法，按照请求的顺序进行调度。有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。另外，对I/O密集型进程也不利，因为这种进程每次进行I/O操作之后又得重新排队。

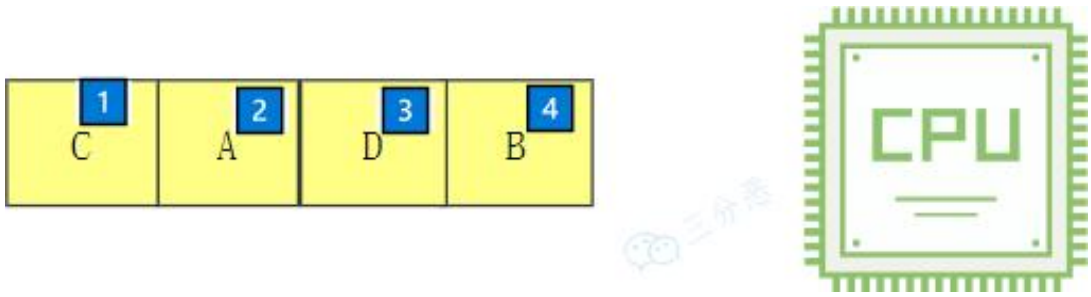


- **短作业优先**

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

- **优先级调度**

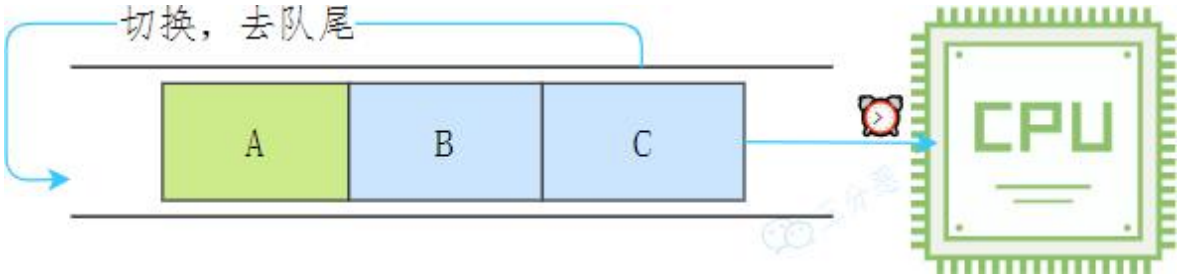
为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。



- **时间片轮转**

将所有就绪进程按 先来先服务的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

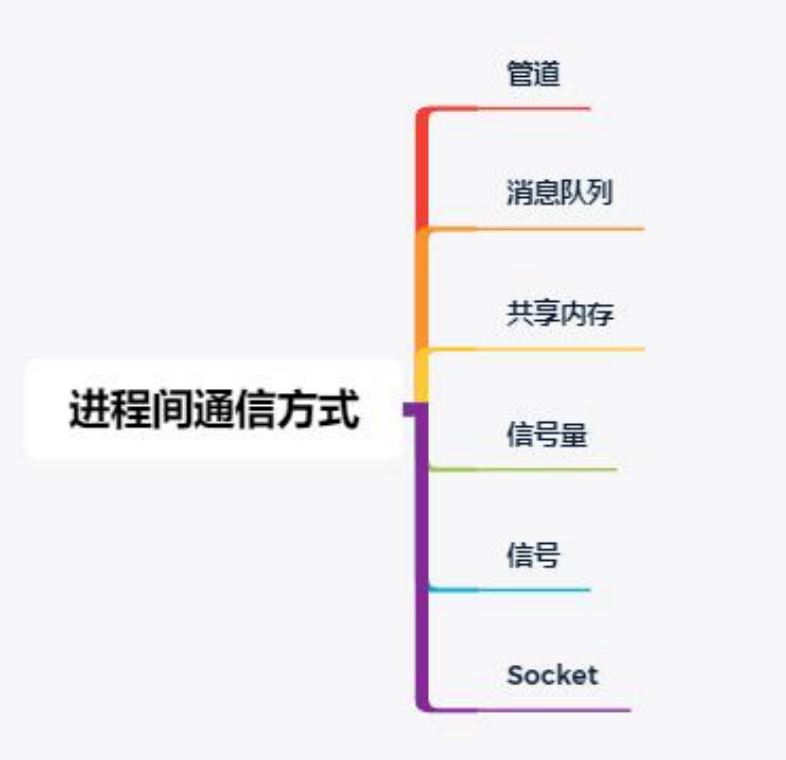
时间片轮转算法的效率和时间片的大小有很大关系：因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。 而如果时间片过长，那么实时性就不能得到保证。



- **最短剩余时间优先**

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。 当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

进程间通信有哪些方式？



- 管道：管道可以理解成不同进程之间的对白，一方发声，一方接收，声音的介质可是是空气或者电缆，进程之间就可以通过管道，**所谓的管道就是内核中的一串缓存**，从管道的一端写入数据，就是缓存在了内核里，另一端读取，也是从内核中读取这段数据。

  管道可以分为两类：**匿名管道**和**命名管道**。匿名管道是单向的，只能在有亲缘关系的进程间通信；命名管道是双向的，可以实现本机任意两个进程通信。

- 信号 ： 信号可以理解成一种电报，发送方发送内容，指定接收进程，然后发出特定的软件中断，操作系统接到中断请求后，找到接收进程，通知接收进程处理信号。

比如kill -9 1050就表示给PID为1050的进程发送SIGKIL信号。Linux系统中常用信号：

(1) SIGHUP：用户从终端注销，所有已启动进程都将收到该进程。系统缺省状态下对该信号的处理是终止进程。

(2) SIGINT：程序终止信号。程序运行过程中，按Ctrl+C键将产生该信号。

(3) SIGQUIT：程序退出信号。程序运行过程中，按Ctrl+\键将产生该信号。

(4) SIGBUS和SIGSEGV：进程访问非法地址。

(5) SIGFPE：运算中出现致命错误，如除零操作、数据溢出等。

(6) SIGKILL：用户终止进程执行信号。shell下执行kill -9发送该信号。

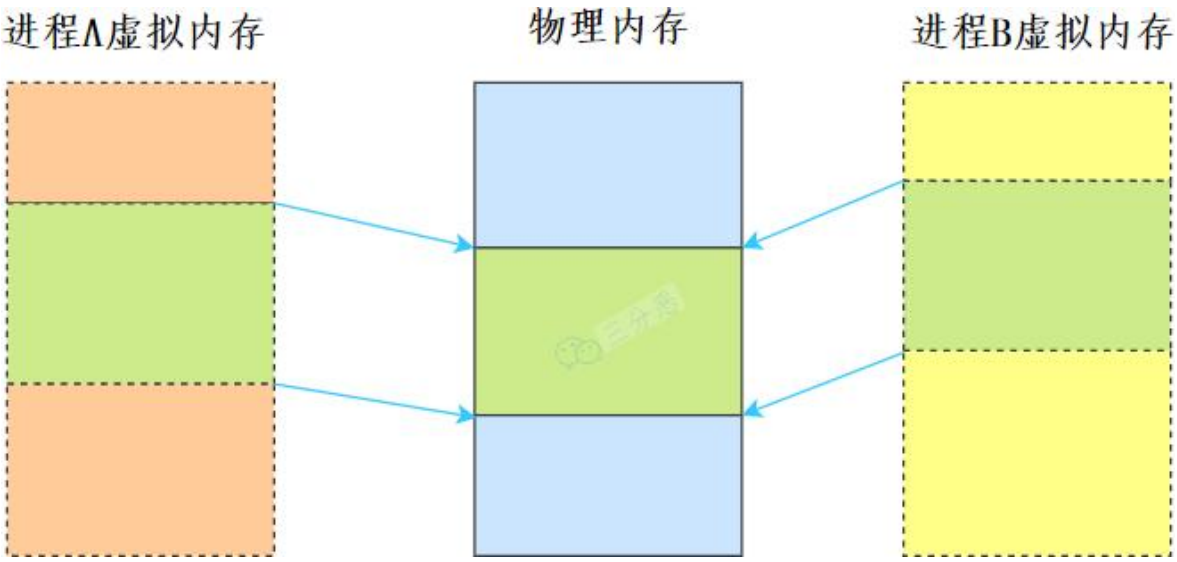
(7) SIGTERM：结束进程信号。shell下执行kill 进程pid发送该信号。

(8) SIGALRM：定时器信号。

(9) SIGCLD：子进程退出信号。如果其父进程没有忽略该信号也没有处理该信号，则子进程退出后将形成僵尸进程。

- 消息队列：**消息队列就是保存在内核中的消息链表**，包括Posix消息队列和System V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

- 共享内存：**共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中**。这样这个进程写入的东西，另外的进程马上就能看到。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

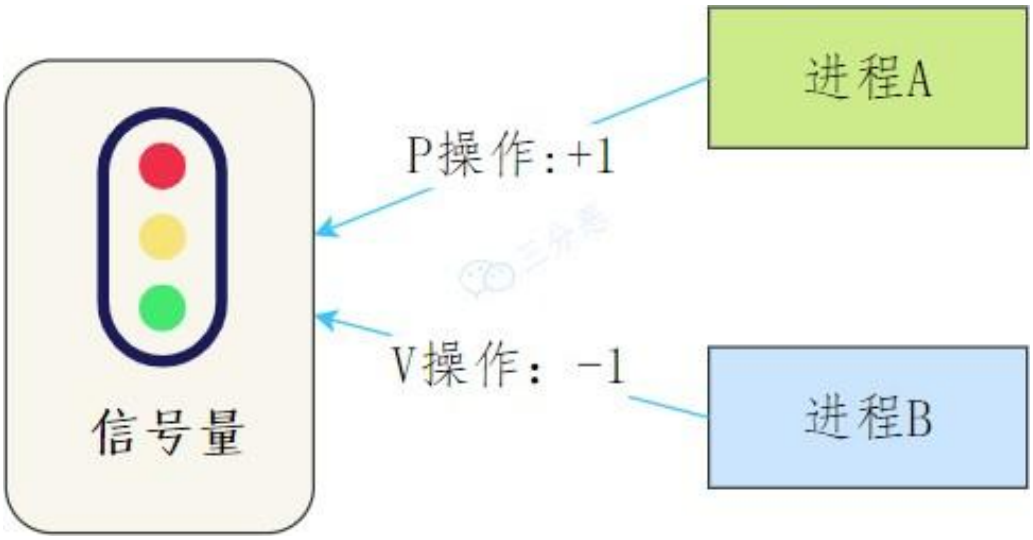


- 信号量：信号量我们可以理解成红绿灯，红灯行，绿灯停。**它本质上是一个整数计数器**，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

  信号量表示资源的数量，控制信号量的方式有两种原子操作：

  - 一个是 P **操作**，这个操作会把信号量减去 1，相减后如果信号量 < 0，则表明资源已被占用，进程需阻塞等待；相减后如果信号量>= 0，则表明还有资源可使用，进程可正常继续执行。
  - 另一个是 V **操作**，这个操作会把信号量加上 1，相加后如果信号量 <= 0，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0，则表明当前没有阻塞中的进程；

  P 操作是用在进入共享资源之前，V 操作是用在离开共享资源之后，这两个操作是必须成对出现的。



- Socket：与其他通信机制不同的是，它可用于不同机器间的进程通信。优

缺点：

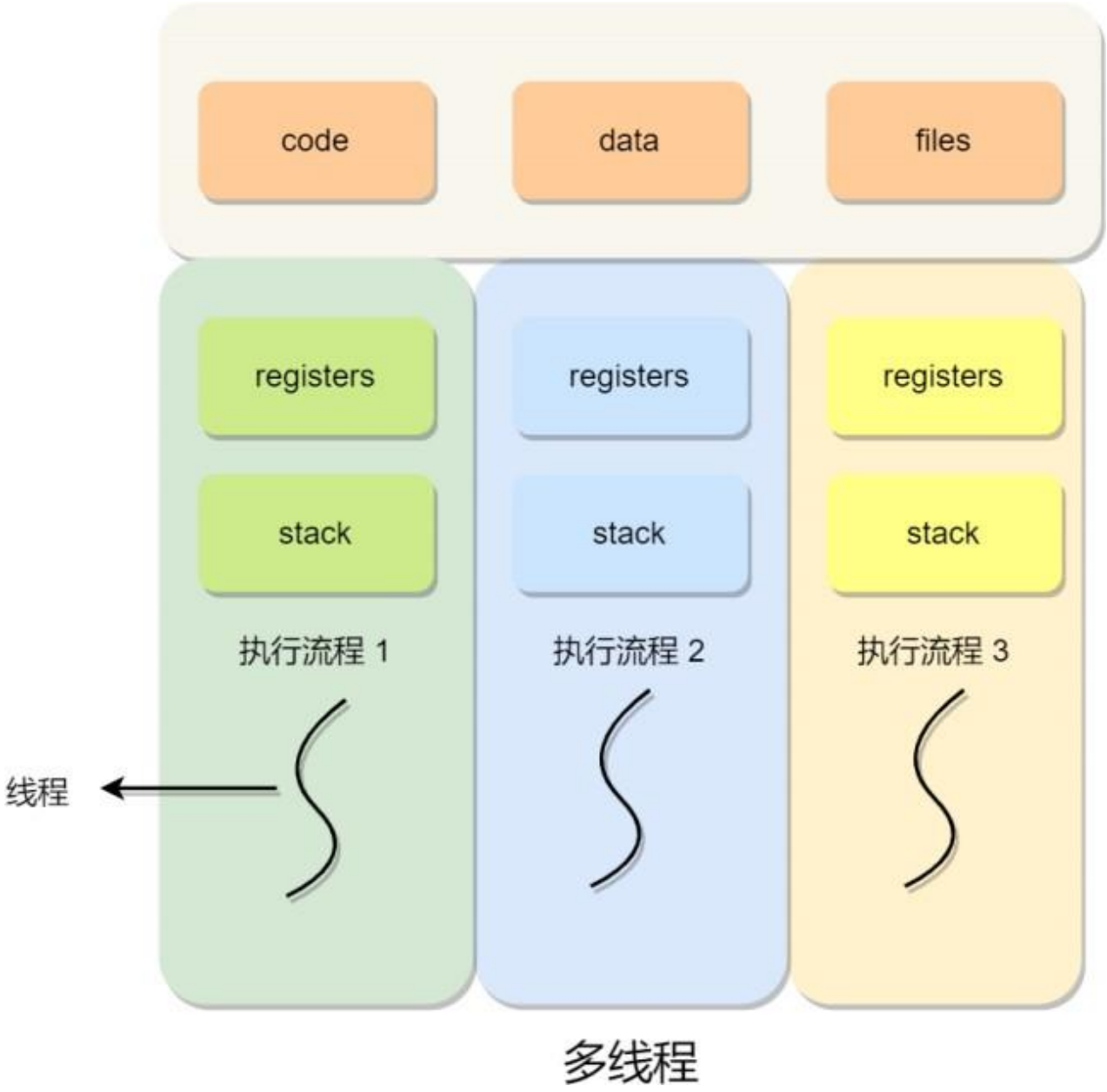
- 管道：简单；效率低，容量有限；
- 消息队列：不及时，写入和读取需要用户态、内核态拷贝。
- 共享内存区：能够很容易控制容量，速度快，但需要注意不同进程的同步问题。信
- 号量：不能传递复杂消息，一般用来实现进程间的同步；
- 信号：它是进程间通信的唯一异步机制。
- Socket：用于不同主机进程间的通信。

进程和线程的联系和区别？

线程和进程的联系：

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，但每个线程各自都有一套独立的寄存器和栈，这样可以确保线程的控制流是相对独立的。



线程与进程的比较如下：

- 调度：**进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位**；资
- 源：进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；
- 拥有资源：线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；
- 系统开销：线程能减少并发执行的时间和空间开销——创建或撤销进程时，系统都要为之分配或回收系统资源，如内存空间，I/O设备等，OS所付出的开销显著大于在创建或撤销线程时的开销，进程切换的开销也远大于线程切换的开销。

线程上下文切换了解吗？

这还得看线程是不是属于同一个进程：

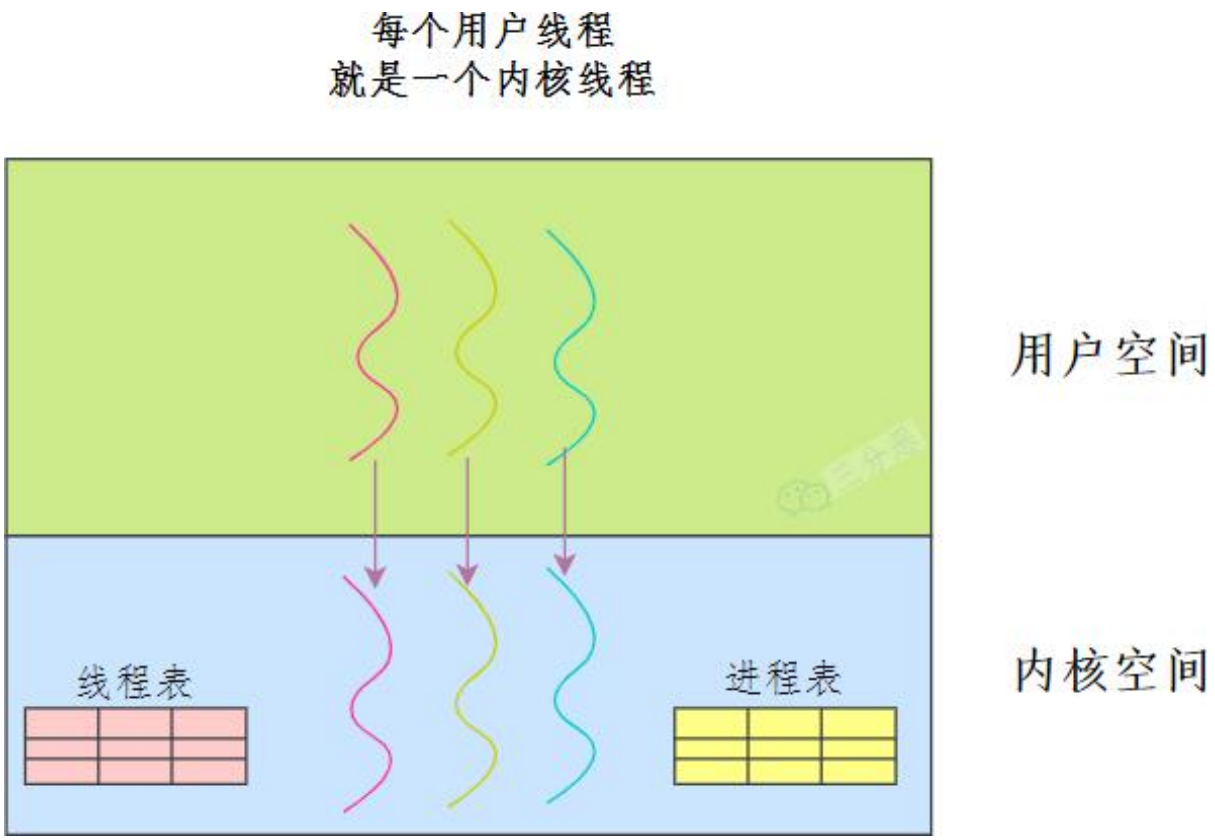
- 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；
- **当两个线程是属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据**；

所以，线程的上下文切换相比进程，开销要小很多。

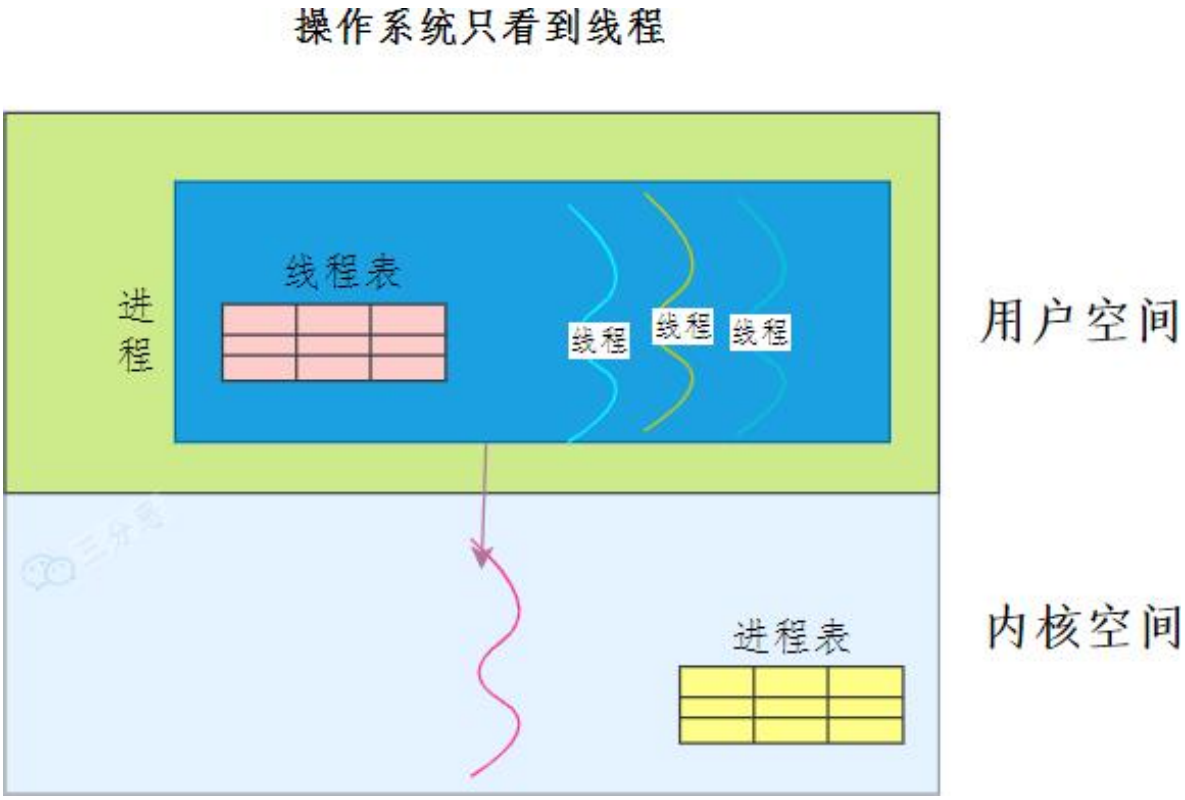
线程有哪些实现方式？

主要有三种线程的实现方式：

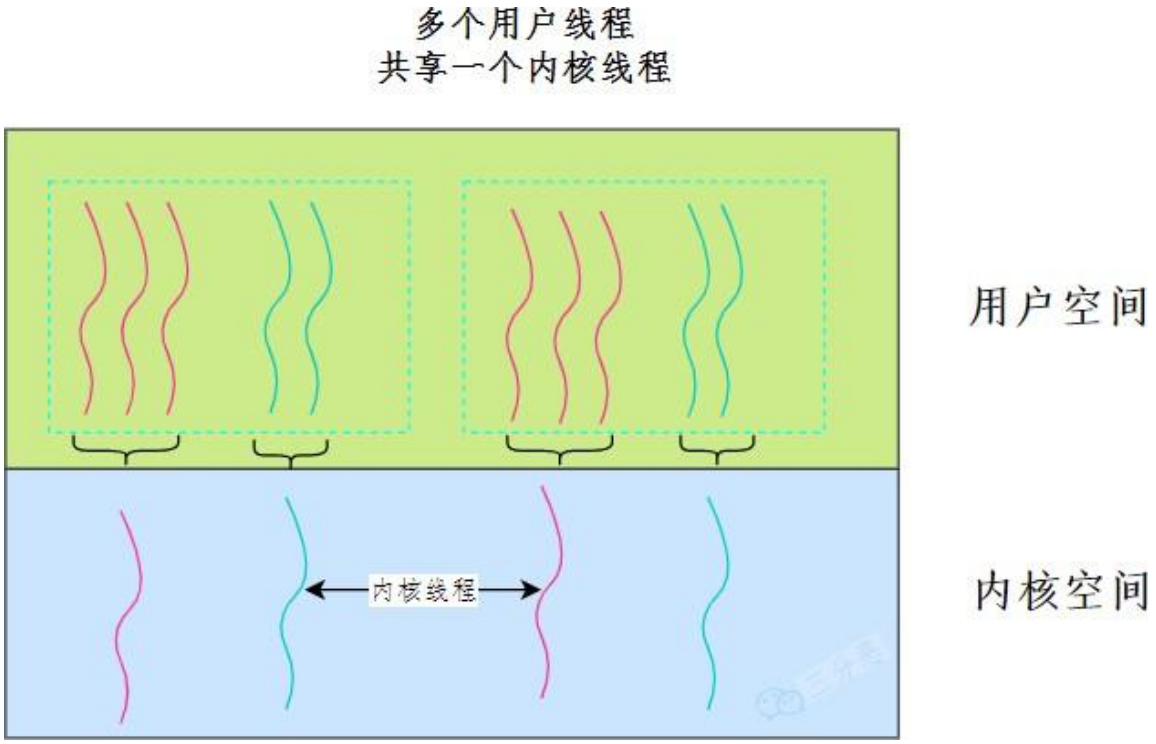
- **内核态线程实现**：在内核空间实现的线程，由内核直接管理直接管理线程。



- **用户态线程实现**：在用户空间实现线程，不需要内核的参与，内核对线程无感知。



- **混合线程实现**：现代操作系统基本都是将两种方式结合起来使用。用户态的执行系统负责进程内部线程在非阻塞时的切换；内核态的操作系统负责阻塞线程的切换。即我们同时实现内核态和用户态线程管理。其中内核态线程数量较少，而用户态线程数量较多。每个内核态线程可以服务一个或多个用户态线程。

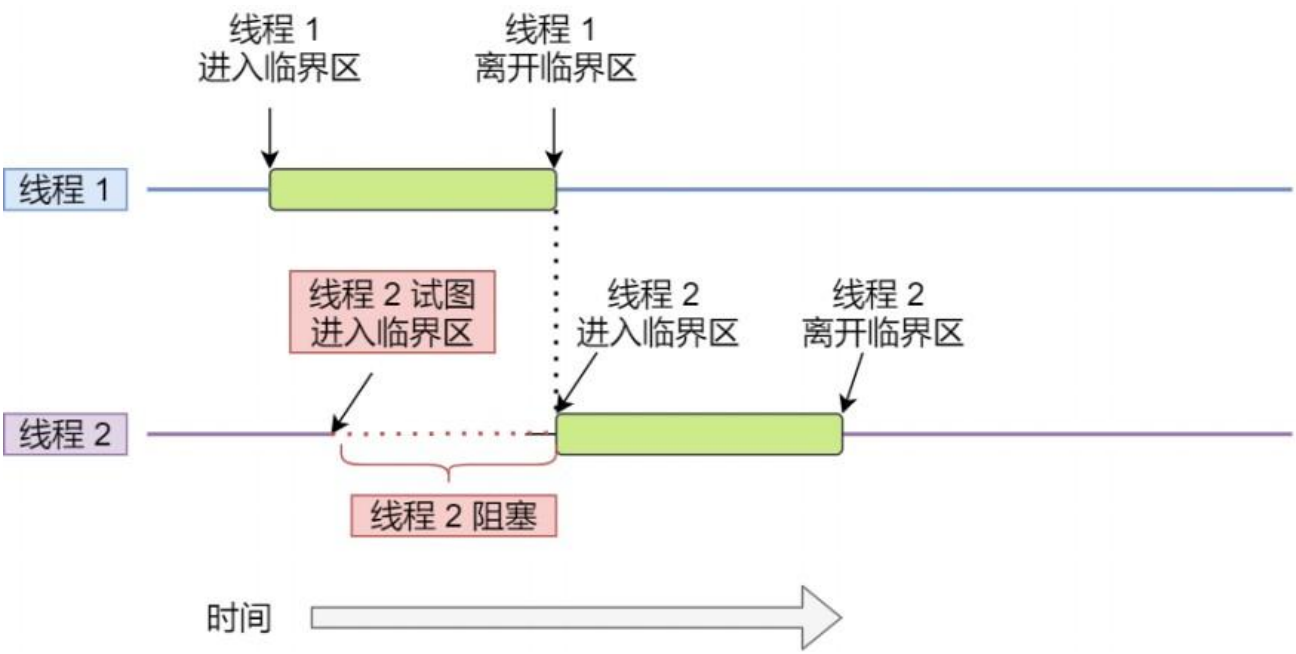


线程间如何同步？

同步解决的多线程操作共享资源的问题，目的是不管线程之间的执行如何穿插，最后的结果都是正确的。

我们前面知道线程和进程的关系：线程是进程当中的一条执行流程。所以说下面的一些同步机制不止针对线程，同样也可以针对进程。

临界区：我们把对共享资源访问的程序片段称为临界区，我们希望这段代码是互斥的，保证在某时刻只能被一个线程执行，也就是说一个线程在临界区执行时，其它线程应该被阻止进入临界区。



临界区不仅针对线程，同样针对进程。

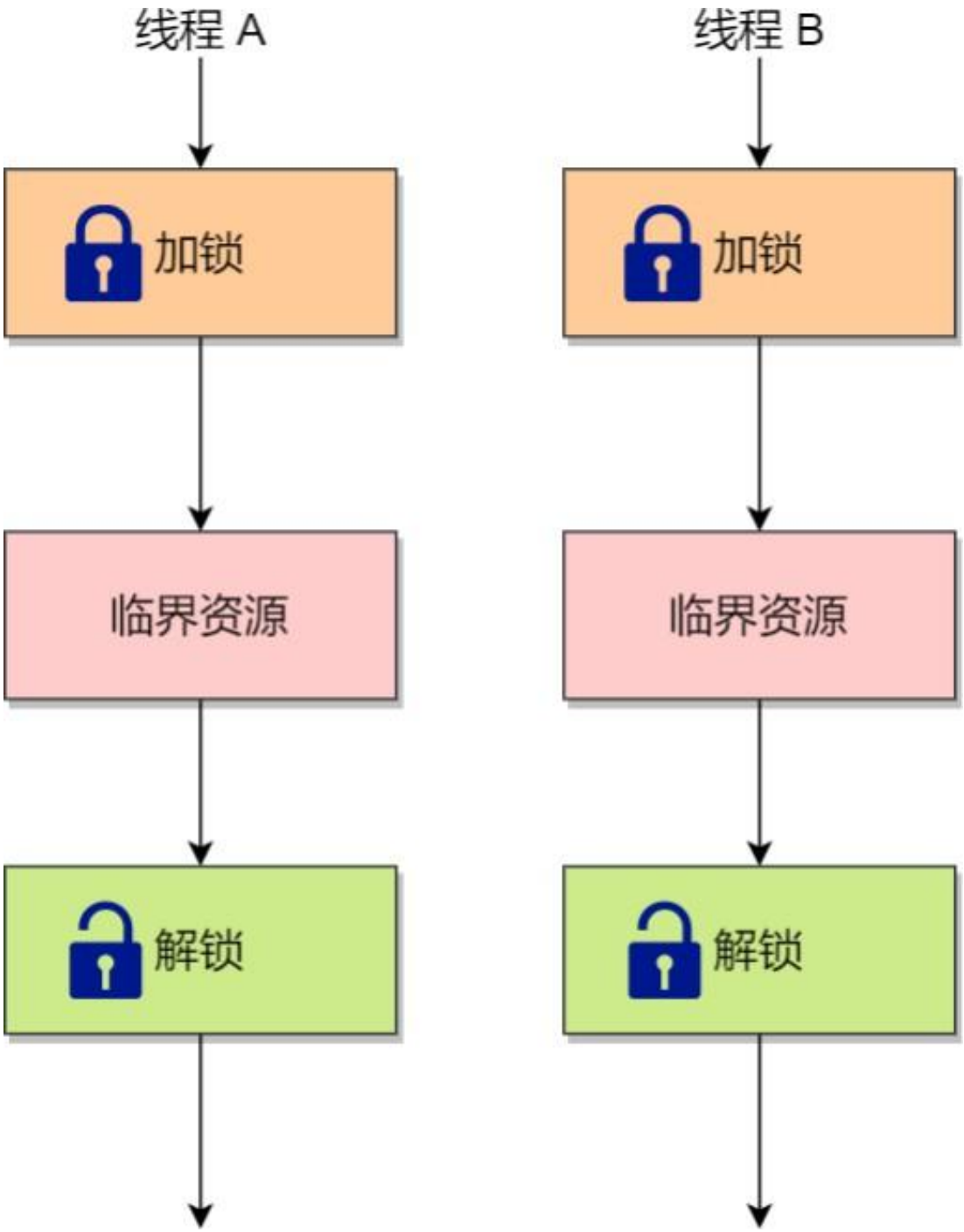
临界区同步的一些实现方式：

1、**锁**

使用加锁操作和解锁操作可以解决并发线程/进程的互斥问题。

任何想进入临界区的线程，必须先执行加锁操作。若加锁操作顺利通过，则线程可进入临界区；在完成对临界资源的访问后再执行解锁操作，以释放该临界资源。

加锁和解锁锁住的是什么呢？可以是`临界区对象`，也可以只是一个简单的`互斥量`，例如互斥量是`0`无锁，`1`表示加锁。



根据锁的实现不同，可以分为忙等待锁和 和 无忙等待锁 。

忙等待锁和 就是加锁失败的线程，会不断尝试获取锁，也被称为自旋锁，它会一直占用CPU。无

忙等待锁 就是加锁失败的线程，会进入阻塞状态，放弃CPU，等待被调度。

2、**信号量**

信号量是操作系统提供的一种协调共享资源访问的方法。

通常**信号量表示资源的数量**，对应的变量是一个整型（ sem ）变量。另

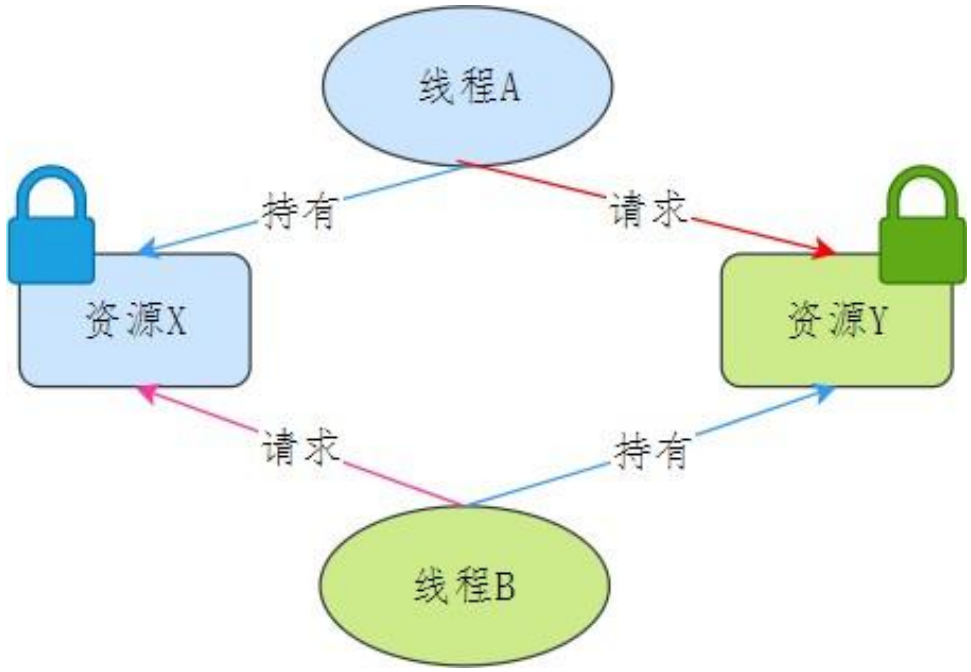
外，还有**两个原子操作的系统调用函数来控制信号量的**，分别是：

- *P* 操作：将 sem 减 1 ，相减后，如果 sem < 0 ，则进程/线程进入阻塞等待，否则继续，表明 P操作可能会阻塞；
- *V* 操作：将 sem 加 1 ，相加后，如果 sem <= 0 ，唤醒一个等待中的进程/线程，表明 V 操作不会阻塞；

P 操作是用在进入临界区之前，V 操作是用在离开临界区之后，这两个操作是必须成对出现的。

什么是死锁？

在两个或者多个并发线程中，如果每个线程持有某种资源，而又等待其它线程释放它或它们现在保持着的资源，在未改变这种状态之前都不能向前推进，称这一组线程产生了死锁。通俗的讲就是两个或多个线程无限期的阻塞、相互等待的一种状态。



死锁产生有哪些条件？

死锁产生需要**同时**满足四个条件：

- **互斥条件**：指线程对已经获取到的资源进行它性使用，即该资源同时只由一个线程占用。如果此时还有其它线程请求获取获取该资源，则请求者只能等待，直至占有资源的线程释放该资源。
- **请求并持有条件**：指一个 线程已经持有了至少一个资源，但又提出了新的资源请求，而新资源已被其它线程

占有，所以当前线程会被阻塞，但阻塞 的同时并不释放自己已经获取的资源。

- **不可剥夺条件**：指线程获取到的资源在自己使用完之前不能被其它线程抢占，只有在自己使用完毕后才由自己释放该资源。
- **环路等待条件**：指在发生死锁时，必然存在一个线程——资源的环形链，即线程集合 {T0，T1，T2,…… ，Tn} 中 T0 正在等待一 T1 占用的资源，T11正在等待 T2用的资源，…… Tn 在等待己被 T0占用的资源。

如何避免死锁呢？

产生死锁的有四个必要条件：互斥条件、持有并等待条件、不可剥夺条件、环路等待条件。避

免死锁，破坏其中的一个就可以。

消除互斥条件

这个是没法实现，因为很多资源就是只能被一个线程占用，例如锁。

消除请求并持有条件

消除这个条件的办法很简单，就是一个线程一次请求其所需要的所有资源。

消除不可剥夺条件

占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可剥夺这个条件就破坏掉了。

消除环路等待条件

可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号小的，再申请资源序号大的，这样线性化后就不存在环路了。

活锁和饥饿锁了解吗？

饥饿锁：

饥饿锁，这个饥饿指的是资源饥饿，某个线程一直等不到它所需要的资源，从而无法向前推进，就像一个人因为饥饿无法成长。

活锁：

在活锁状态下，处于活锁线程组里的线程状态可以改变，但是整个活锁组的线程无法推进。

活锁可以用两个人过一条很窄的小桥来比喻：为了让对方先过，两个人都往旁边让，但两个人总是让到同一边。这样，虽然两个人的状态一直在变化，但却都无法往前推进。

内存管理

什么是虚拟内存？

我们实际的物理内存主要是主存，但是物理主存空间有限，所以一般现代操作系统都会想办法把一部分内存块放到磁盘中，用到的时候再装入主存，但是对用户程序而言，是不需要注意实际的物理内存的，为什么呢？因为有虚拟内存的机制。

简单说，虚拟内存是操作系统提供的一种机制，将不同进程的虚拟地址和不同内存的物理地址映射起来。

每个进程都有自己独立的地址空间，再由操作系统映射到到实际的物理内存。

于是，这里就引出了两种地址的概念：

程序所使用的内存地址叫做**虚拟内存地址**（*Virtual Memory Address*）

实际存在硬件里面的空间地址叫**物理内存地址**（*Physical Memory Address*）。

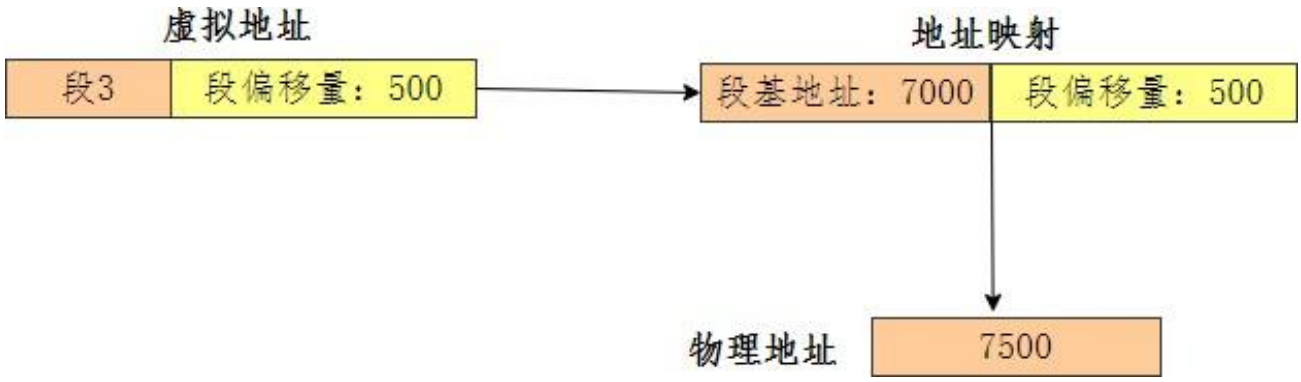
什么是内存分段？

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。

分段机制下的虚拟地址由两部分组成，**段号**和**段内偏移量**。

虚拟地址和物理地址通过段表映射，段表主要包括**段号**、 `段的界限` 。

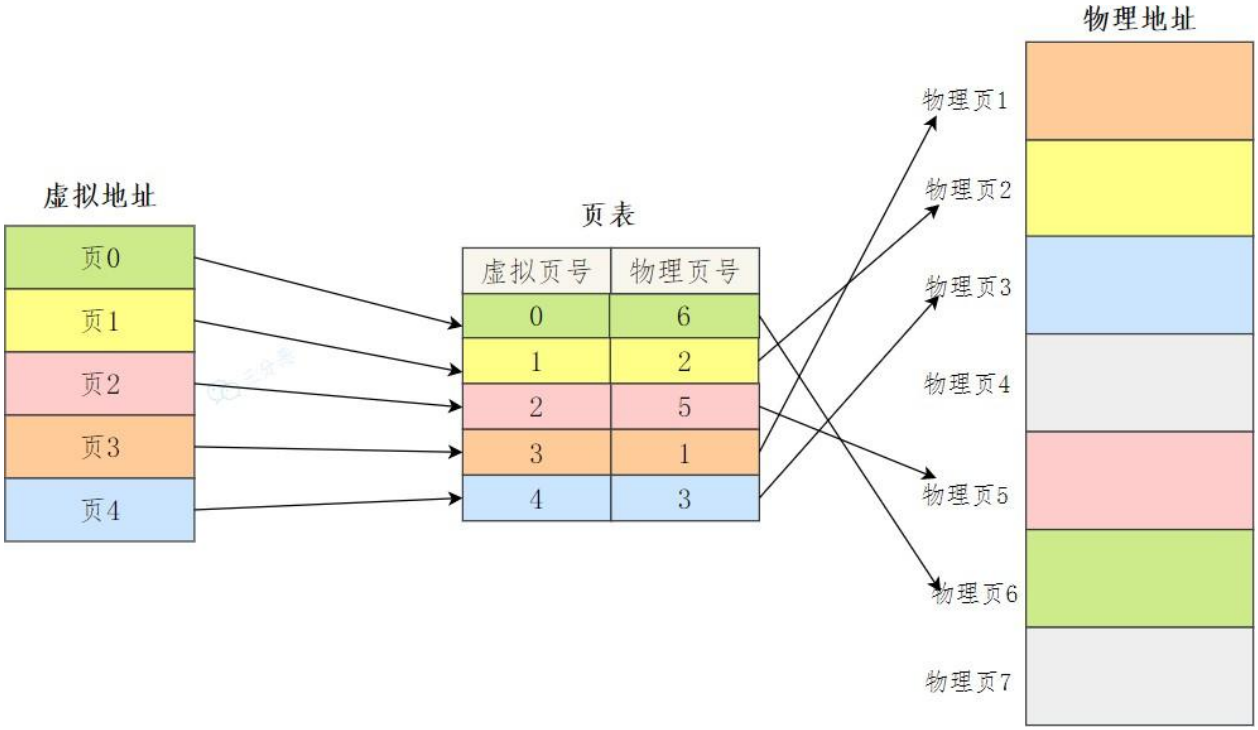
我们来看一个映射，虚拟地址：段3、段偏移量500 ----> 段基地址7000+段偏移量500----------> 物理地址：7500。



什么是内存分页？

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 。

访问分页系统中内存数据需要两次的内存访问 ：一次是从内存中访问页表，从中找到指定的物理页号，加上页内偏移得到实际物理地址，第二次就是根据第一次得到的物理地址访问内存取出数据。

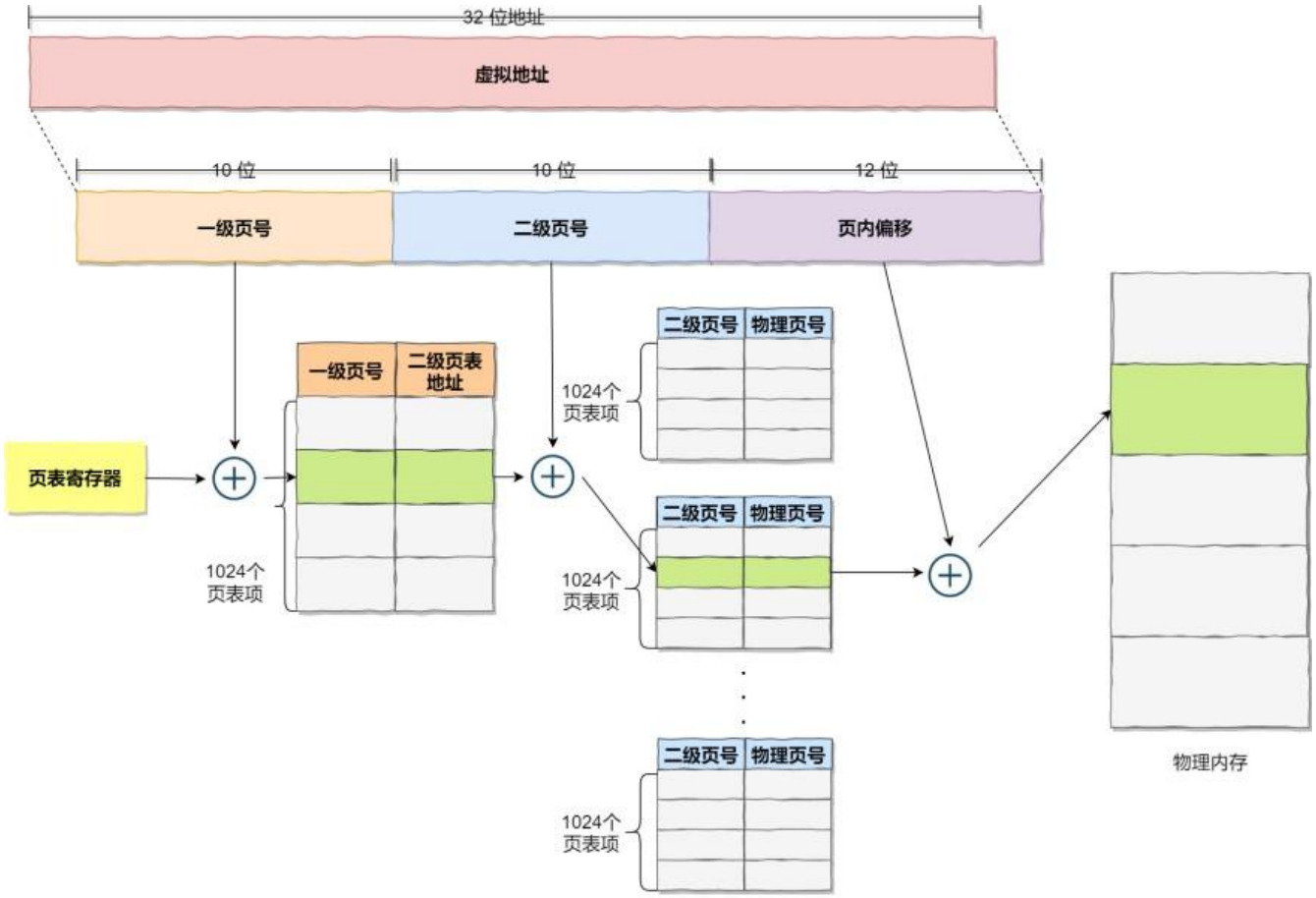


多级页表知道吗？

操作系统可能会有非常多进程，如果只是使用简单分页，可能导致的后果就是页表变得非常庞大。

所以，引入了多级页表的解决方案。

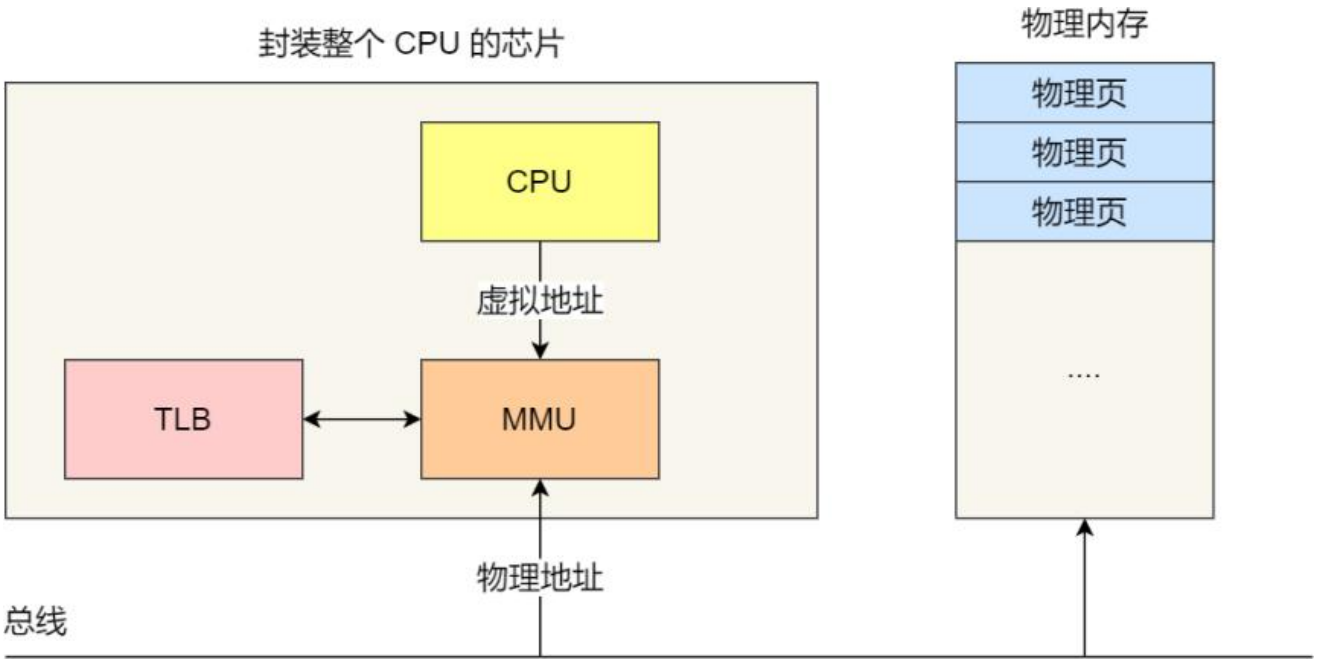
所谓的多级页表，就是把我们原来的单级页表再次分页，这里利用了局部性原理，除了顶级页表，其它级别的页表一来可以在需要的时候才被创建，二来内存紧张的时候还可以被置换到磁盘中。



什么是块表？

同样利用了局部性原理，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，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，这个 Cache 就是 TLB（*Translation Lookaside Buffer*），通常称为页表缓存、转址旁路缓存、快表等。



分页和分段有什么区别？

- 段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的 ；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的。
- 段的大小不固定，有它所完成的功能决定；页的大小固定，由系统决定
- 段向用户提供二维地址空间；页向用户提供的是一维地址空间
- 段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

什么是交换空间？

操作系统把物理内存(Physical RAM)分成一块一块的小内存，每一块内存被称为页(page)。当内存资源不足时，Linux把某些页的内容转移至磁盘上的一块空间上，以释放内存空间。磁盘上的那块空间叫做交换空间(swap space),而这一过程被称为交换(swapping)。物理内存和交换空间的总容量就是虚拟内存的可用容量。

用途：

- 物理内存不足时一些不常用的页可以被交换出去，腾给系统。
- 程序启动时很多内存页被用来初始化，之后便不再需要，可以交换出去。

页面置换算法有哪些？

在分页系统里，一个虚拟的页面可能在主存里，也可能在磁盘中，如果CPU发现虚拟地址对应的物理页不在主存里，就会产生一个缺页中断，然后从磁盘中把该页调入主存中。

如果内存里没有空间，就需要从主存里选择一个页面来置换。

常见的页面置换算法：

- **最佳页面置换算法（*OPT*）**

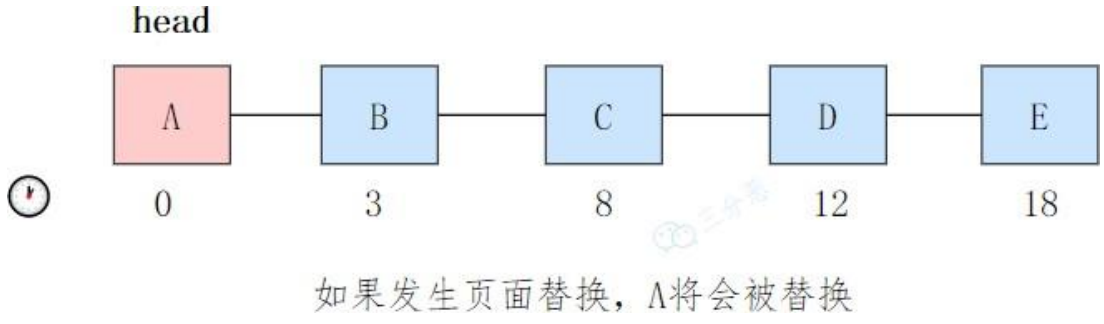
最佳页面置换算法是一个理想的算法，基本思路是，**置换在未来最长时间不访问的页面**。

所以，该算法实现需要计算内存中每个逻辑页面的下一次访问时间，然后比较，选择未来最长时间不访问的页面。但这个算法是无法实现的，因为当缺页中断发生时，操作系统无法知道各个页面下一次将在什么时候被访问。

- **先进先出置换算法**（FIFO）

既然我们无法预知页面在下一次访问前所需的等待时间，那可以**选择在内存驻留时间很长的页面进行中置换**，这个就是「先进先出置换」算法的思想。

FIFO的实现机制是使用链表将所有在内存的页面按照进入时间的早晚链接起来，然后每次置换链表头上的页面就行了，新加进来的页面则挂在链表的末端。

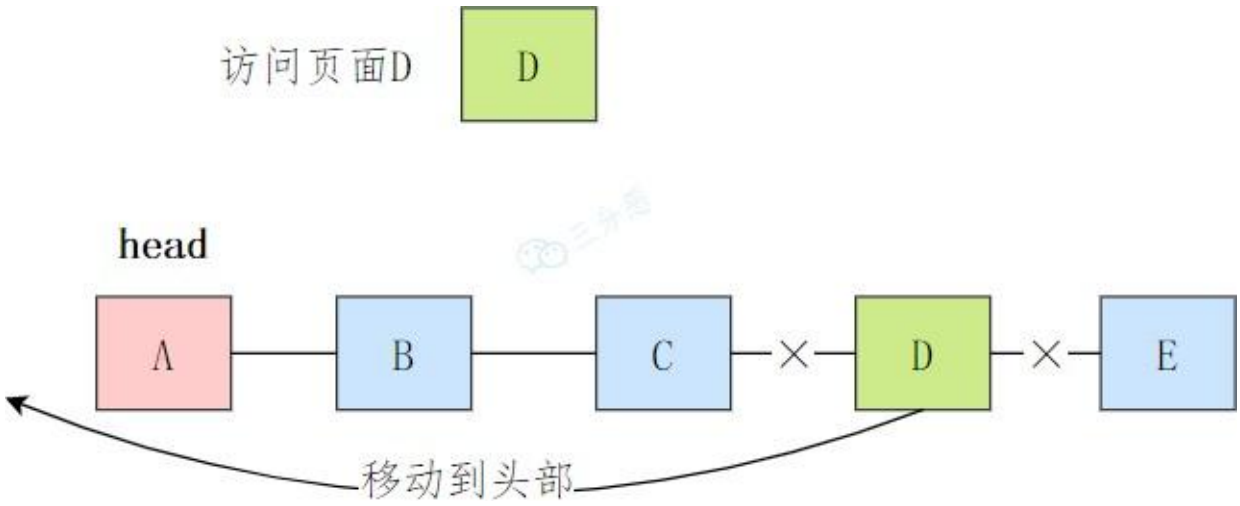


- **最近最久未使用的置换算法**（LRU）

最近最久未使用（*LRU*）的置换算法的基本思路是，发生缺页时，**选择最长时间没有被访问的页面进行置换**，也就是说，该算法假设已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。

这种算法近似最优置换算法，最优置换算法是通过「未来」的使用情况来推测要淘汰的页面，而 LRU 则是通过历史的使用情况来推测要淘汰的页面。

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。

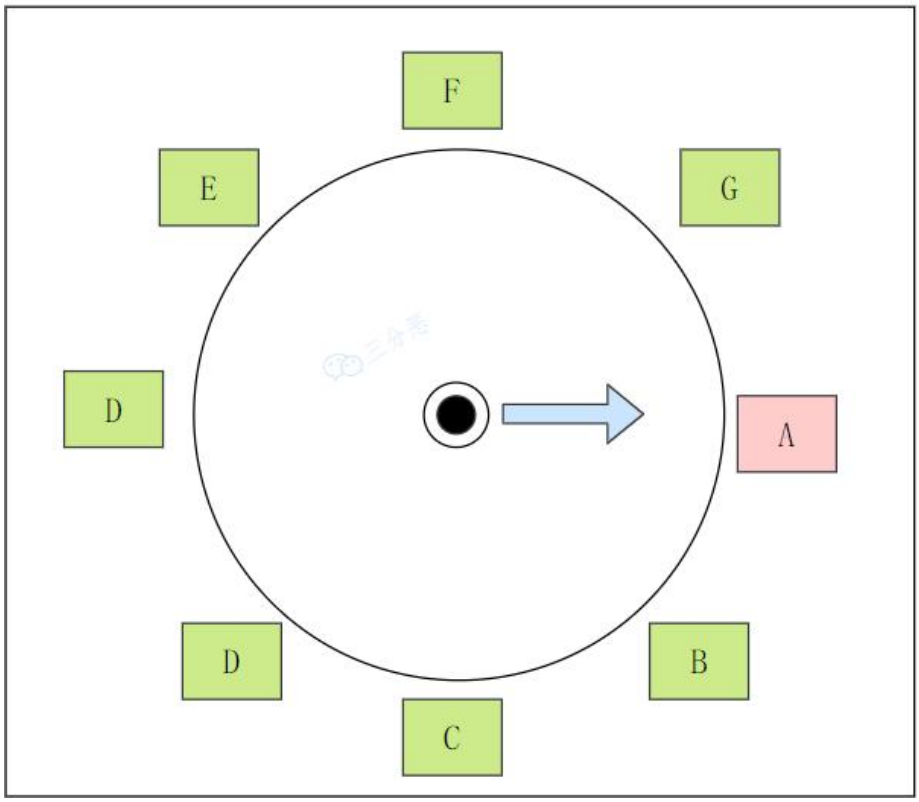


困难的是，在每次访问内存时都必须要更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常费时的操作。

所以，LRU 虽然看上去不错，但是由于开销比较大，实际应用中比较少使用。

- **时钟页面置换算法**

这个算法的思路是，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。



当发生缺页中断时，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；

- **最不常用置换算法**

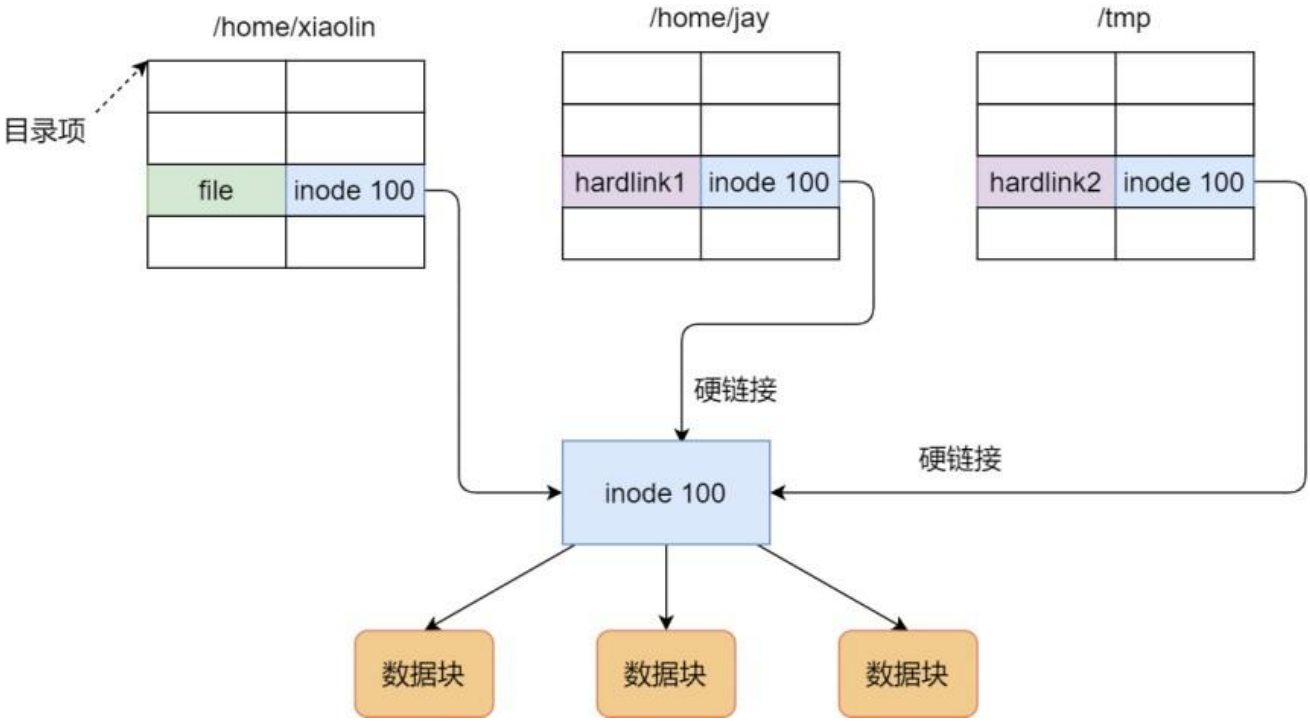
最不常用算法（LFU），**当发生缺页中断时，选择访问次数最少的那个页面，将其置换**。

它的实现方式是，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。

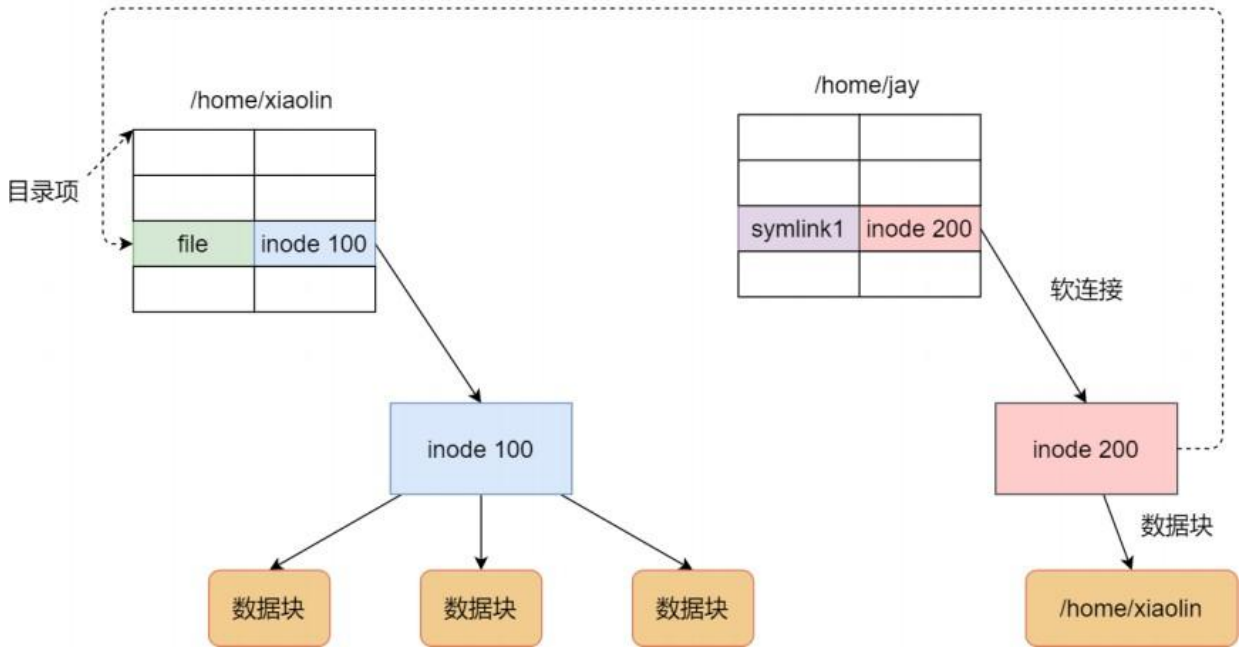
文件

硬链接和软链接有什么区别？

- 硬链接就是在目录下创建一个条目，记录着文件名与 inode 编号，这个 inode 就是源文件的 inode。删除任意一个条目，文件还是存在，只要引用数量不为 0。但是硬链接有限制，它不能跨越文件系统，也不能对目录进行链接。



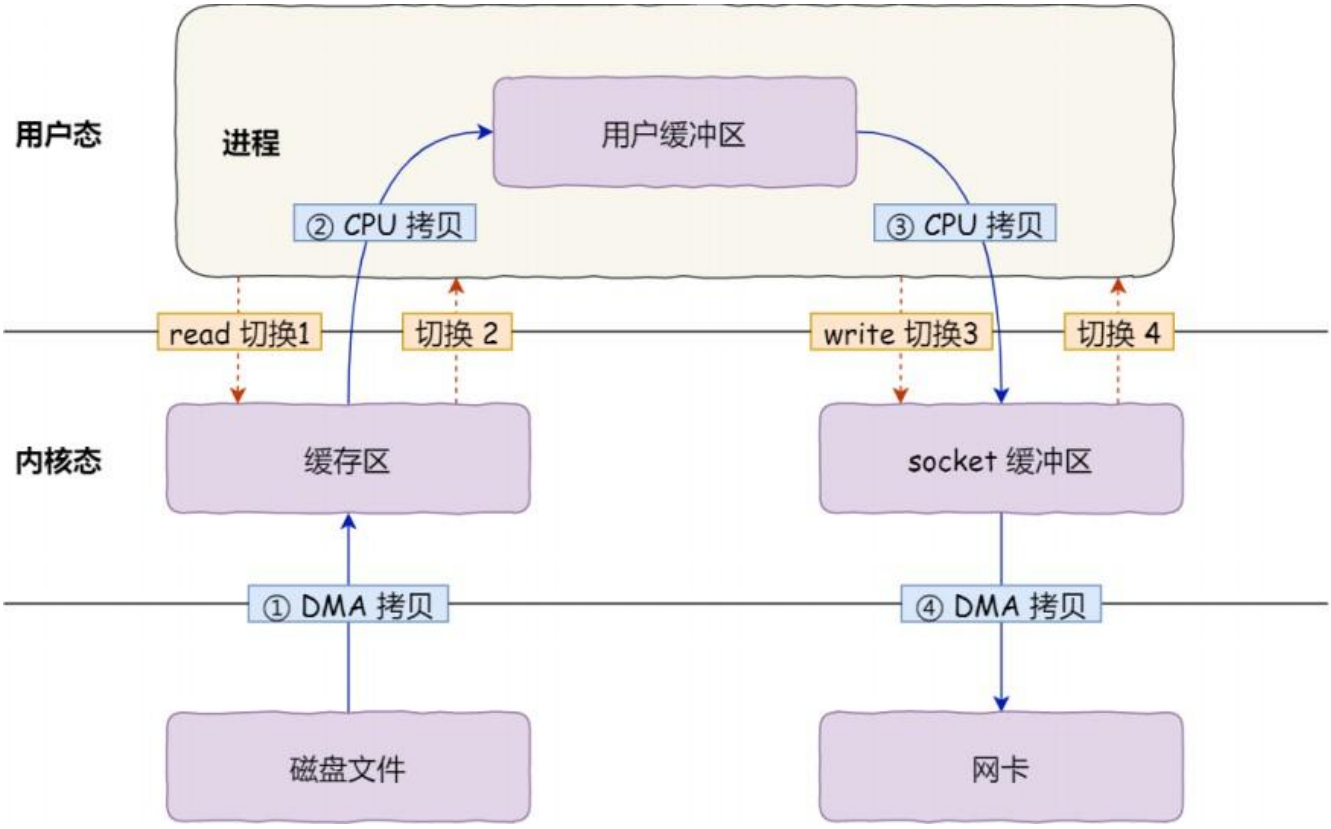
- 软链接相当于重新创建一个文件，这个文件有**独立的** inode，但是这个**文件的内容是另外一个文件的路径**，所以访问软链接的时候，实际上相当于访问到了另外一个文件，所以**软链接是可以跨文件系统的**，甚至**目标文件被删除了，链接文件还是在的，只不过打不开指向的文件了而已。**



IO

零拷贝了解吗？

假如需要文件传输，使用传统I/O，数据读取和写入是用户空间到内核空间来回赋值，而内核空间的数据是通过操作系统的I/O接口从磁盘读取或者写入，这期间发生了多次用户态和内核态的上下文切换，以及多次数据拷贝。

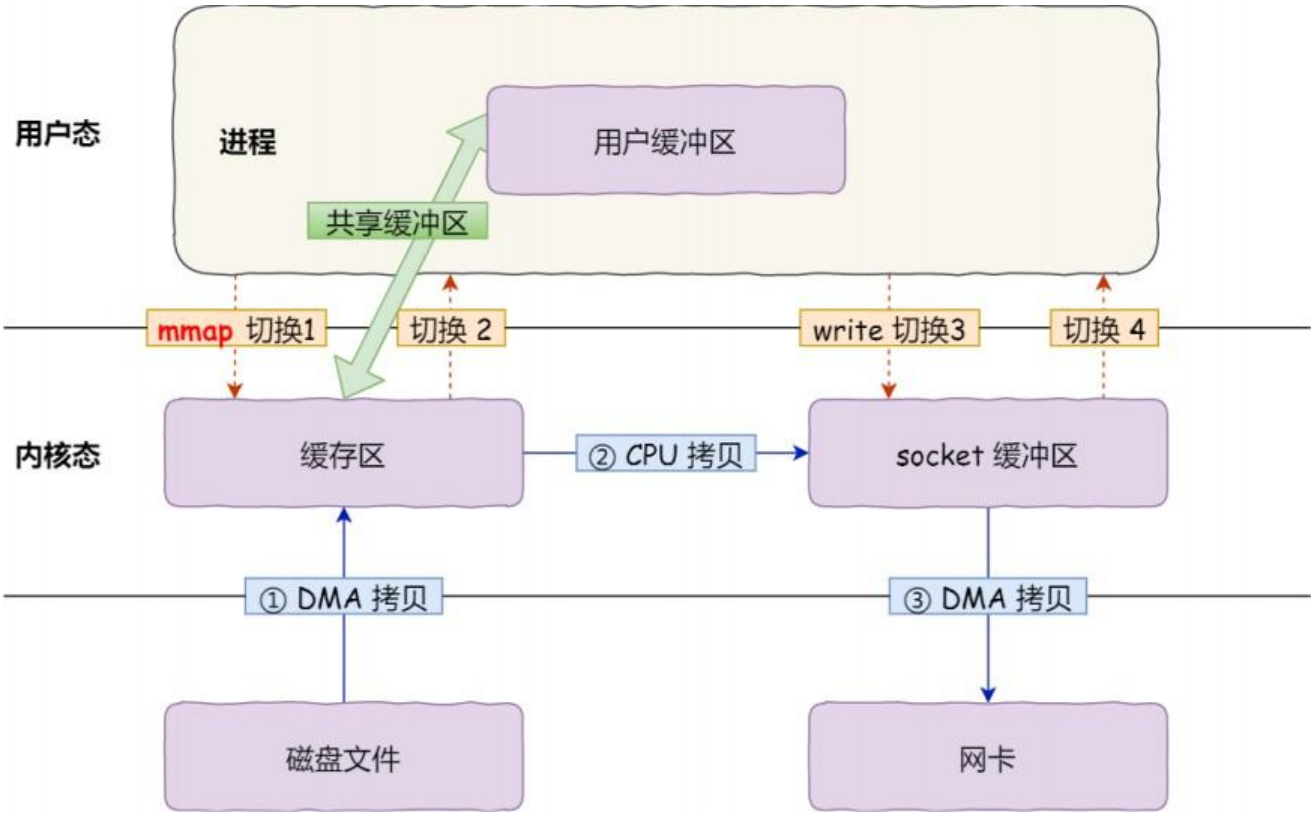


为了提升I/O性能，就需要**减少用户态与内核态的上下文切换**和**内存拷贝的次数**。

这就用到了我们零拷贝的技术，零拷贝技术实现主要有两种：

- mmap + write

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。

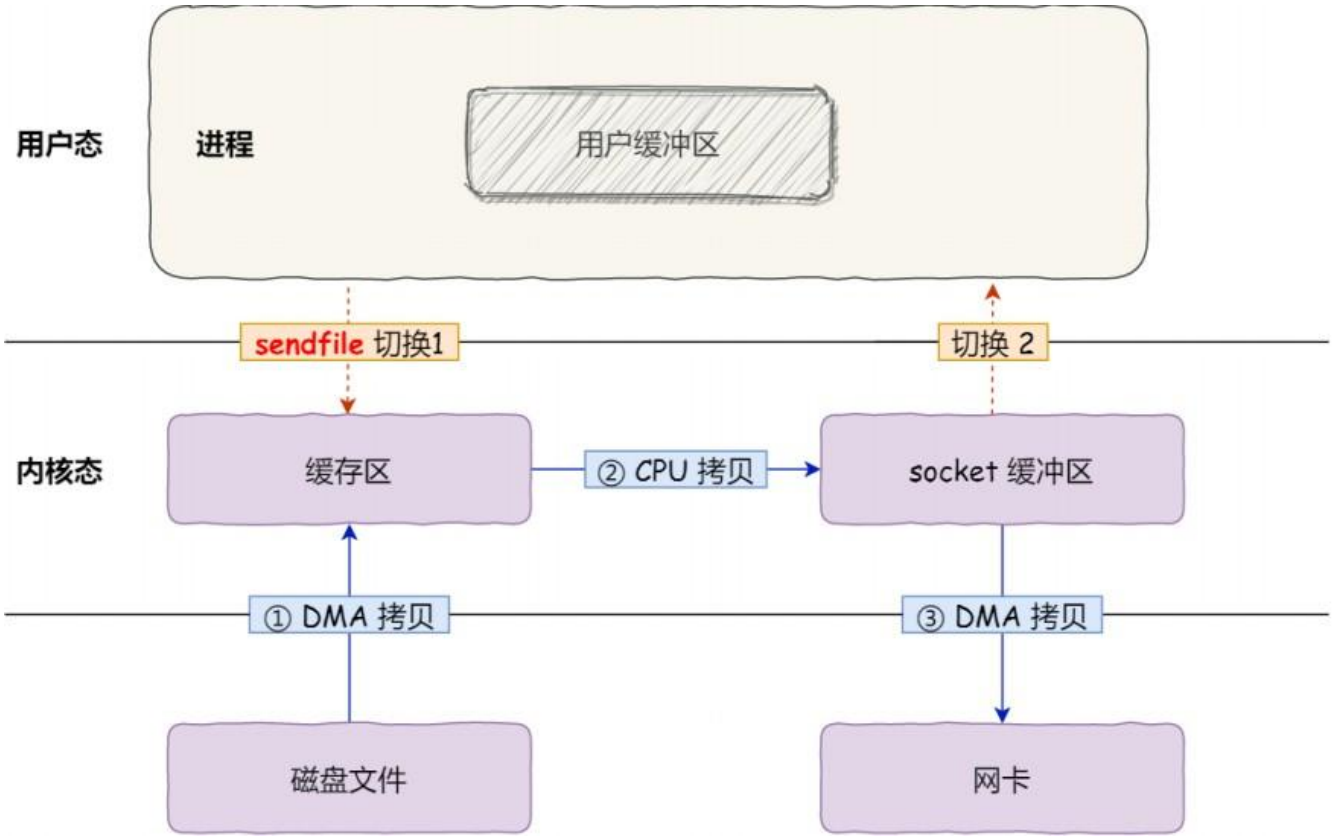


- sendfile

在 Linux 内核版本 2.1 中，提供了一个专门发送文件的系统调用函数 sendfile() 。

首先，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。

其次，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。



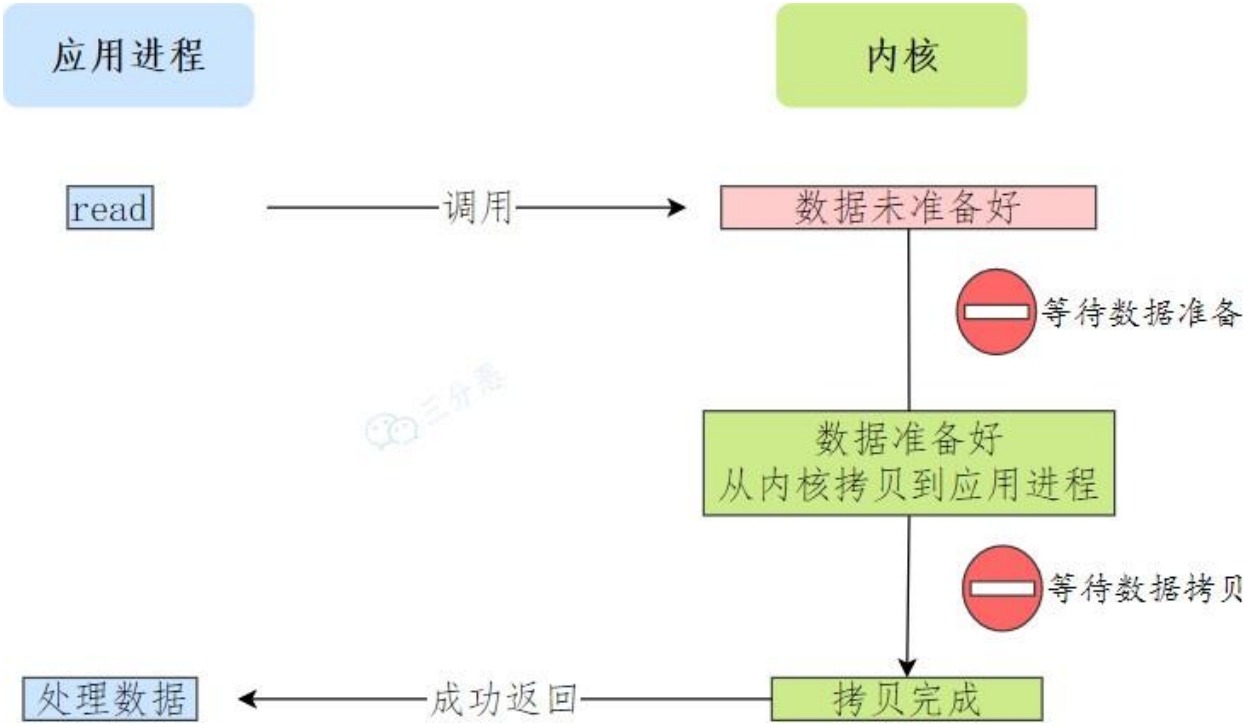
很多开源项目如Kafka、RocketMQ都采用了零拷贝技术来提升IO效率。

聊聊阻塞与非阻塞 I/O 、 同步与异步 I/O?

- **阻塞**I/O

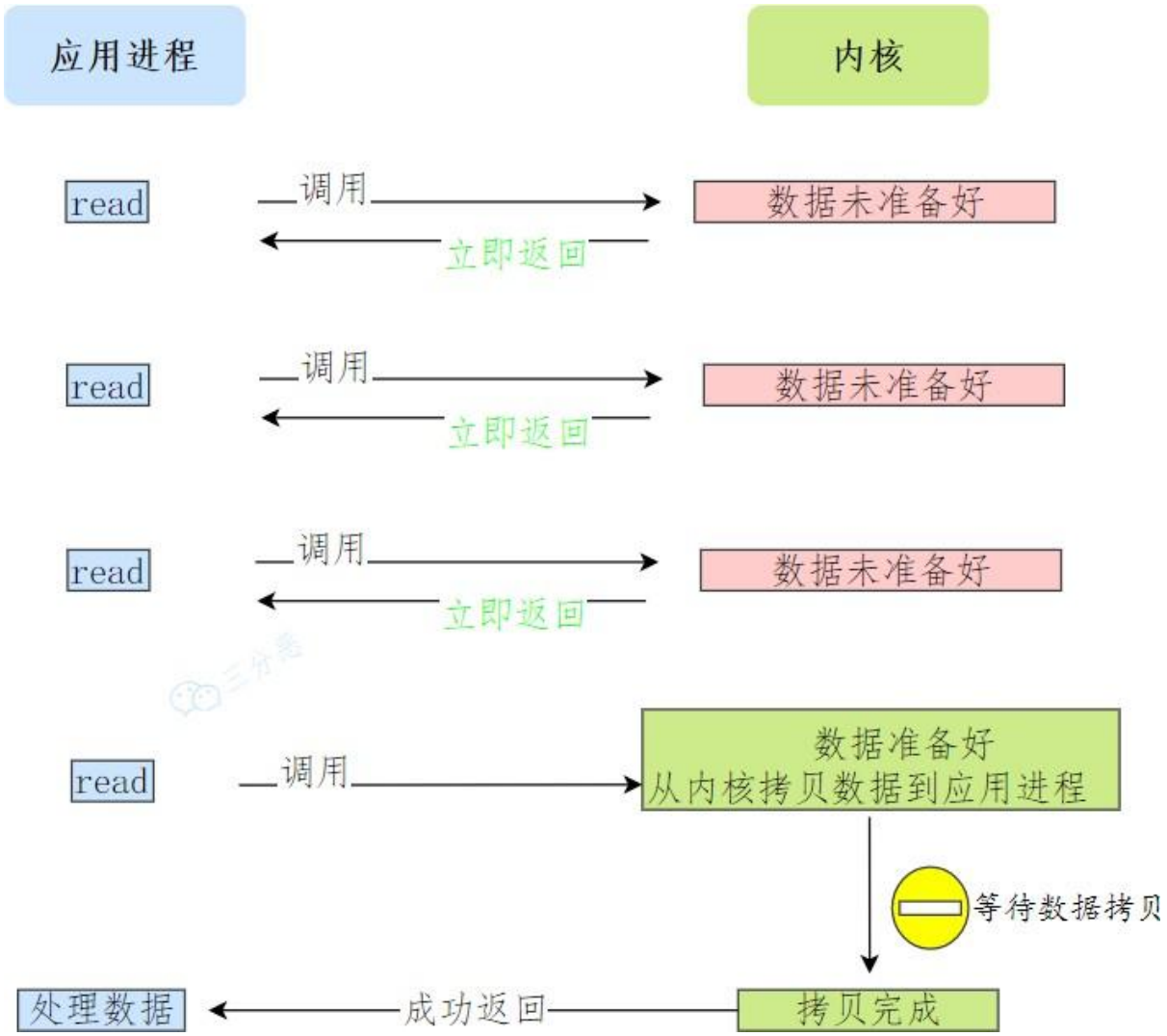
先来看看**阻塞** I/O，当用户程序执行read，线程会被阻塞，一直等到内核数据准备好，并把数据从内核缓冲区拷贝到应用程序的缓冲区中，当拷贝过程完成， read 才会返回。

注意，**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

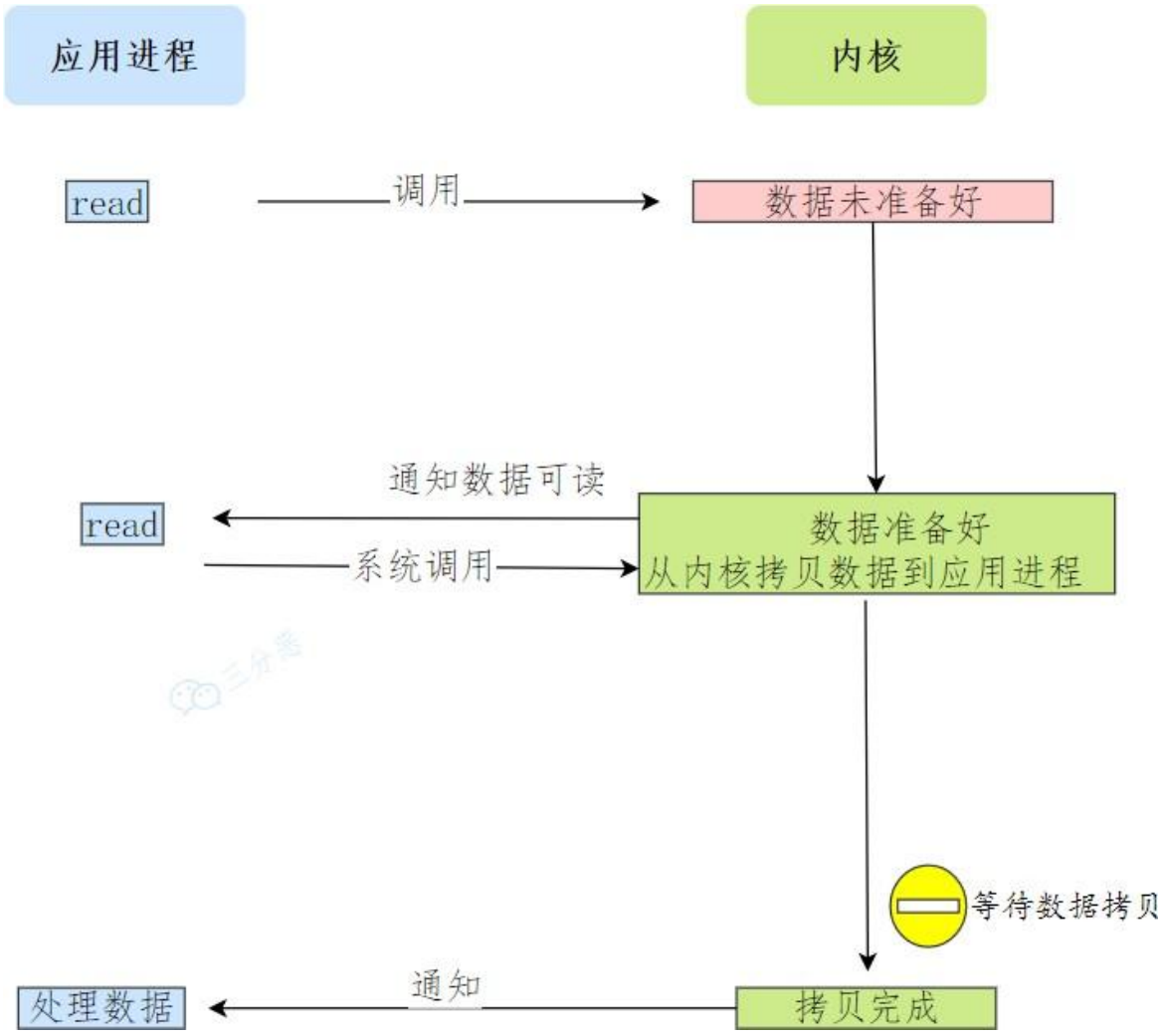
非阻塞的 read 请求在数据未准备好的情况下立即返回，可以继续往下执行，此时应用程序不断轮询内核，直到数据准备好，内核将数据拷贝到应用程序缓冲区， read 调用才可以获取到结果。



- **基于非阻塞的I/O多路复用**

我们上面的非阻塞I/O有一个问题，什么问题呢？应用程序要一直轮询，这个过程没法干其它事情，所以引入了I/O **多路复用**技术。

当内核数据准备好时，以事件通知应用程序进行操作。

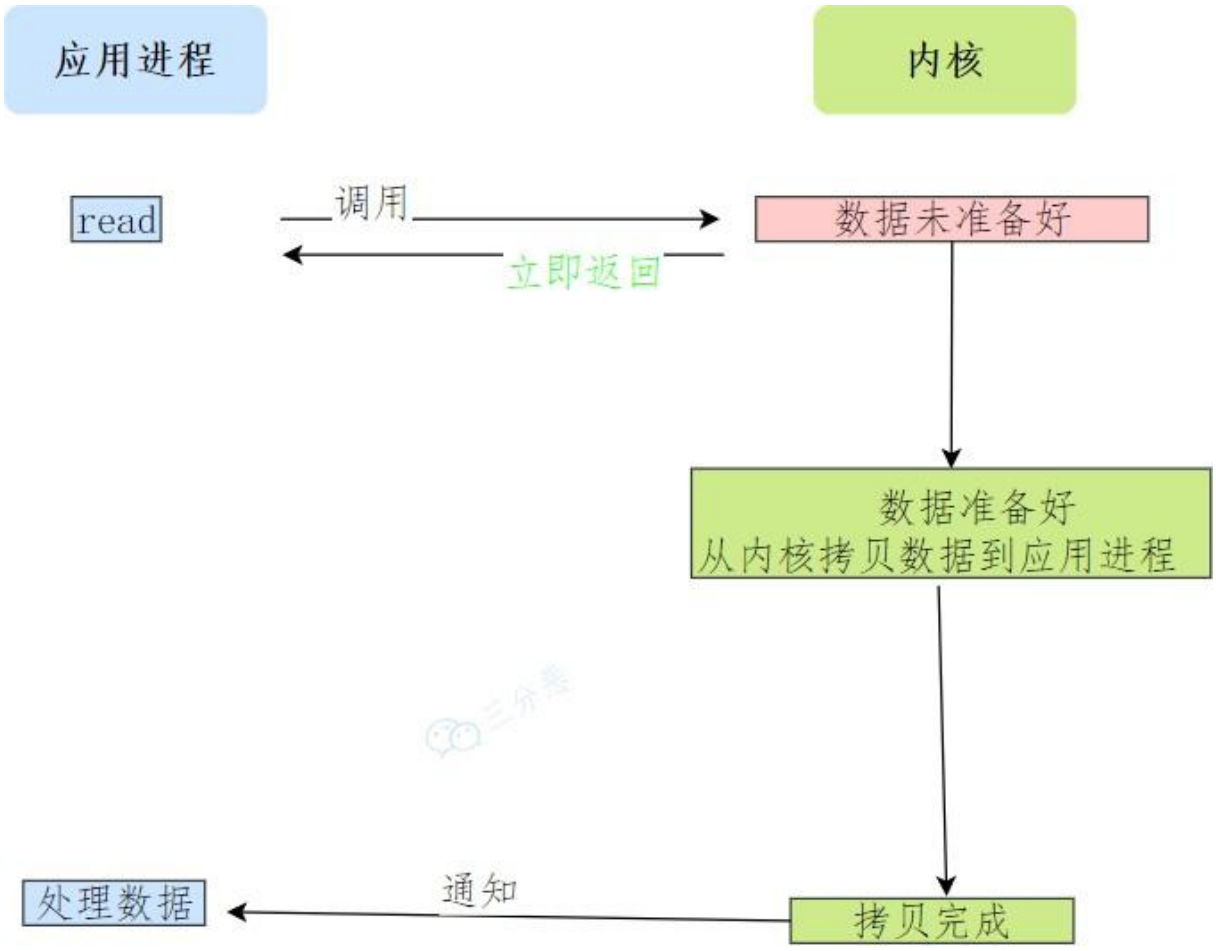


注意：无论是阻塞 I/O、还是非阻塞 I/O、非阻塞I/O多路复用，都是同步调用。因为它们在read调用时，内核将数据从内核空间拷贝到应用程序空间，过程都是需要等待的，也就是说这个过程是**同步**的，如果内核实现的拷贝效率不高，read调用就会在这个同步过程中等待比较长的时间。

- **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。

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。



> 拿例子理解几种I/O模型

老三关注了很多UP主，有些UP主是老鸽子，到了更新的时间：阻

塞I/O就是，老三不干别的，就干等着，盯着UP的更新。

非阻塞I/O就是，老三发现UP没更，就去喝个茶什么的，过一会儿来盯一次，一直等到UP更新。

基于非阻塞的 I/O 多路复用好比，老三发现UP没更，就去干别的，过了一会儿B站推送消息了，老三一看，有很多条，就去翻动态，看看等的UP是不是更新了。

异步I/O就是，老三说UP你该更了，UP赶紧爆肝把视频做出来，然后把视频亲自呈到老三面前，这个过程不用等待。

详细讲一讲I/O多路复用？

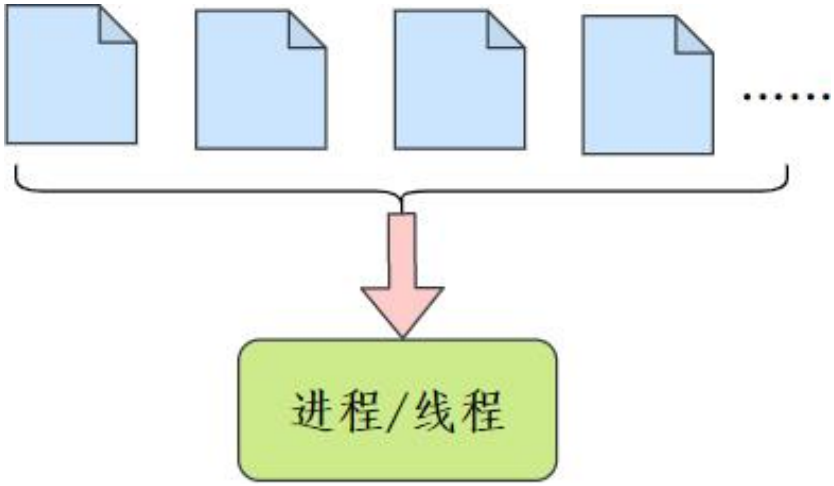
> 我们先了解什么是I/O多路复用？

我们在传统的I/O模型中，如果服务端需要支持多个客户端，我们可能要为每个客户端分配一个进程/线程。

不管是基于重一点的进程模型，还是轻一点的线程模型，假如连接多了，操作系统是扛不住的。

所以就引入了I/O**多路复用** 技术。

简单说，就是一个进程/线程维护多个Socket，这个多路复用就是多个连接复用一个进程/线程。



我们来看看I/O多路复用三种实现机制：

- select

select 实现多路复用的方式是：

将已连接的 Socket 都放到一个**文件描述符集合**fd_set，然后调用 select 函数将fd_set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fd_set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fd_set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，再对其处理。

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。

> select机制的缺点：

(1) 每次调用select，都需要把fd_set集合从用户态拷贝到内核态，如果fd_set集合很大时，那这个开销也很大，比如百万连接却只有少数活跃连接时这样做就太没有效率。

(2) 每次调用select都需要在内核遍历传递进来的所有fd_set，如果fd_set集合很大时，那这个开销也很大。

(3) 为了减少数据拷贝带来的性能损坏，内核对被监控的fd_set集合大小做了限制，一般为1024，如果想要修改会比较麻烦，可能还需要编译内核。

(4) 每次调用select之前都需要遍历设置监听集合，重复工作。

- poll

poll 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。

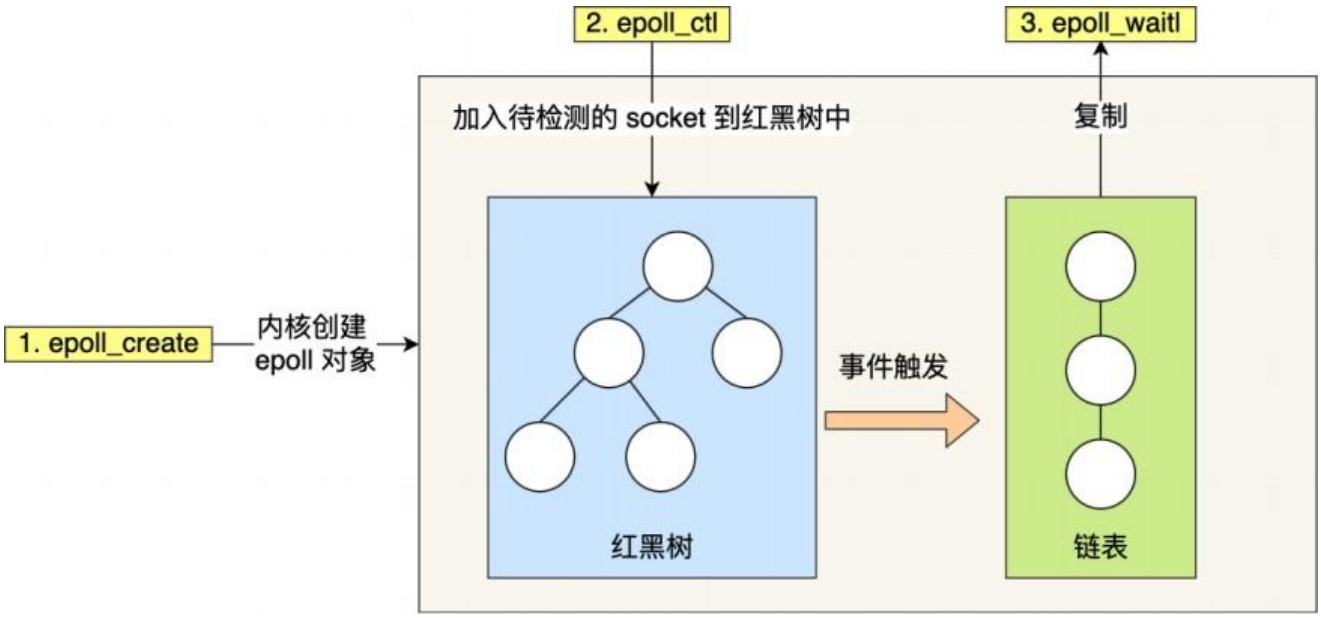
但是 poll 和 select 并没有太大的本质区别，都是使用线性结构存储进程关注的Socket集合，因此都需要遍历文件描述符集合来找到可读或可写的Socke，时间复杂度为O(n)，而且也需要在用户态与内核态之间拷贝文件描述符集合，这种方式随着并发数上来，性能的损耗会呈指数级增长。

- epoll

epoll 通过两个方面，很好解决了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字**，把需要监控的 socket 通过epoll_ctl() 函数加入内核中的红黑树里，红黑树是个高效的数据结构，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**。