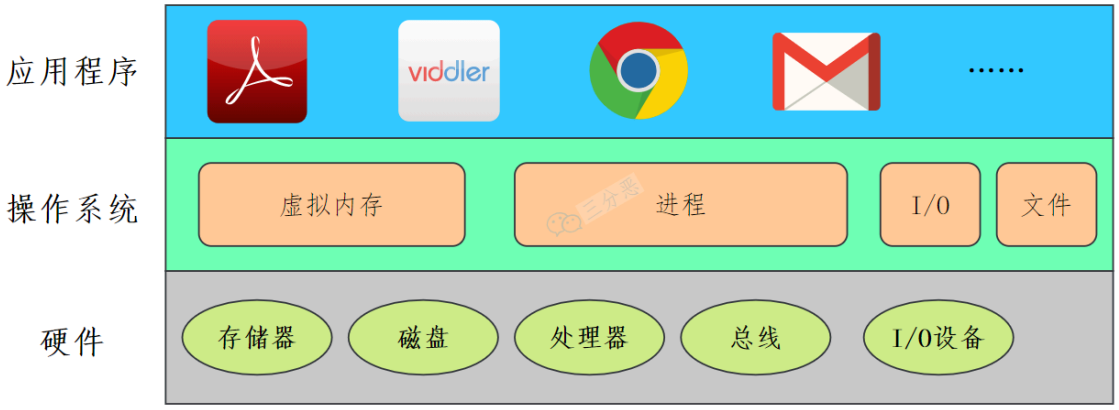
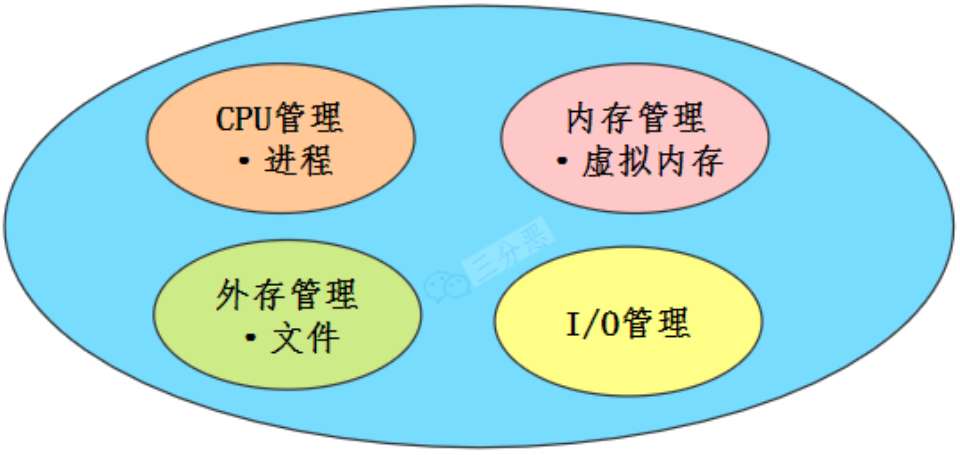
引论

什么是操作系统?

操作系统是计算机系统中管理硬件和软件资源的中间层系统，屏蔽了硬件的复杂性，并且为用户提供了便捷的交互方式



操作系统主要有哪些功能



操作系统主要功能

- 负责创建和终止进程
- 负责为进程分配资源
- 提供创建、删除、读写文件的功能，并组织文件的存储结构，比如说目录
- 通过设备驱动程序控制和管理计算机的硬件设为，如鼠标、键盘、打印机等

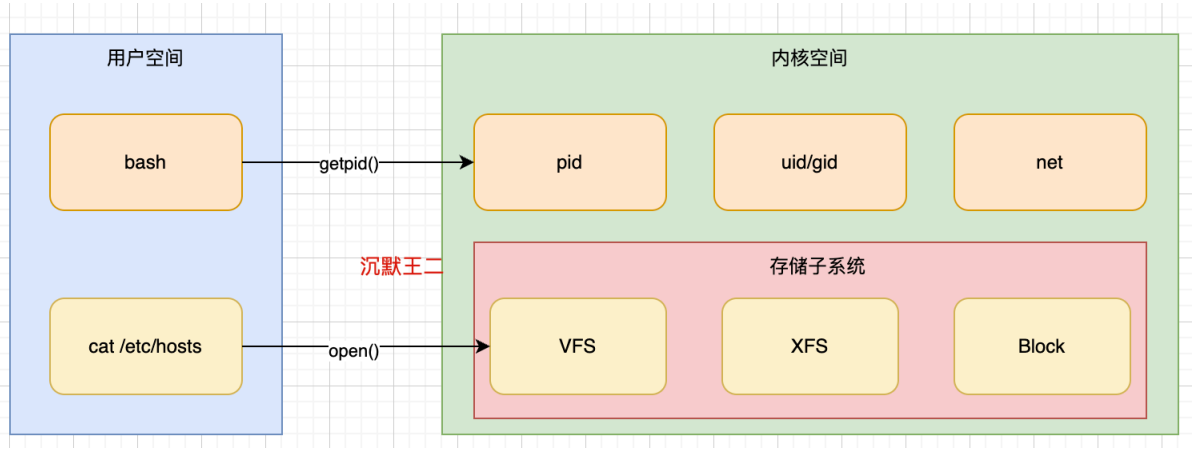
操作系统结构

什么是内核

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

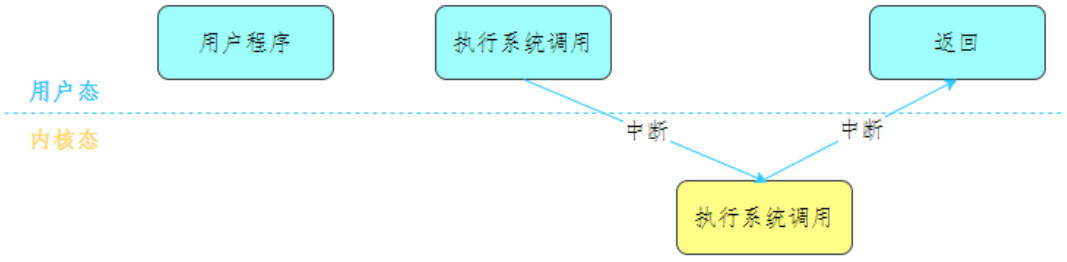
在计算机系统中，内存可分为两大区域：**内核空间**和**用户空间**。这种划分主要用户保护系统的稳定性和安全性。



- 内核空间
  - 操作系统内核代码及其运行时数据结构所在的内存区域，拥有对系统所有资源的完全访问权限，如进程管理、内存管理、文件系统、网络堆栈等。
- 用户空间
  - 是操作系统为应用程序（如用户运行进程）分配的内存区域，用户空间中的进程不能直接访问硬件或内核数据结构

用户态和内核态是如何切换的

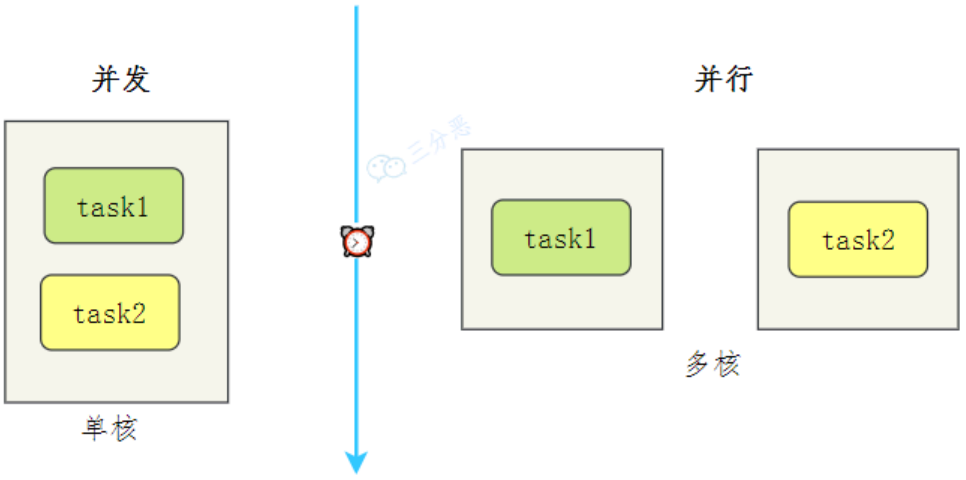
当应用程序执行系统调用时，CPU 将从用户态切换到内核态，进入内核空间执行相应的内核代码，然后再切换回用户态。



系统调用是应用程序请求操作系统内核提供服务的接口，如文件操作（如 open、read、write）、进程控制（如 fork、exec）、内存管理（如 mmap）等。

进程和线程

并行和并发有什么区别



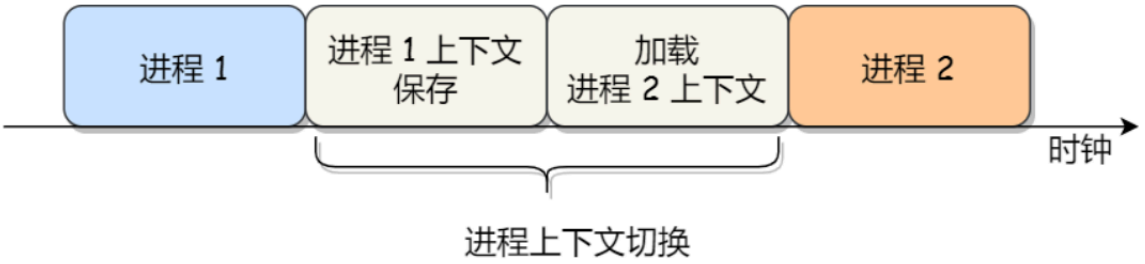
并发就是在一段时间内，多个任务都会被处理；**但在某一时刻，只有一个任务在执行。**

- 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执行。

什么是进程上下文切换



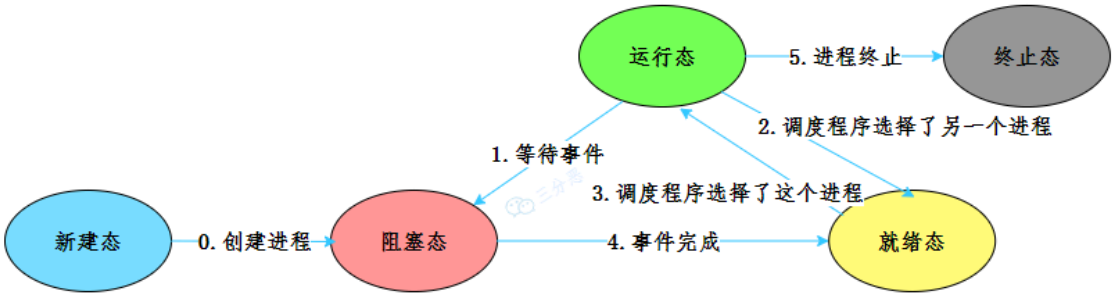
上下文切换是操作系统在多任务处理环境中，将CPU从一个进程切换到另一个进程的过程。通过让多个进程共享CPU资源，使系统能够并发执行多个任务。

进程上下文切换包含以下几个步骤：

- 保存当前进程的上下文
  - 操作系统保存当前进程的CPU寄存器，程序状态等关键信息
- 选择下一个进程
  - 调度程序选择下一个要执行的进程

- 恢复上一个进程的上下文
- 切换到下一个进程

进程有哪些状态



- 新建状态（New）：进程正在被创建时的状态
- 终止状态（Exit）：进程正在从系统中消失时的状态
- 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；

什么是僵尸进程

僵尸进程是已完成且处于终止状态，但在进程表中仍然存在的进程。

为什么会有僵尸进程

- 父进程没有调用 `wait()`
  - 如果父进程没有调用 `wait()` 或 `waitpid()`，操作系统不会从进程表中移除子进程的信息，导致子进程变成**僵尸进程**。
- 父进程还在运行，但没有管理子进程
  - 父进程可能**忘记回收子进程**，但它自己仍然在运行，导致子进程进入僵尸状态。

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。

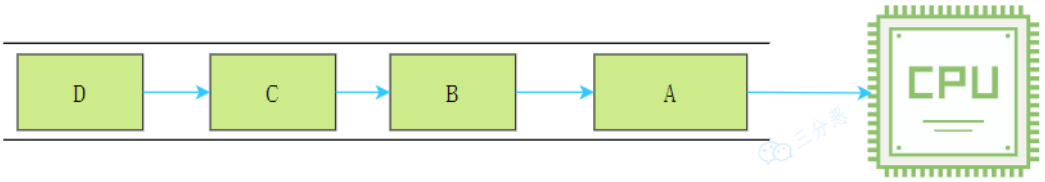
孤儿进程将被 init 进程 (进程 ID 为 1 的进程) 所收养，并由 init 进程对它们完成状态收集工作。因为孤儿进程会被 init 进程收养，所以孤儿进程不会对系统造成危害。

进程有哪些调度算法

进程调度是操作系统中的核心功能，负责决定哪些进程在何时使用CPU。

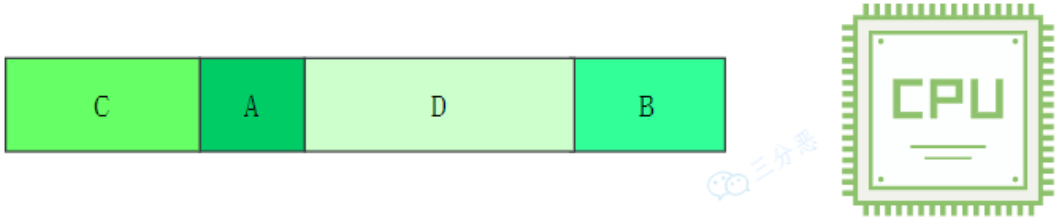
1. 先来先服务

进程按照请求CPU的顺序进行调度。易于实现，但可能会导致较短的进程等待较长进程执行完成，从而产生“饥饿”现象



2. 短作业优先

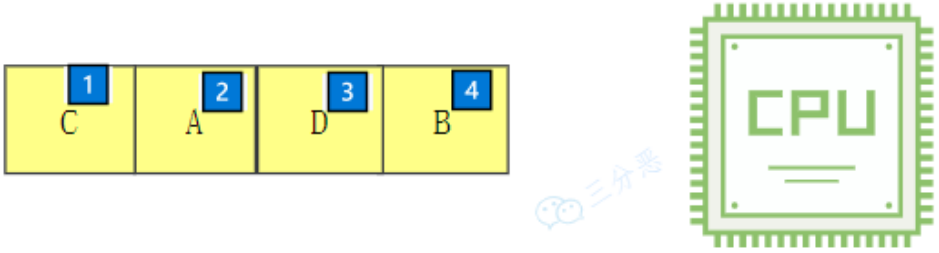
选择预计运行时间最短的进程优先执行。这种方式可以减少平均等待时间和响应时间，但缺点是很难准确预知进程的执行时间，并且可能因为一直有短作业在执行导致长作业持续被推迟。



3. 优先级调度

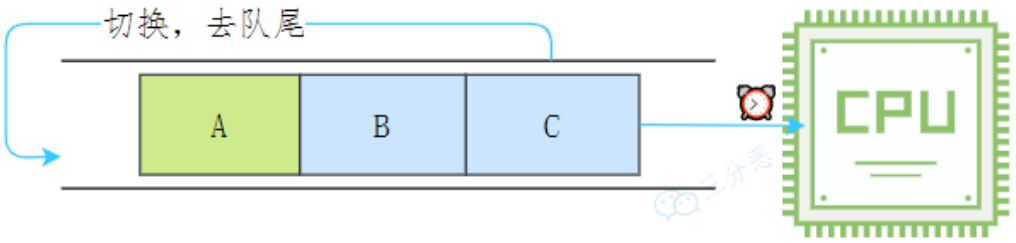
每个进程被分配一个优先级。CPU首先分配给优先级最高的进程。

优先级调度可以是非抢占式的或抢占式的。在非抢占式优先级调度中，进程一旦开始执行将一直运行直到完成；在抢占式优先级调度中，更高优先级的进程可以中断正在执行的低优先级进程。



4. 时间片轮转

时间片轮转调度为每个进程分配一个固定的时间段，称为时间片，进程可以在这个时间片内运行。如果进程在时间片结束时还没有完成，它将被放回队列的末尾。时间片轮转是公平的调度方式，可以保证所有进程得到公平的 CPU 时间，适用于共享系统。



5. 最短剩余时间优先

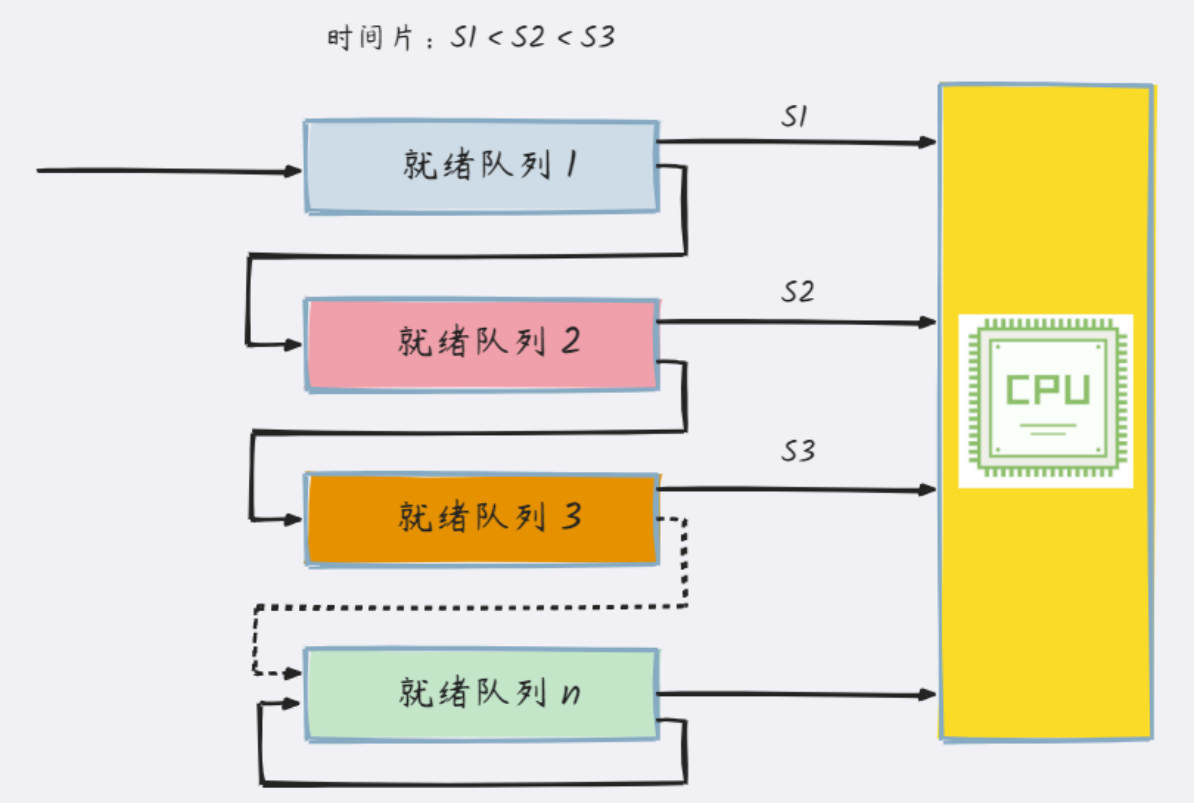
这是短作业优先的一种改进形式，它是抢占式的。即如果一个新进程的预计执行时间比当前运行进程的剩余时间短，调度器将暂停当前的进程，并切换到新进程。这种方法也可以最小化平均等待时间，但同样面临预测执行时间的困难。

6. 多级反馈队列

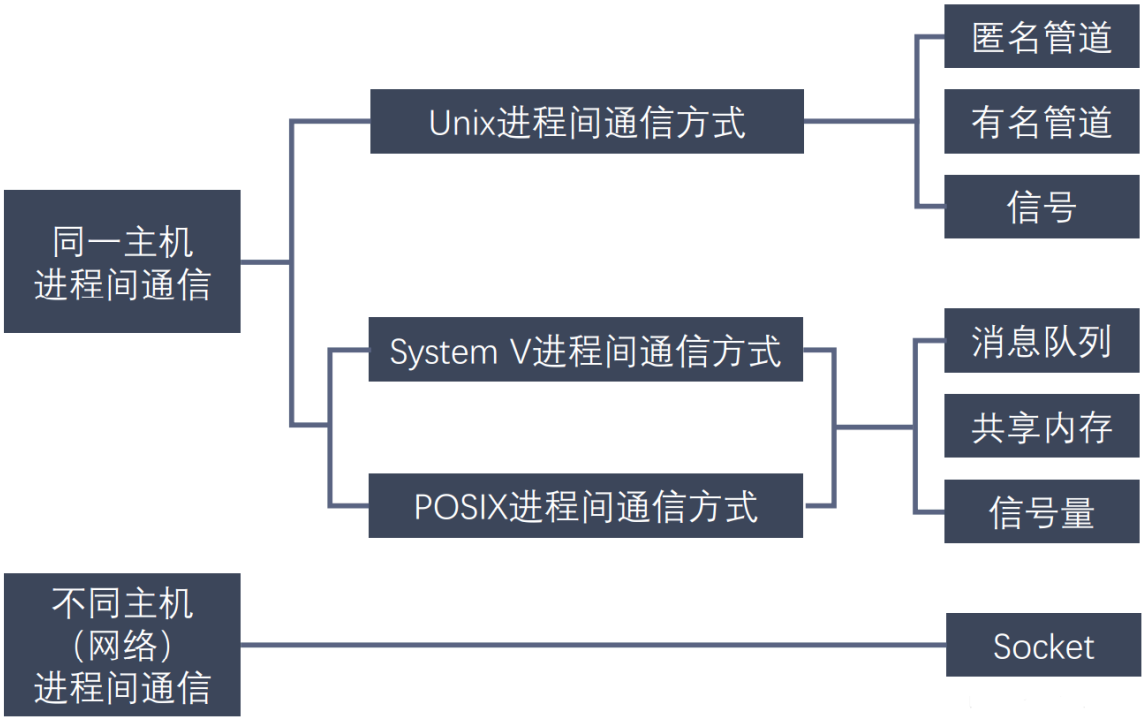
一个进程需要执行100 个时间片，如果采用时间片轮转调度算法，那么需要交互 100 次。

多级队列就是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列的时间片大小不同，比如 2,4,6,8……。进程在第一个队列没执行完，就会被移到下一个队列。

这种方式下，之前的进程只需要交换 7 次就可以了。每个队列优先权不一样，最上面的队列优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。



进程间通信有哪些方式



1. 管道

- 管道可以理解成不同进程之间的传话筒，一方发声，一方接收
- **进程间的管道就是内核中的一串缓存**，从管道的一端写入数据，另一端读取。数据只能单向流动，遵循先进先出（FIFO）的原则。
- 缺点：管道的效率低，不适合进程间频繁地交换数据。

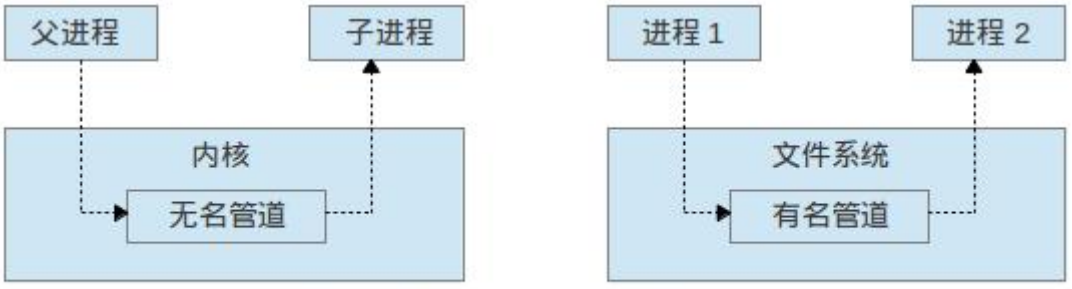


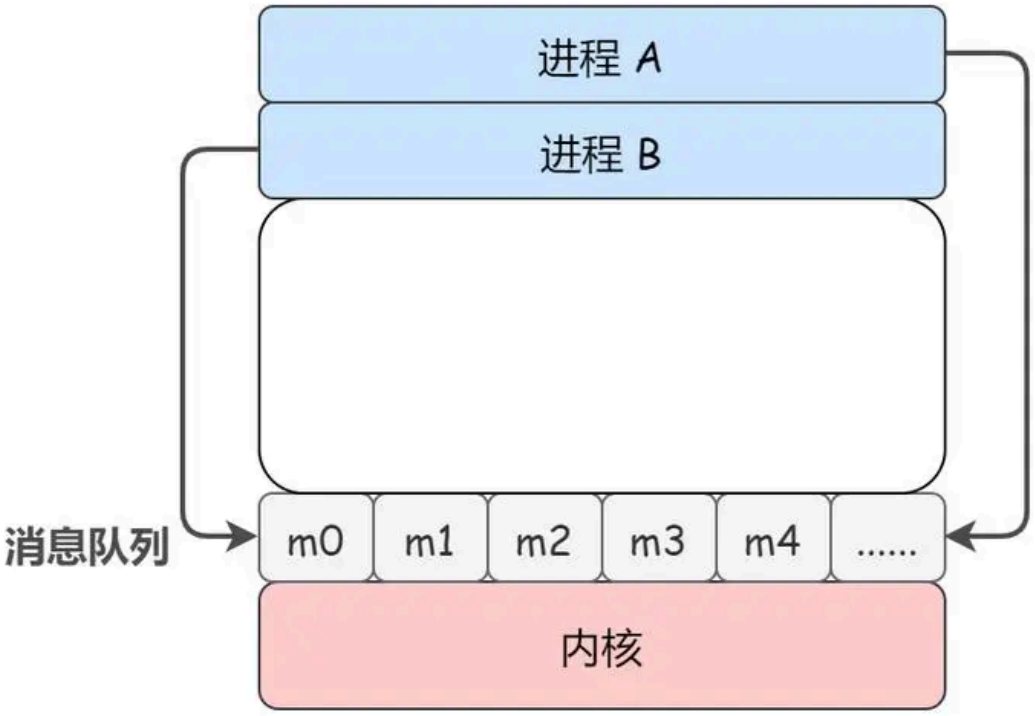
图 1　无名管道（左）和有名管道（右）

2. 信号

- 用于通知接收进程某件事情发生了，是一种较为简单的通信方式，主要用于处理异步事件。
- 常用信号
  - SIGHUP：当我们退出终端（Terminal）时，由该终端启动的所有进程都会接收到这个信号，默认动作为终止进程。
  - SIGINT：程序终止（interrupt）信号。按 `Ctrl+C` 时发出，大家应该在操作终端时有过这种操作。
  - SIGQUIT：和 SIGINT 类似，按 `Ctrl+\` 键将发出该信号。它会产生核心转储文件，将内存映像和程序运行时的状态记录下来。
  - SIGKILL：强制杀死进程，本信号不能被阻塞和忽略。
  - SIGTERM：与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出。

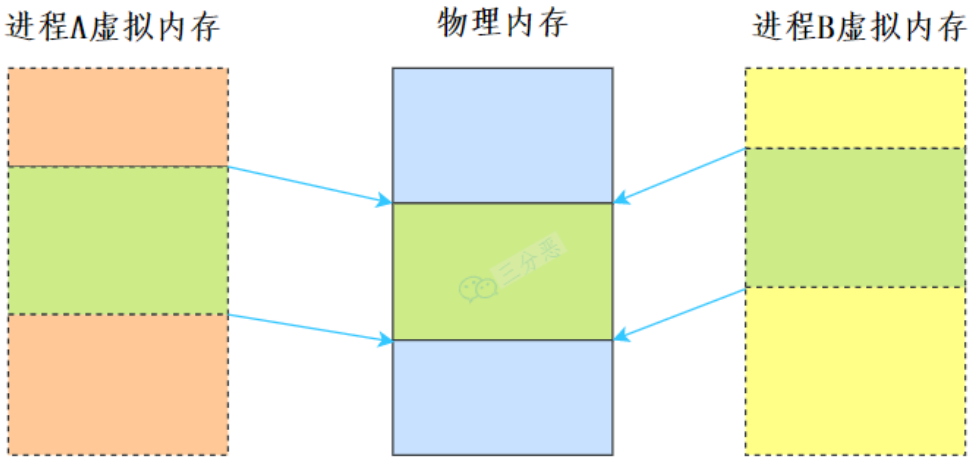
3. 消息队列

- 消息队列是保存在内核中的消息链表，按照消息的类型进行消息传递，具有较高的可靠性和稳定性。
- 缺点：消息体有一个最大长度的限制，不适合比较大的数据传输；存在用户态与内核态之间的数据拷贝开销。



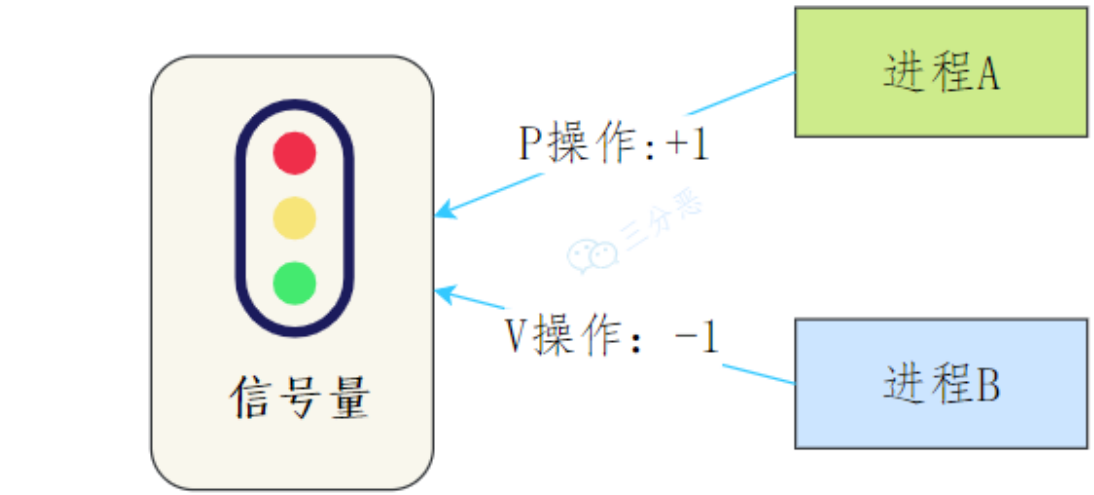
4. 共享内存

- 允许两个或多个进程共享一个给定的内存区，一个进程写入的东西，其他进程马上就能看到。
- 共享内存是最快的进程间通信方式，它是针对其他进程间通信方式运行效率低而专门设计的。
- 缺点：当多进程竞争同一个共享资源时，会造成数据错乱的问题。



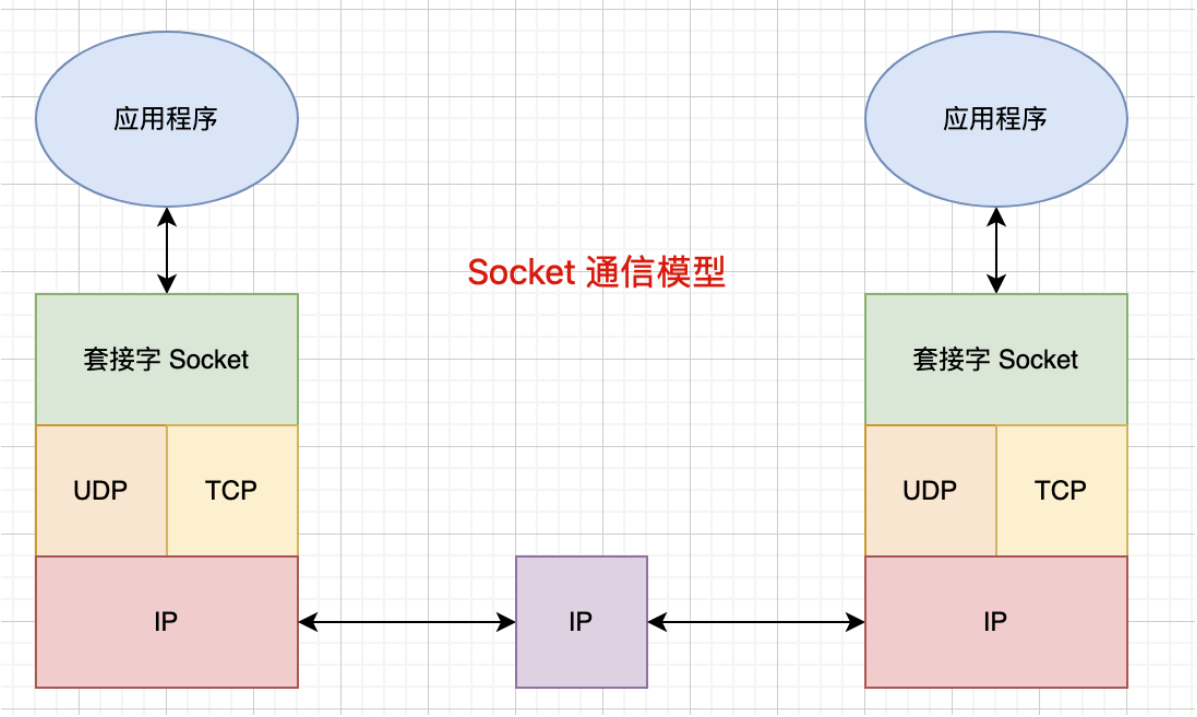
5. 信号量

- 信号量可以理解成红绿灯，红灯停（信号量为零），绿灯行（信号量非零）。**它本质上是一个计数器**，用来控制对共享资源的访问数量。



6. 套接字

- 提供网络通信的端点，可以让不同机器上运行的进程之间进行双向通信。



进程和线程的联系和区别

- 进程是一个正在执行的程序实例。每个进程都有自己独立的地址空间、全局变量、堆栈、和文件描述符等资源。
- 线程是进程中的一个执行单元。一个进程可以包含多个线程，它们共享进程的地址空间和资源。
- 进程切换需要保存和恢复大量的上下文信息，代价较高。线程切换相对较轻量，因为线程共享进程的地址空间，**只需要保存和恢复线程私有的数据。**
- 线程的生命周期由进程控制，进程终止时，其所有线程也会终止。

特性	进程	线程
地址空间	独立	共享

特性	进程	线程
内存开销	高	低
上下文切换	慢，开销大	快，开销小
通信	需要 IPC 机制，开销较大	共享内存，直接通信
创建销毁	开销大，较慢	开销小，较快
并发性	低	高
崩溃影响	一个进程崩溃不会影响其他进程	一个线程崩溃可能导致整个进程崩溃

线程上下文切换了解吗

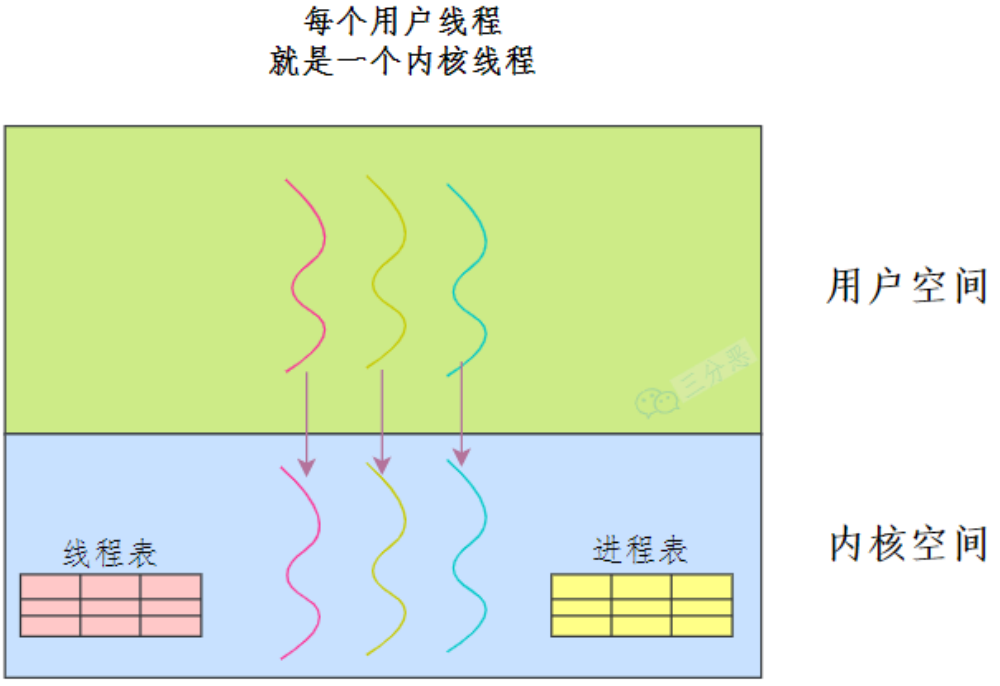
- 当两个线程不是属于同一个进程，则切换的过程就跟进程上下文切换一样；
- 当两个线程属于同一个进程，因为虚拟内存是共享的，所以在切换时，虚拟内存这些资源就保持不动，只需要切换线程的私有数据、寄存器等不共享的数据

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

1. 内核态线程实现

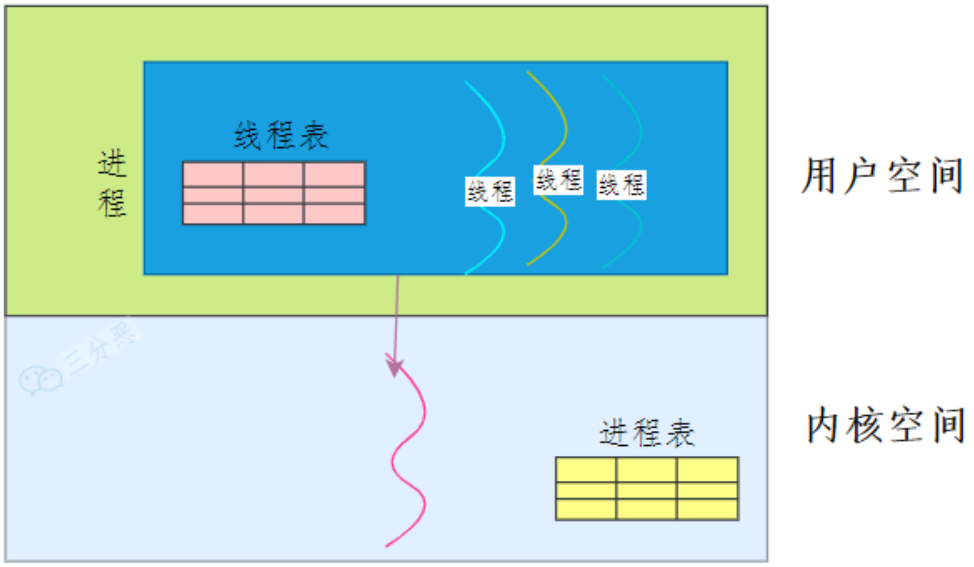
- 在内核空间实现的线程，由内核直接管理直接管理线程



2. 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

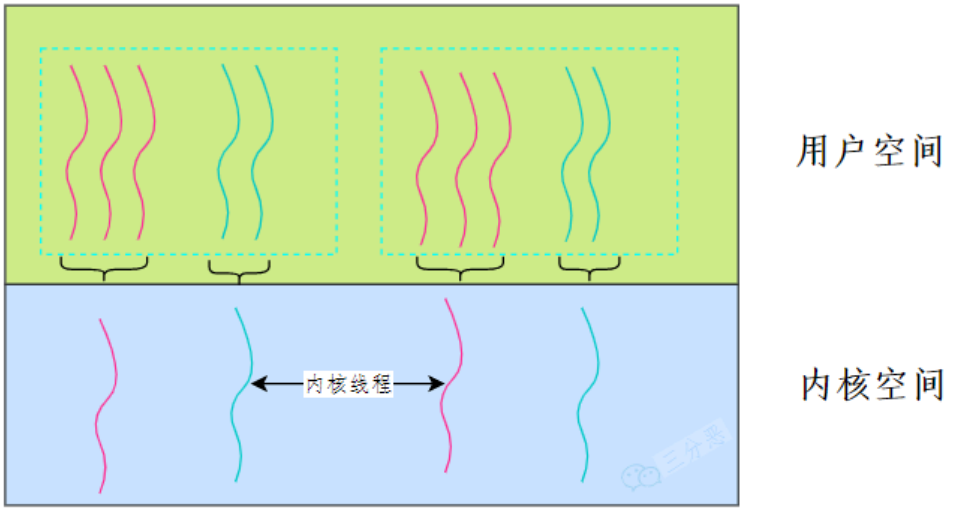
操作系统只看到线程



3.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。
- 多个用户级线程（N）映射到多个内核级线程（M），即 N:M 线程调度。

多个用户线程
共享一个内核线程



用户态线程和内核态线程有什么区别

用户级线程（ULT）：

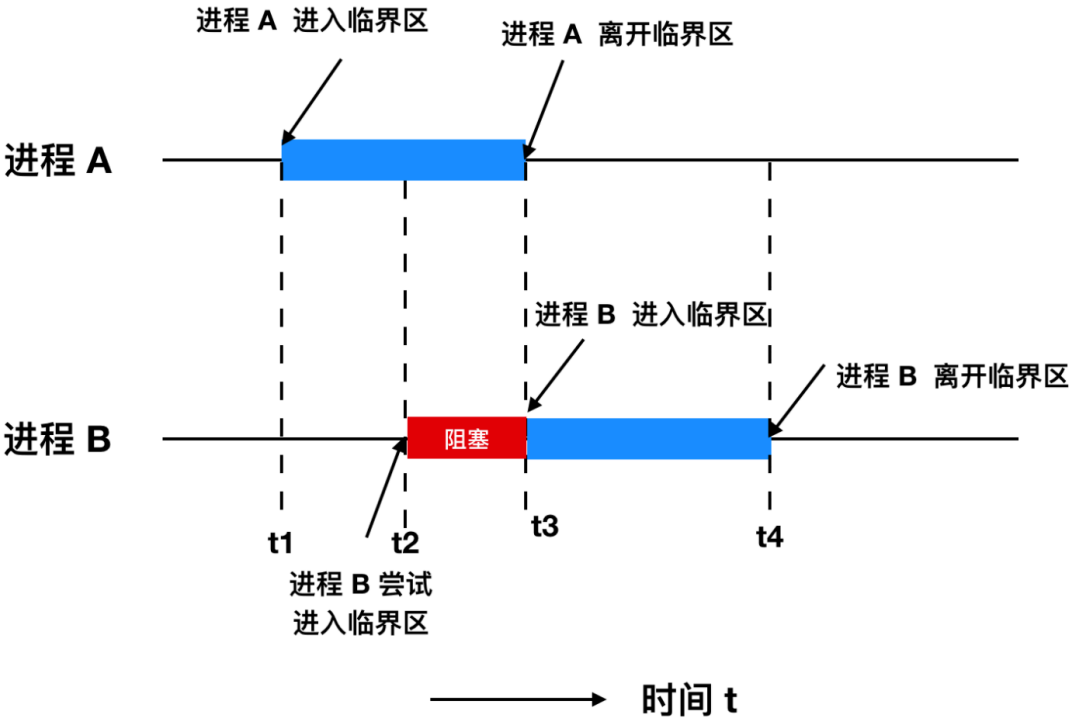
- **优点**：上下文切换快（不需要内核干预）。
- **缺点**：不能利用多核 CPU，且如果一个线程阻塞，整个进程都会被挂起。

内核级线程（KLT）：

- **优点**：支持多核 CPU 并行执行，线程阻塞不影响其他线程。
- **缺点**：线程切换涉及内核态，**开销大**（寄存器切换、TLB 刷新）。

线程间如何同步

同步解决的是多线程操作共享资源的问题，不管线程之间是如何穿插执行的，最后的结果都是正确的。



使用临界区的互斥

临界区：对共享资源访问的程序片段，我们希望这段代码是`互斥`的，可以保证在某个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。

1. 互斥锁:

- 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。
- 根据锁的实现不同，可以分为`忙等待锁`和`无忙等待锁`。
    - 忙等待锁（也称为自旋锁，Spinlock）是指当一个线程试图获取锁时，如果该锁已经被其他线程持有，当前线程不会立即进入休眠或阻塞，而是不断地检查锁的状态，直到该锁可用为止。
        - 优点是避免了线程的上下文切换
        - 缺点是浪费CPU资源
    - 无忙等待锁是指当一个线程尝试获取锁时，如果锁已经被其他线程持有，当前线程不会忙等待，而是主动让出 CPU，进入阻塞状态或休眠状态，等待锁释放
    - 当锁被释放时，线程被唤醒并重新尝试获取锁。

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，分别是：

- *P* 操作：当线程想要进入临界区时，会尝试执行 P 操作。如果信号量的值大于 0，信号量值减 1，线程可以进入临界区；否则，线程会被阻塞，直到信号量大于 0。
- *V* 操作：当线程退出临界区时，执行 V 操作，信号量的值加 1，释放一个被阻塞的线程。

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。

死锁产生有哪些条件

产生死锁需要同时满足四个必要条件:

- **互斥条件**
  - 资源不能被多个进程共享
  - 如果一个资源已经被分配给了一个进程，其他进程必须等待，直到该资源被释放。
- **持有并等待**
  - 一个进程已经持有了至少一个资源,同时还在等待获取其它被占用的资源, 这个期间不会释放已经持有的资源
- **不可剥夺**
  - 已分配给进程的资源不能被强制剥夺，只有持有该资源的进程可以主动释放资源
- **循环等待**
  - 存在一个进程集合 P1,P2,...,Pn，其中 P1 等待 P2 持有的资源，P2 等待 P3 持有的资源，依此类推，直到 Pn 等待 P1 持有的资源，形成一个进程等待环。

如何避免死锁

打破任意一个条件即可

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。

消除循环等待条件

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

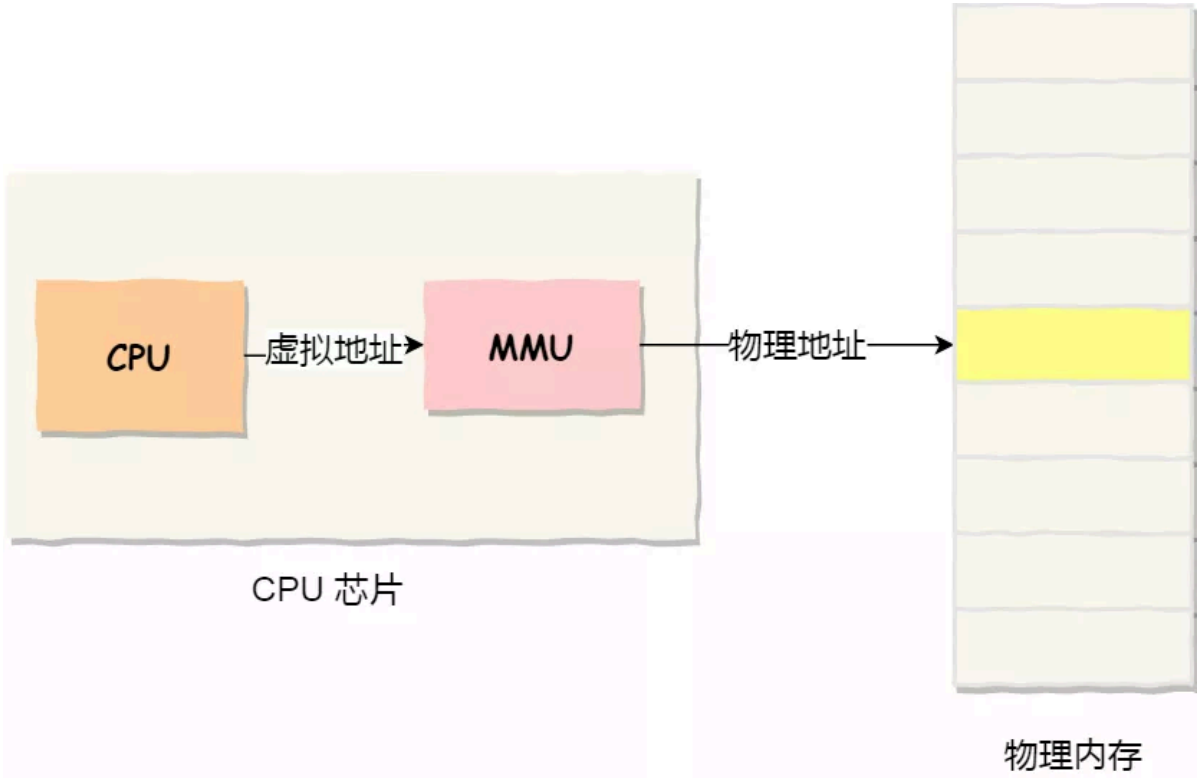
- 这个饥饿指的是资源饥饿, 某个线程一直等不到它所需要的资源,从而无法向前推进

活锁

- 两个线程不断互相让步，导致没有线程能真正完成任务

内存管理

物理内存和虚拟内存有什么区别



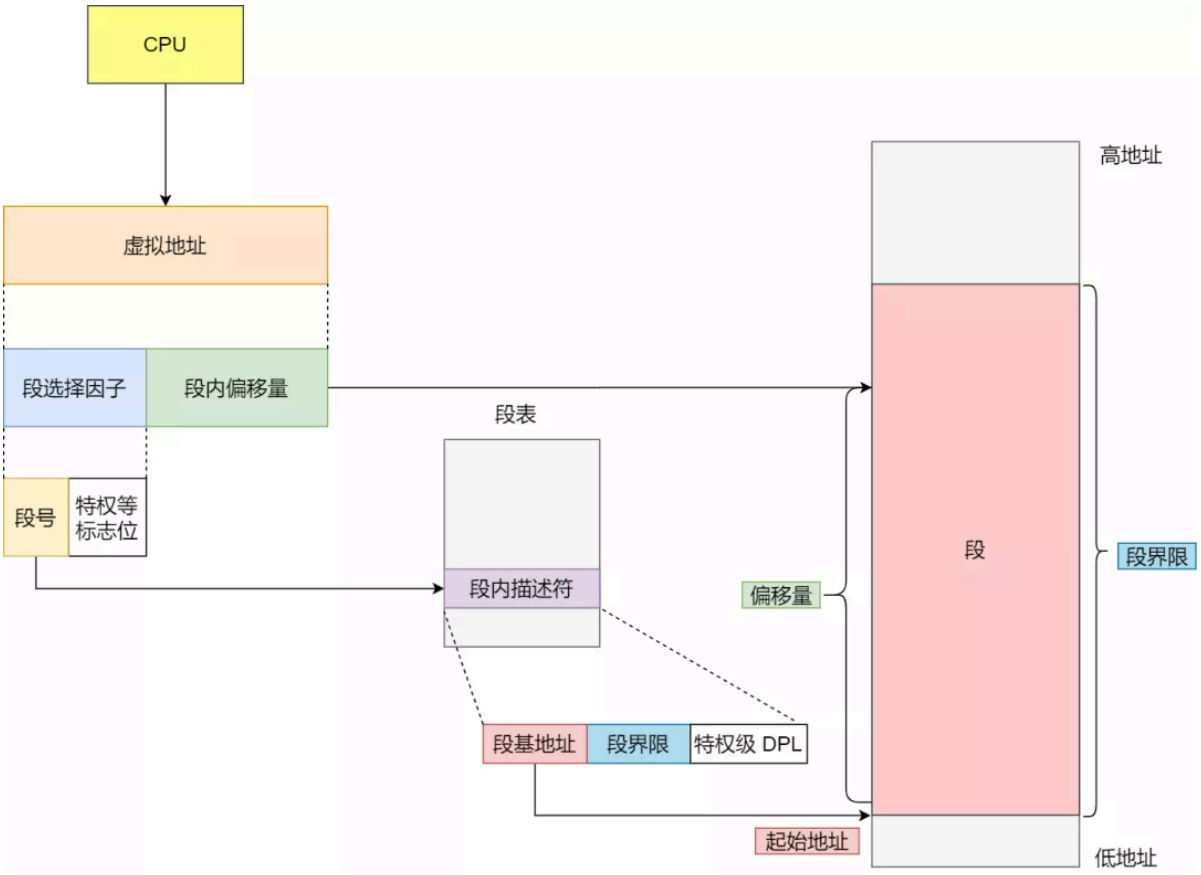
物理内存指的是计算机中实际存在的硬件内存。物理内存是计算机用于存储运行中程序和数据的实际内存资源，操作系统和应用程序最终都必须使用物理内存来执行。

虚拟内存是操作系统提供的一种内存管理技术，它使得应用程序认为自己有连续的、独立的内存空间，而实际上，这个虚拟内存可能**部分存储在物理内存上，部分存储在磁盘（如硬盘的交换分区或页面文件）**中。

虚拟内存的核心思想是通过硬件和操作系统的配合，为每个进程提供一个独立的、完整的虚拟地址空间，解决物理内存不足的问题。

- 每个进程都有自己的虚拟地址空间，虚拟内存使用的是逻辑地址，它与实际的物理内存地址不同，必须经过地址转换才能映射到物理内存。
- 操作系统通过 **页表（Page Table）** 将虚拟地址映射到物理地址。当程序访问某个虚拟地址时，CPU 会通过页表找到对应的物理地址。
- 操作系统将虚拟内存划分为若干个**页（Pages）**，每个页可以被映射到物理内存中的一个页面。如果物理内存不够，操作系统会将不常用的页暂时存储到磁盘的交换区（Swap）中，这个过程叫做页交换（Paging）。

什么是内存分段

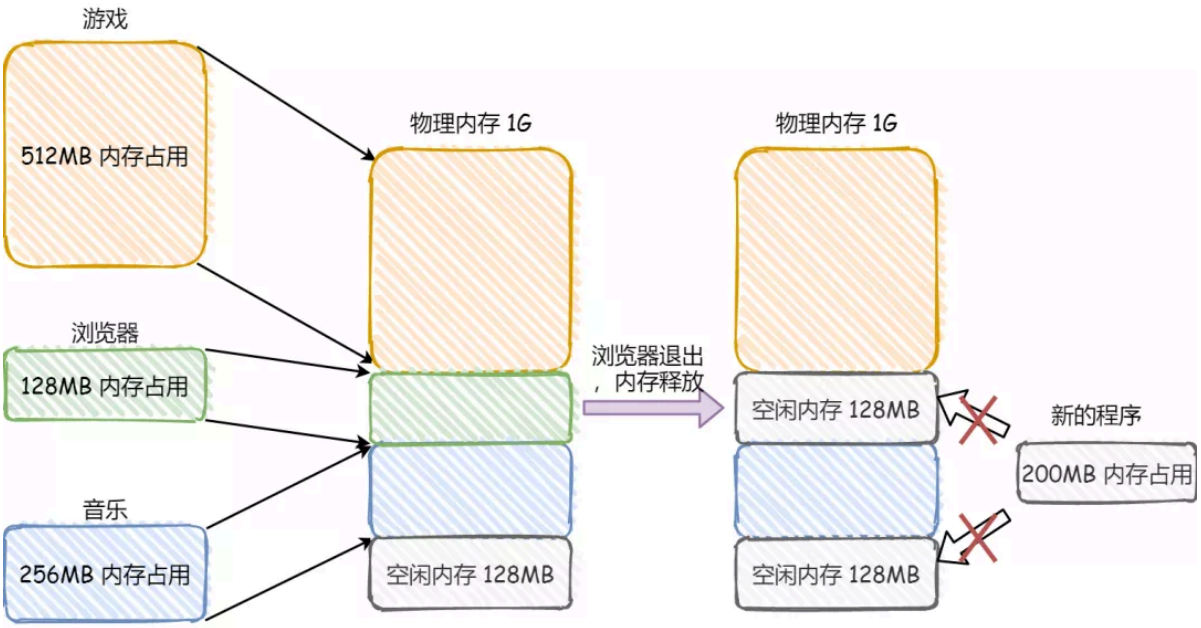


内存分段是**操作管理虚拟地址与物理地址之间关系**的方式之一，还有一种是内存分页。

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，所以就用分段的形式把这些段分离出来。**

分段的好处是能产生连续的内存空间，但是会出现内存碎片和内存交换的空间太大的问题。

分段为什么会产生内存碎片问题



这里的内存碎片的问题共有两处地方：

- **外部内存碎片**，也就是产生了多个不连续的小物理内存，导致新的程序无法被装载；
- **内部内存碎片**，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；

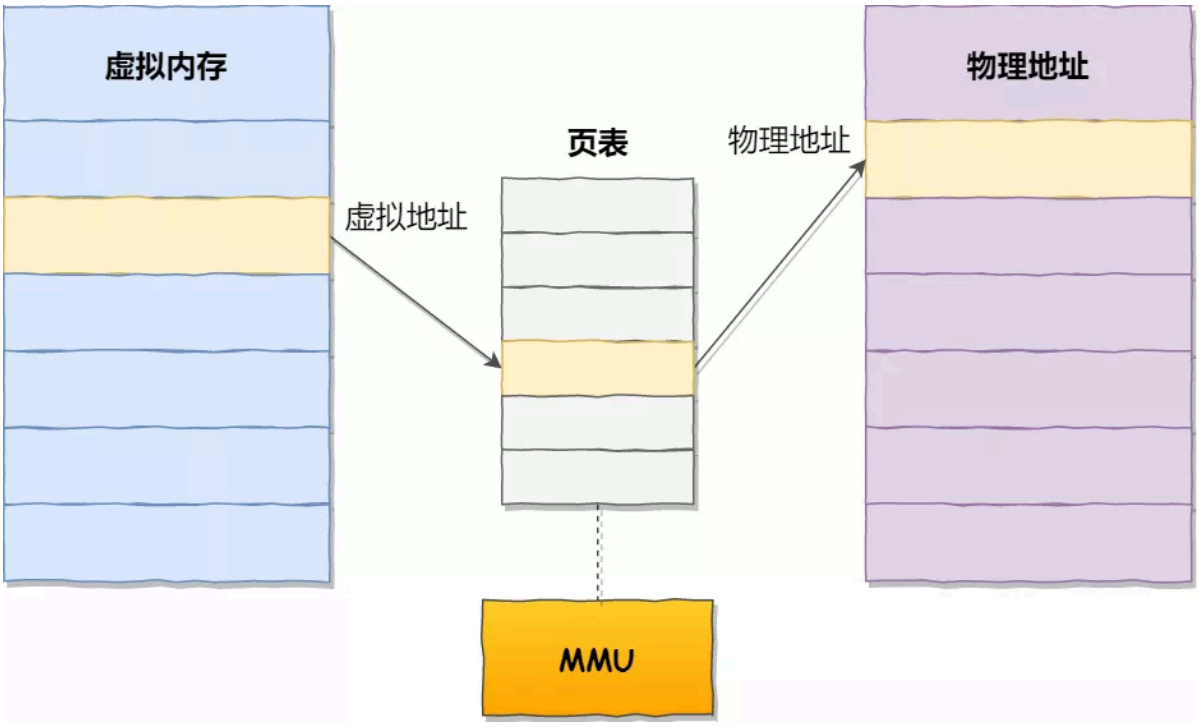
解决外部内存碎片的问题就是**内存交换。**

- 可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回来到内存里。不过再读回的时候，我们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 程序就可以装载进来，但是效率很低。

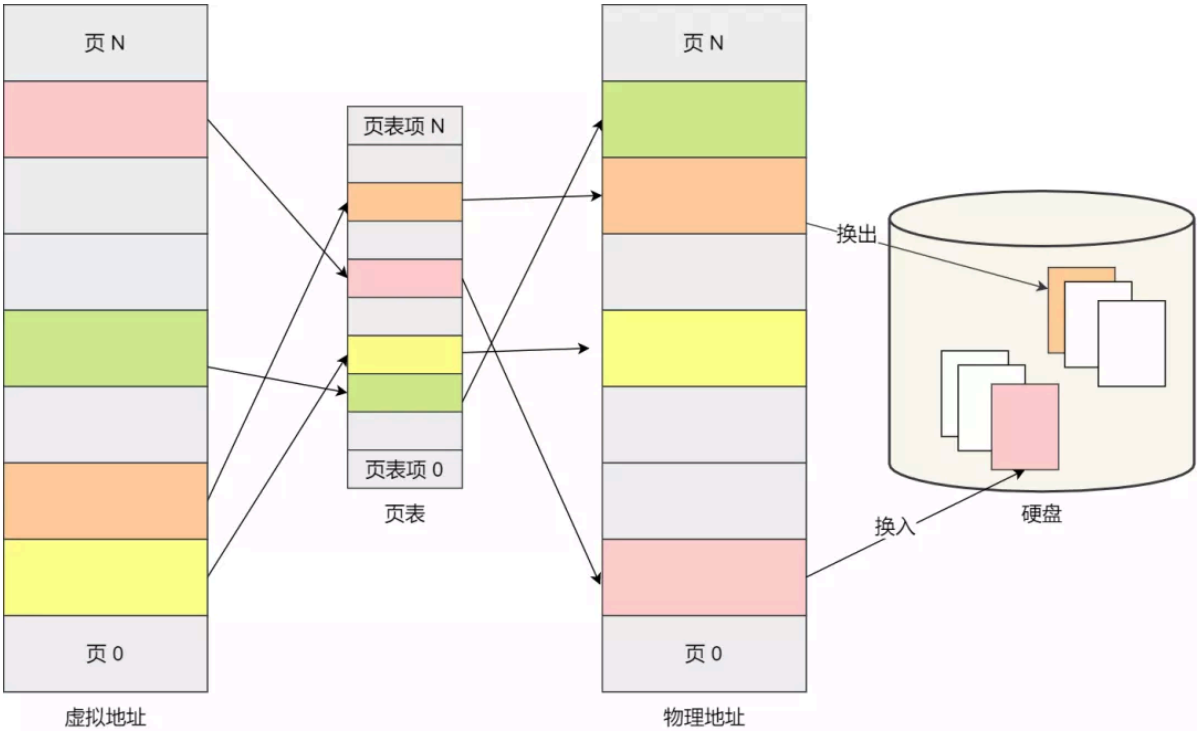
什么是内存分页

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就叫作页。

虚拟地址和物理地址之间通过**页表**来映射：



分页是怎么解决分段的内存碎片和内存交换效率低的问题的

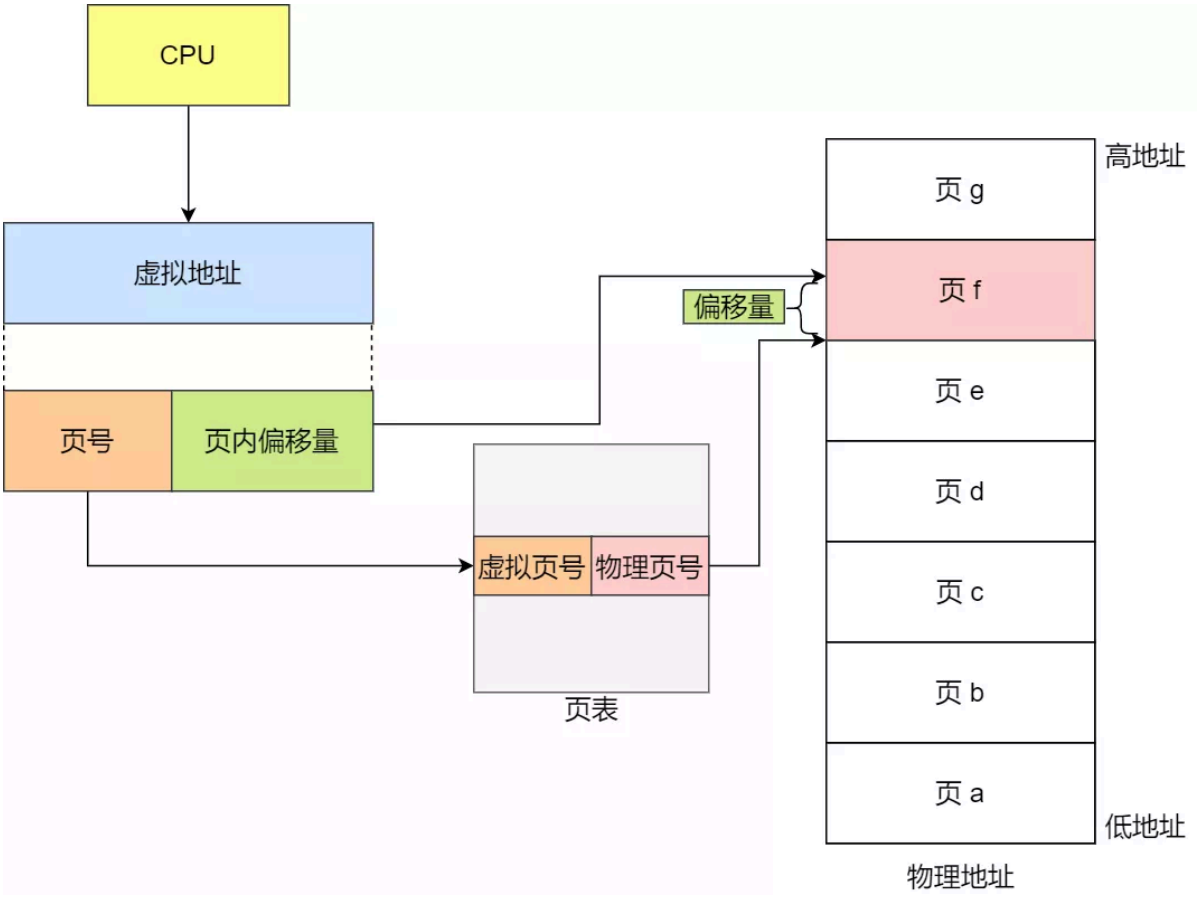


由于内存空间都是预先划分好的，也就不会像分段会产生间隙非常小的内存，这正是分段会产生内存碎片的原因。而**采用了分页，那么释放的内存都是以页为单位释放的，也就不会产生无法给进程使用的小内存。**

如果内存空间不够，操作系统会把其他正在运行的进程中的「最近没被使用」的内存页面给释放掉，也就是暂时写在硬盘上，称为**换出**（*Swap Out*）。一旦需要的时候，再加载进来，称为**换入**（*Swap In*）。所以，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。**

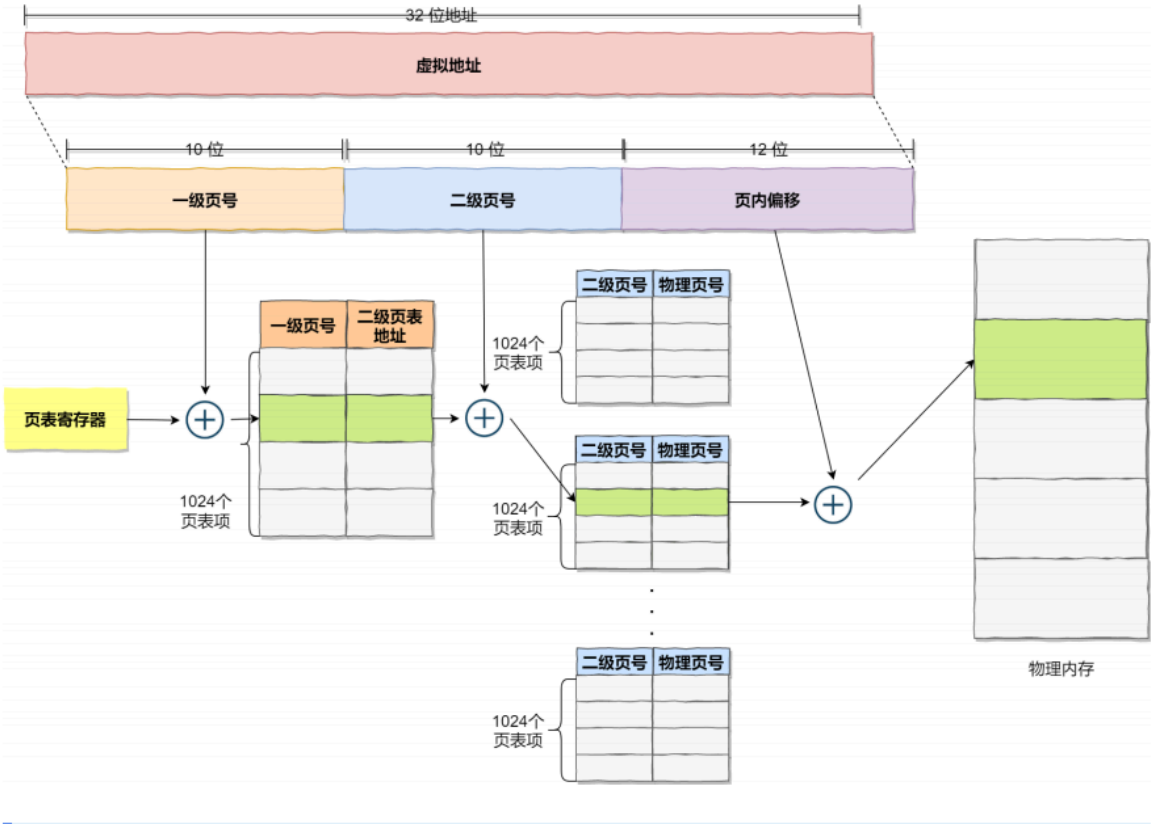
分页机制下，虚拟地址和物理地址是如何映射的

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。



多级页表知道吗

多级页表（Multilevel Page Table）是一种内存管理技术，用于在虚拟内存系统中高效地管理和转换虚拟地址到物理地址。它通过分层结构减少页表所需的内存开销，以解决单级页表在大地址空间中的效率问题。



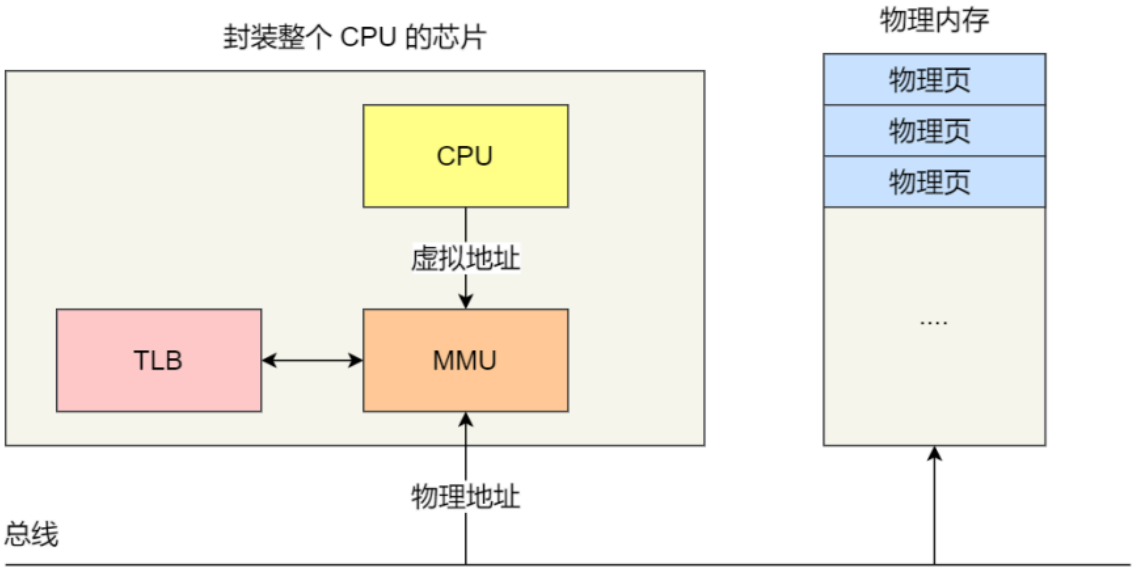
第一层存放**页目录**，每个目录项指向一个 **页表**。这样，一级页表就可以覆盖整个 4GB 虚拟地址空间，但**如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。**

那么为什么不分级的页表就做不到这样节约内存呢？我们从页表的性质来看，保存在内存中的页表承担的职责是将虚拟地址翻译成物理地址。假如虚拟地址在页表中找不到对应的页表项，计算机系统就不能工作了。所以**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）。

什么是快表

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。



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)。

虚拟内存的可用容量 = 物理内存的容量 + 交换空间的容量

什么是缺页中断

缺页中断（Page Fault）是虚拟内存管理的一个重要概念。当一个程序访问的页（页面）不在物理内存中时，就会发生缺页中断。操作系统需要从磁盘上的交换区（或页面文件）中将缺失的页调入内存。

页面置换算法有哪些

1. 最佳页面置换算法

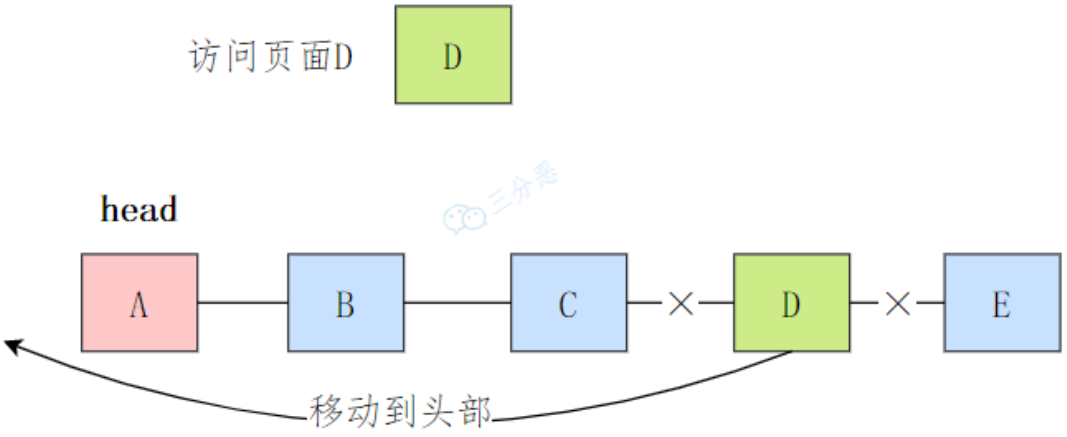
基本思路是，淘汰以后不会使用的页面。这是理论上的最佳算法，因为它可以保证最低的缺页率。但在实际应用中，由于无法预知未来的访问模式，OPT 通常无法实现。

2. 先进先出置换算法

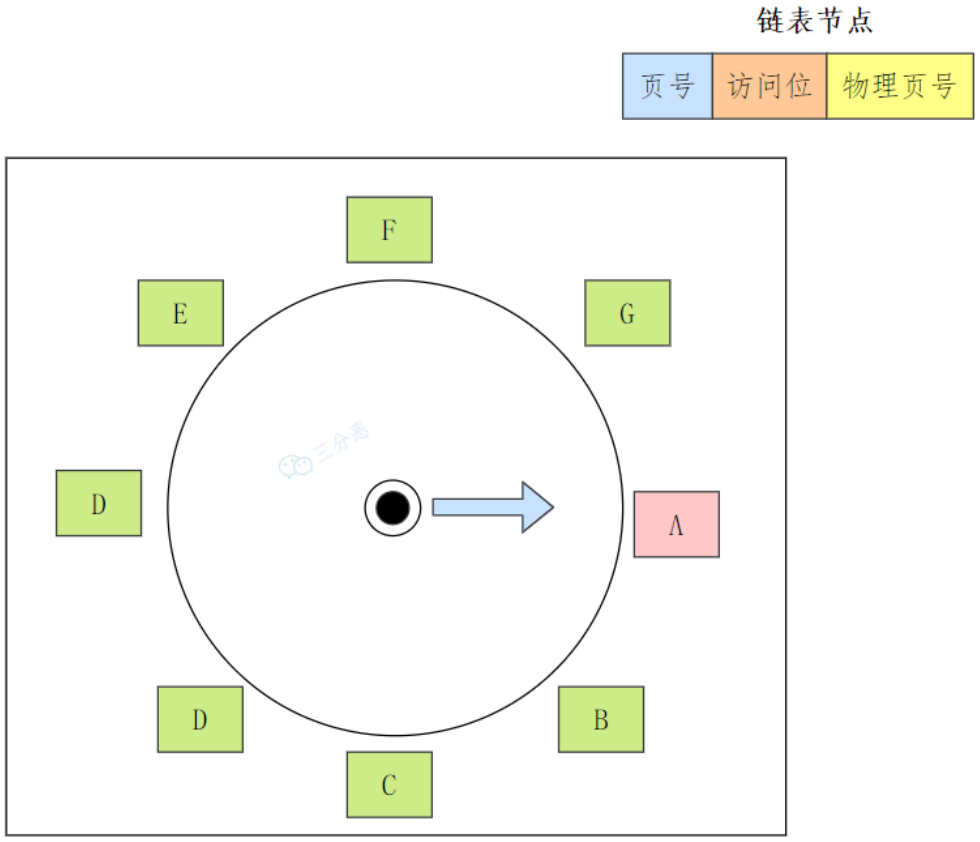
基本思路是，优先淘汰最早进入内存的页面。FIFO 算法维护一个队列，新来的页面加入队尾，当发生页面置换时，队头的页面（即最早进入内存的页面）被移出。

3. 最近最久未使用的置换算法

淘汰最近没有使用的页面。LRU 算法根据页面的访问历史来进行置换，最长时间未被访问的页面将被置换出去。



4. 时钟页面置换算法



一个指针（Clock Hand） 在队列中顺时针移动，寻找可替换的页面：

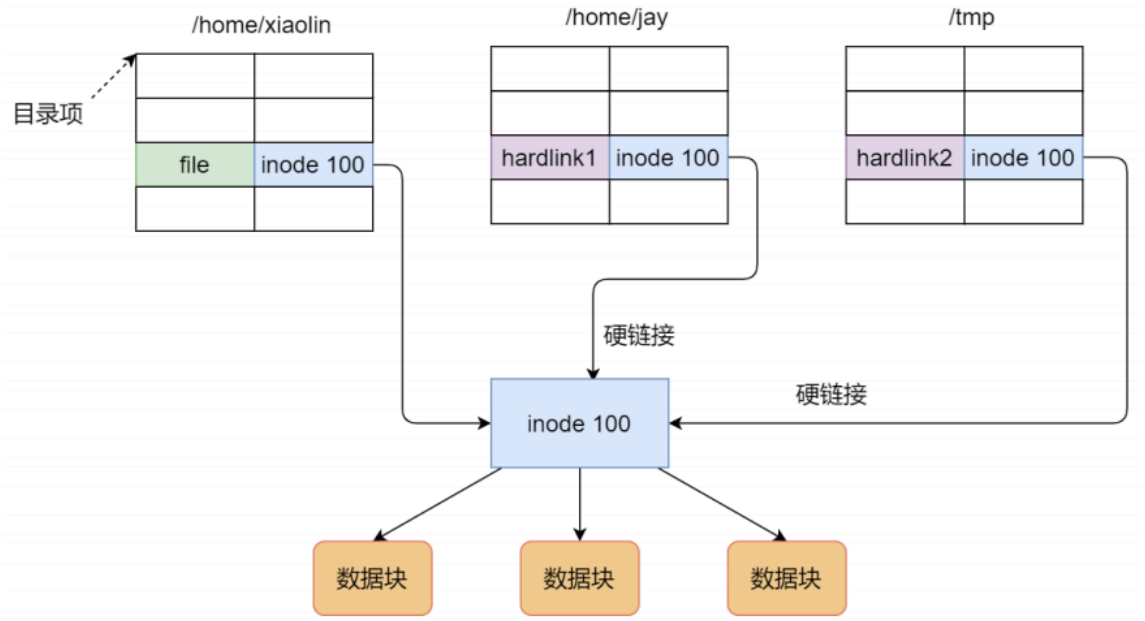
- **如果当前页面的访问位是 1**：清除该访问位（设为 0），指针继续向前移动。
- **如果当前页面的访问位是 0**：表示该页面较久未使用，直接替换它。

5. 最不常用置换算法

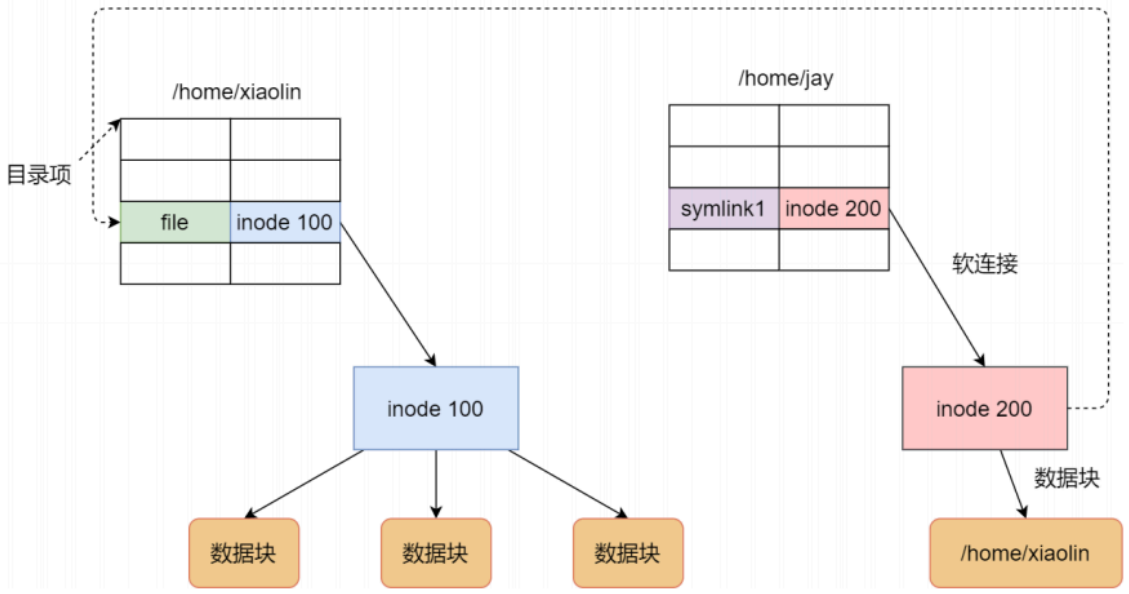
根据页面被访问的频率进行置换，访问次数最少的页面最先被置换。实现较为复杂，需要记录每个页面的访问频率。

文件

硬链接和软连接有什么区别



硬链接，以文件副本的形式存在，所有的硬链接都指向同一个iNode X，他们都享有同一个inode X和一个数据块（data block）。但硬链接本身并不占用实际存储空间。如果A文件和B文件的关系是硬链接的关系，当用户修改了A文件的内容，那么B文件的内容也会发生更改。反之一样。

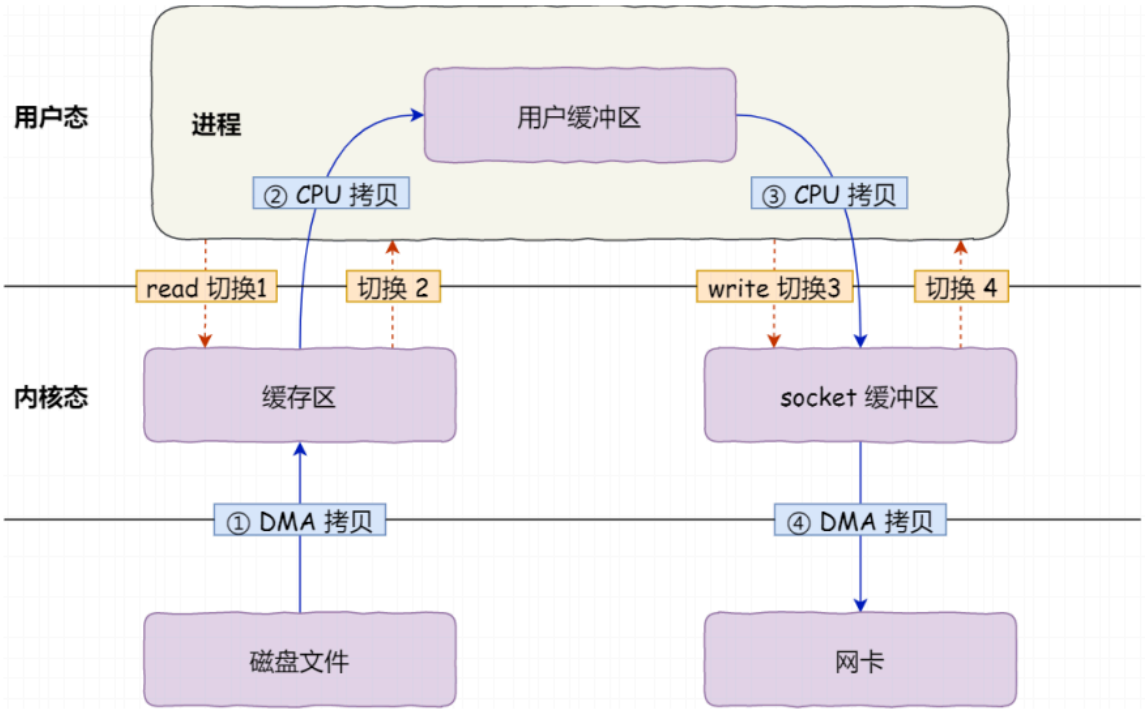


软链接（file-软链接）本身是一个单独的文件,拥有自己的inode Y 和自己的数据块。因此拥有自己的文件属性和权限。 最大的特点是：明确自己是符号链接文件,并存放了指向源文件（file）的inode X（告知别人自己是继承谁）

软链接类似Windows中的快捷方式，为一个源文件创建一个快捷方式。如果源文件被删除了，也没有办法使用该快捷方式；一旦以同样文件名创建了源文件，链接将继续指向该文件的新数据。

IO

零拷贝了解吗



如图所示，展示了从磁盘文件读取数据并通过网络发送文件的流程。整个流程涉及**四次数据拷贝**和**四次上下文切换。**

四次拷贝：

- DMA 拷贝（磁盘 → 内核缓冲区）
- CPU 拷贝（内核缓冲区 → 用户缓冲区）
- CPU 拷贝（用户缓冲区 → socket 缓冲区）
- DMA 拷贝（socket 缓冲区 → 网卡）

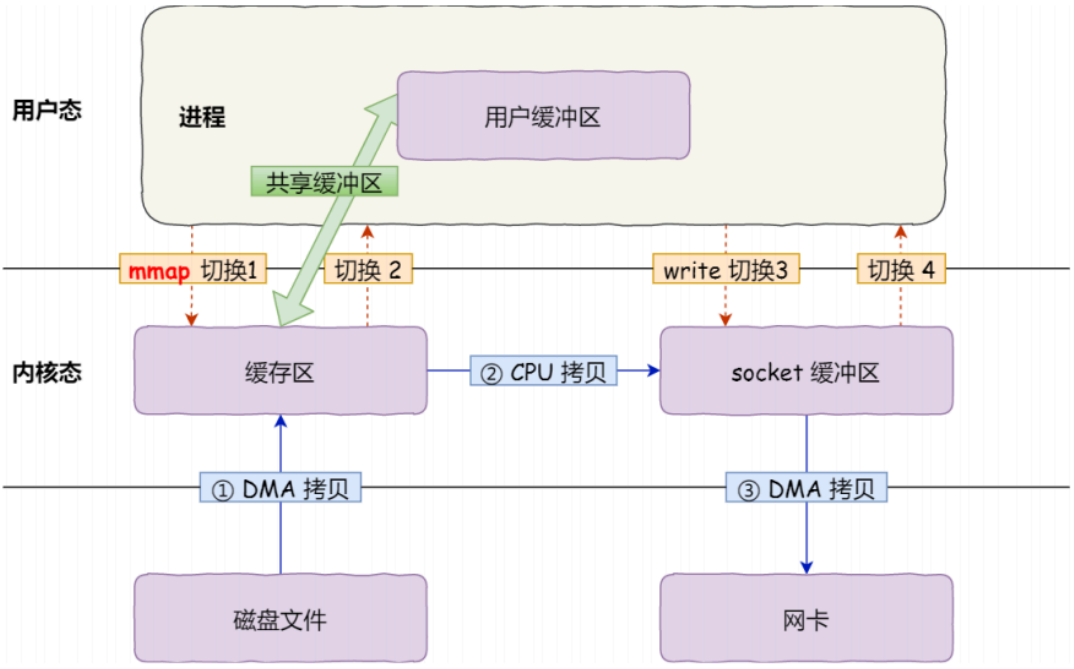
四次上下文切换：

- `read()` 触发 用户态 → 内核态
- `read()` 返回 内核态 → 用户态
- `write()` 触发 用户态 → 内核态
- `write()` 返回 内核态 → 用户态

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。**

零拷贝技术的实现主要有两种：

- mmap + write



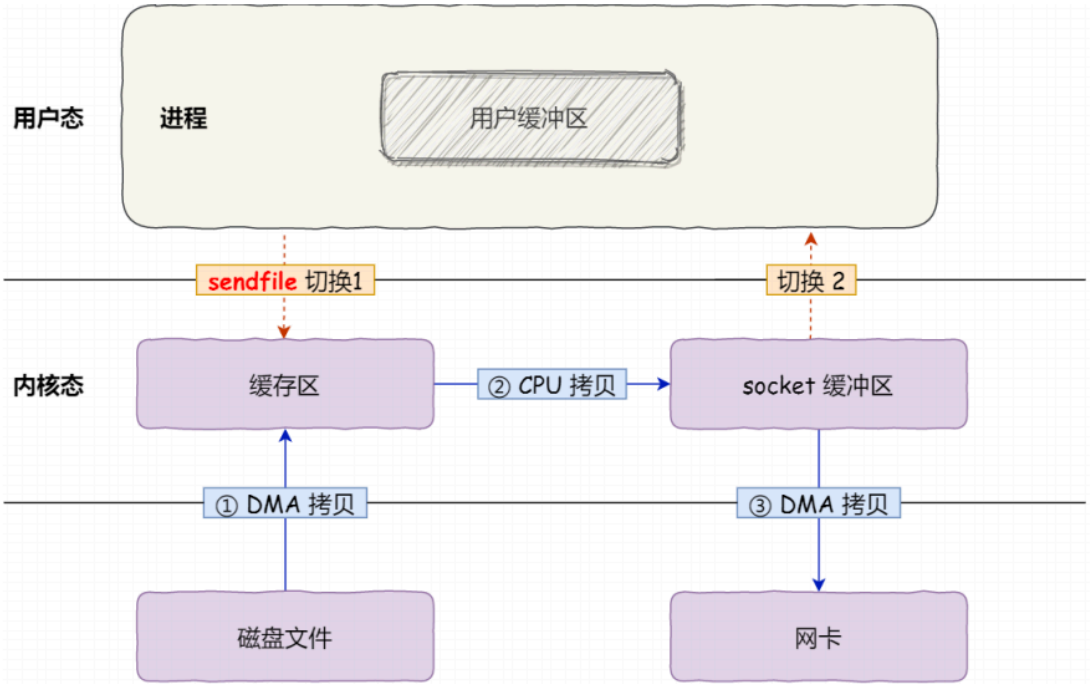
DMA 拷贝（磁盘 → 内核缓冲区）

用户进程通过 `mmap` 访问数据（零拷贝，无 CPU 拷贝）

CPU 拷贝（共享缓冲区 → socket 缓冲区）

DMA 拷贝（socket 缓冲区 → 网卡）

- sendfile



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。

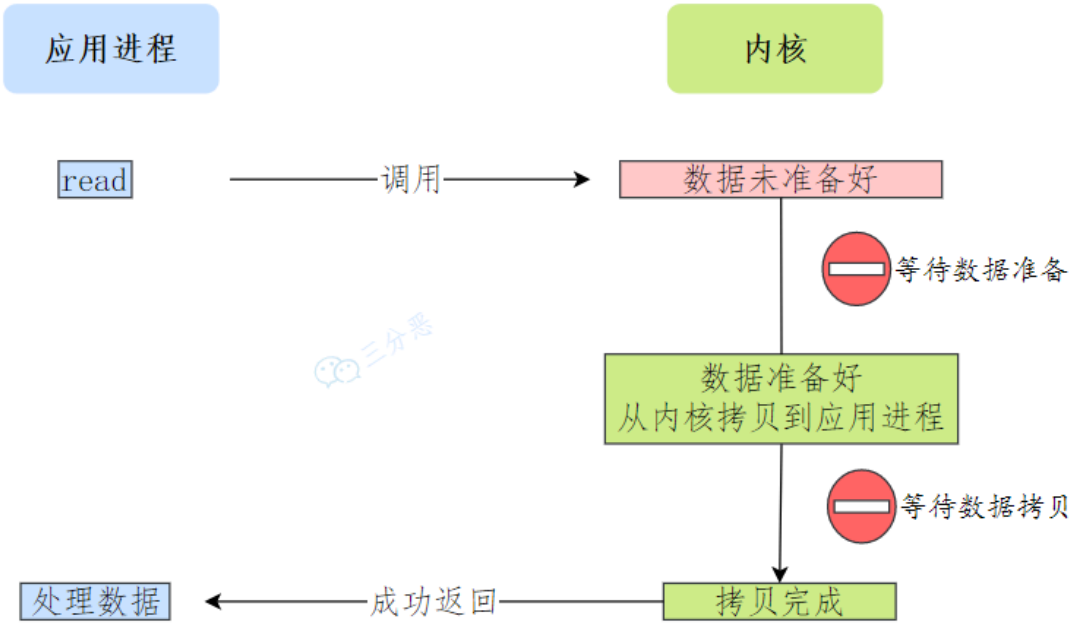
其次，该系统调用，可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，不再拷贝到用户态，这样就只有 2 次上下文切换，和 3 次数据拷贝。

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

Kafka是sendfile，RocketMQ是mmap

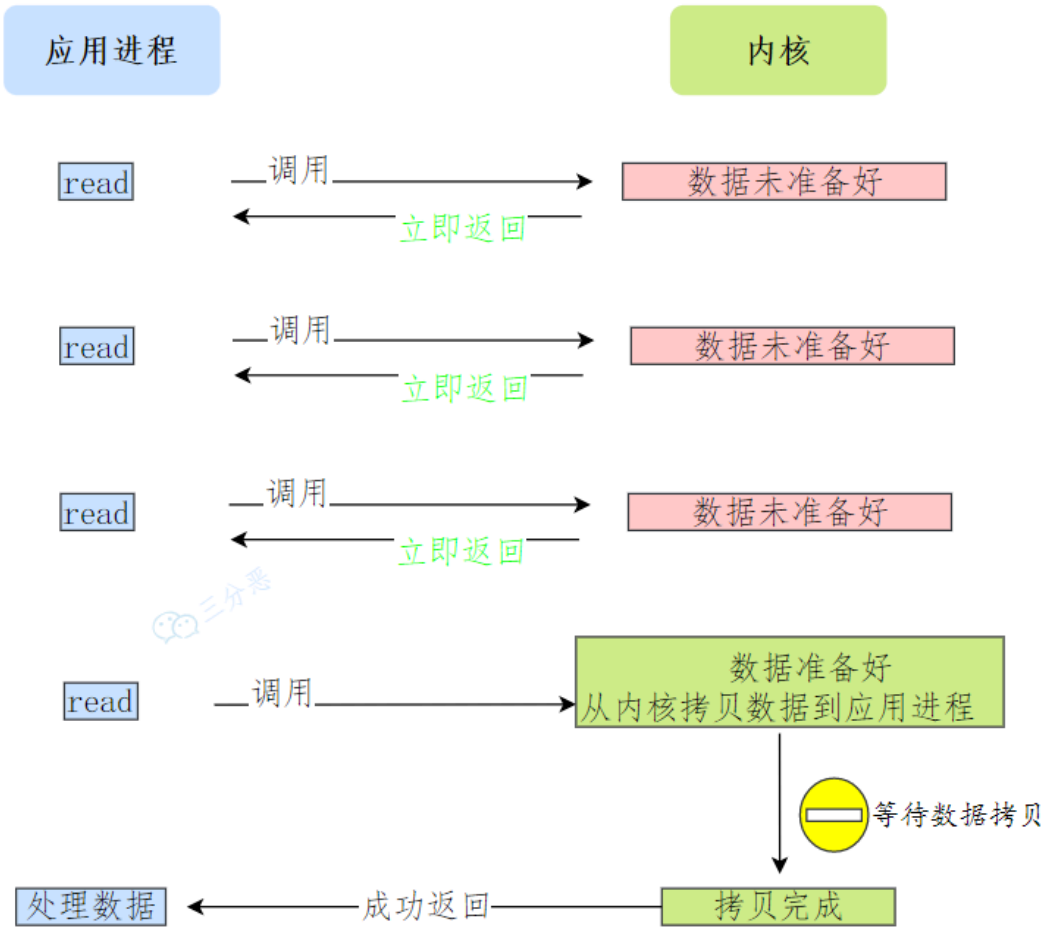
聊聊非阻塞IO、同步、异步IO

1. 阻塞I/O



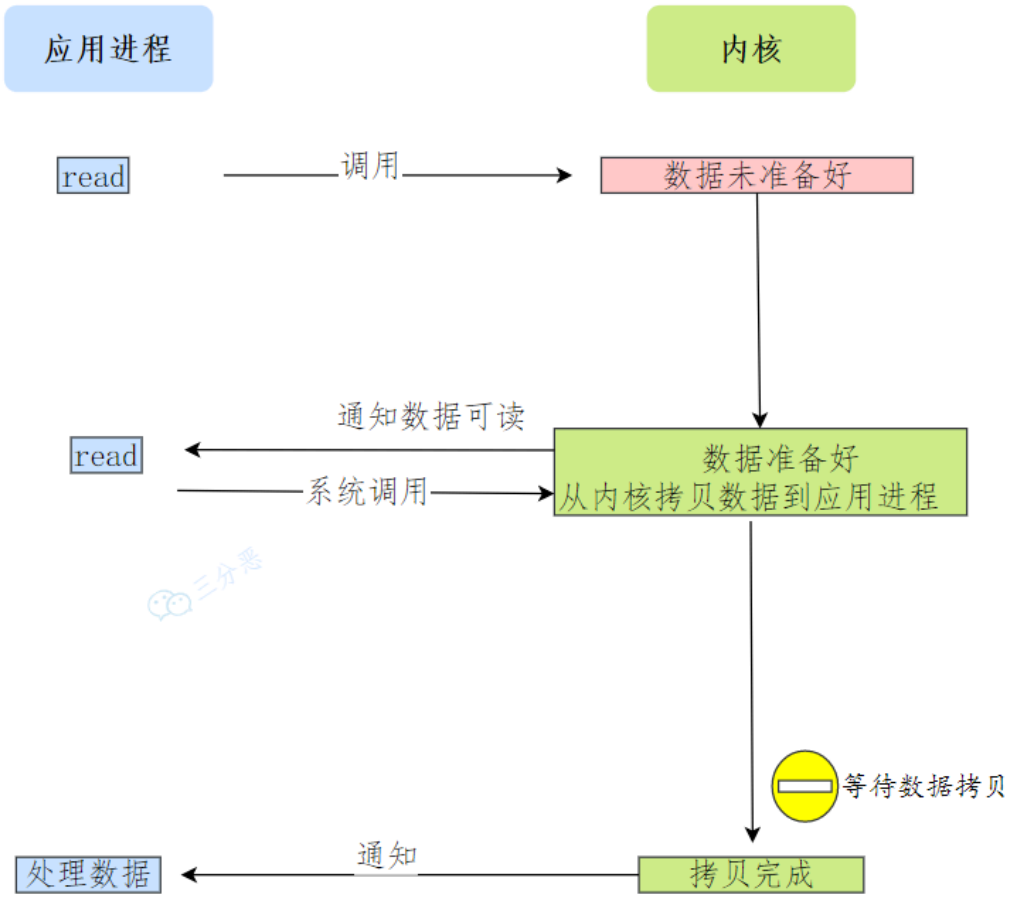
- 用户程序执行 read ，线程会被阻塞，一直等到内核数据准备好，并把数据从内核缓冲区拷贝到应用程序的缓冲区中，当拷贝过程完成， read 才会返回。
- 阻塞等待的是**内核数据准备好**和**数据从内核态拷贝到用户态**这两个过程。

2. 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。

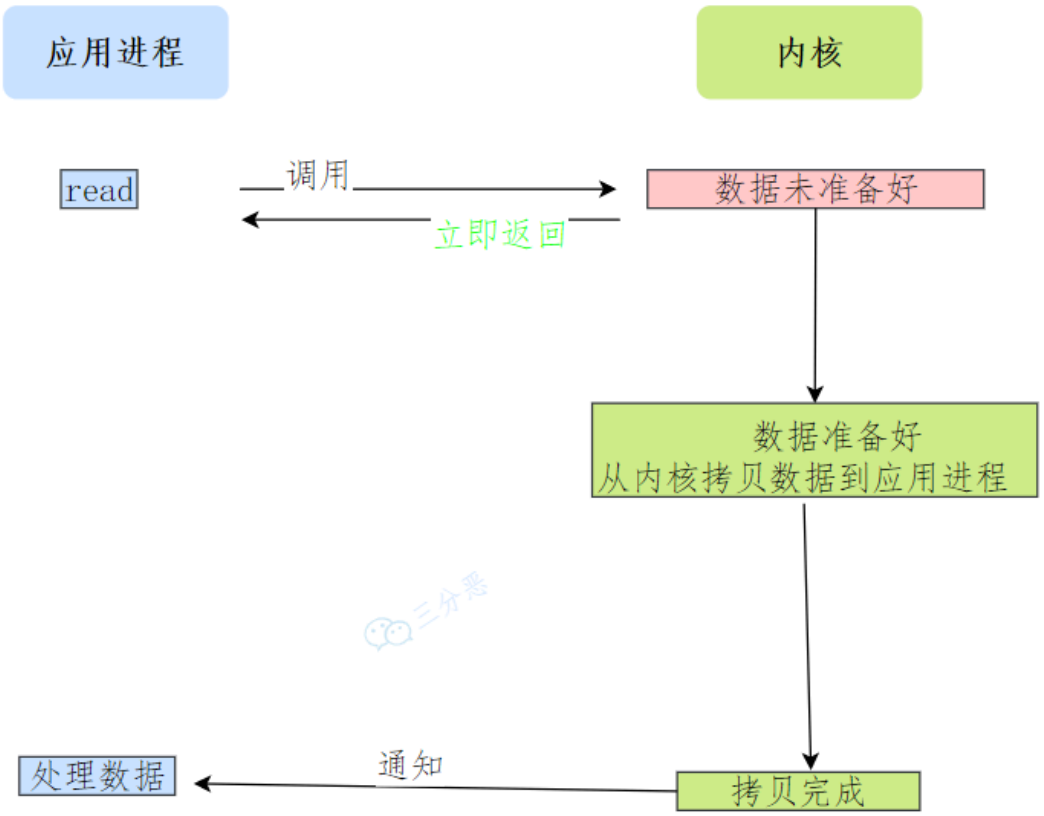
3. 基于非阻塞的I/O多路复用



- 当内核数据准备好时，才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。

4. 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。

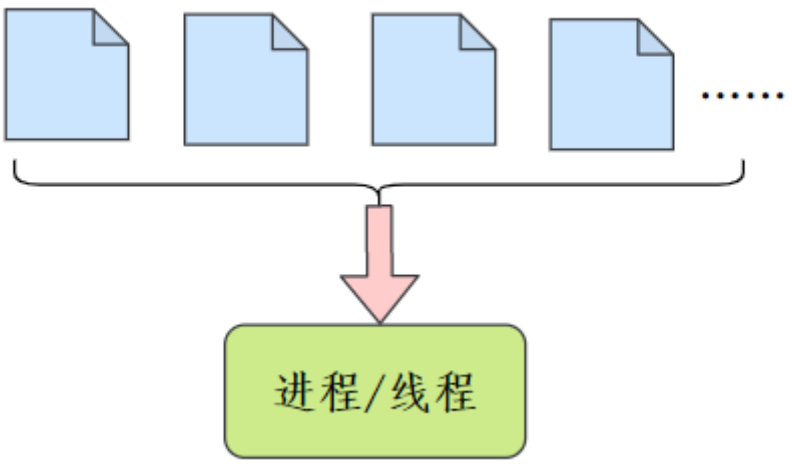
详细讲一讲IO多路复用

在传统的 I/O 模型中，如果服务端需要支持多个客户端，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。



select

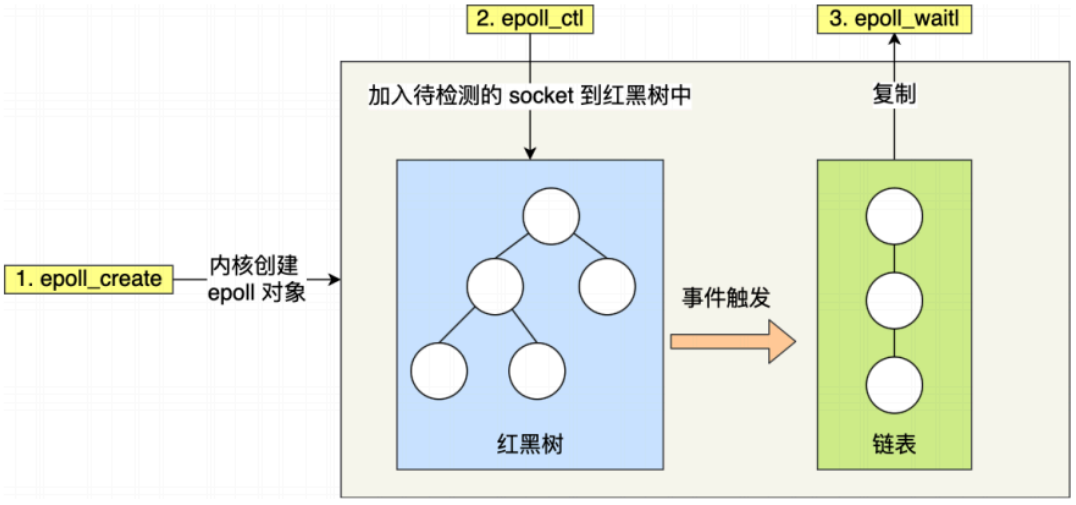
- 将已连接的 Socket 都放到一个**文件描述符集合**fd_set，然后调用 select 函数将 fd_set 集合拷贝到内核里，让内核来检查是否有网络事件产生，检查的方式很粗暴，就是通过遍历 fd_set 的方式，当检查到有事件产生后，将此 Socket 标记为可读或可写， 接着再把整个 fd_set 拷贝回用户态里，然后用户态还需要再通过遍历的方法找到可读或可写的 Socket，再对其处理。
- select 使用固定长度的 BitsMap，表示文件描述符集合，而且所支持的文件描述符的个数是有限制的，在 Linux 系统中，由内核中的 FD_SETSIZE 限制， 默认最大值为 1024 ，只能监听 0~1023 的文件描述符。

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 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。

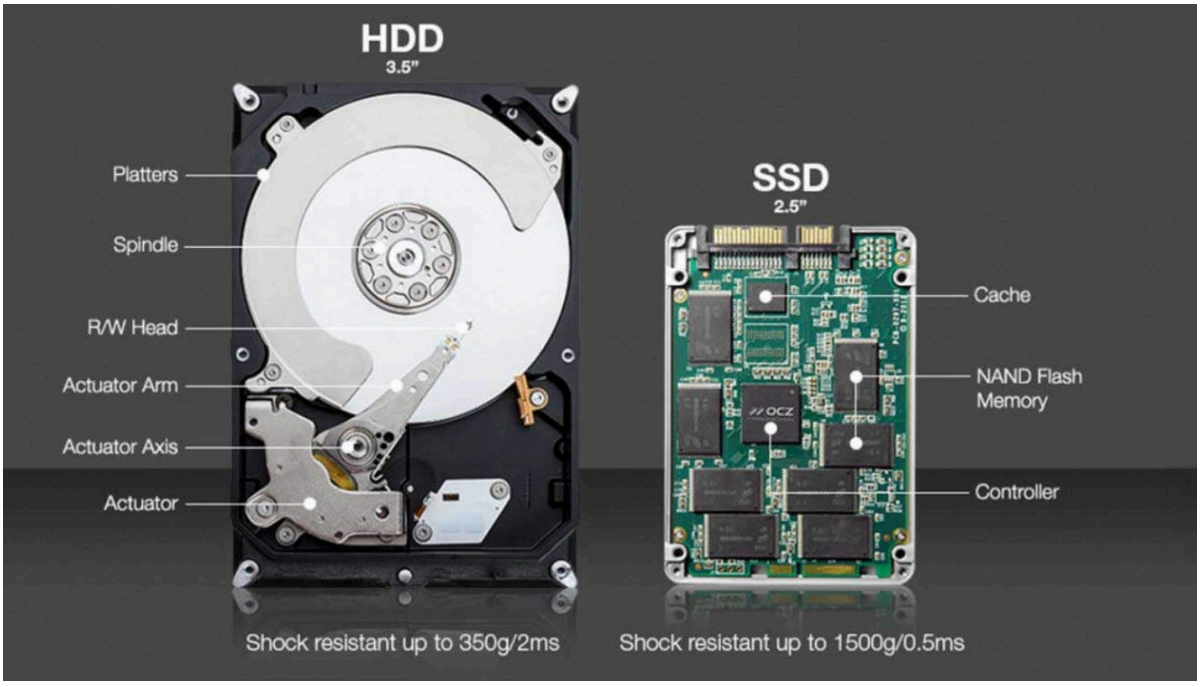
epoll

-



- 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
- 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。

普通内存比一般的机械硬盘快多少



机械硬盘，也叫 HDD（Hard Disk Drive），是一种通过磁盘旋转和磁头移动来存储数据的设备，读写速度比较慢。

- HDD 的访问时间大约在 5-10ms，数据传输速率约为 100 到 200 MB/s。
- 内存，也就是 RAM（Random Access Memory），访问时间大约在 10-100ns，数据传输速率约为数十 GB/s。

固态硬盘（Solid State Drive，SSD），SSD 的读写速度比 HDD 快 200 倍左右，价格也在逐渐下降，已经逐渐取代了 HDD。