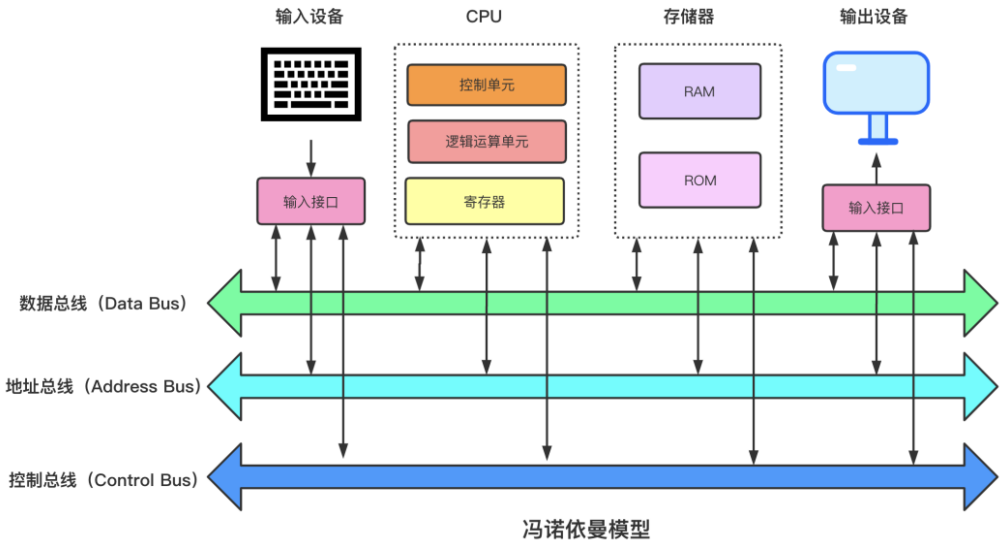
更多嵌入式 Linux 学习资料，请关注：一口 Linux 回复关键字:1024

1 冯诺伊曼体系

1.1 冯诺伊曼体系简介

现代计算机之父冯诺伊曼最先提出程序存储的思想，并成功将其运用在计算机的设计之中，该思想约定了用二进制进行计算和存储，还定义计算机基本结构为 5 个部分，分别是**中央处理器（CPU）、内存、输入设备、输出设备、总线**。



冯诺依曼模型

1. **存储器**：代码跟数据在 RAM 跟 ROM 中是线性存储， 数据存储的单位是一个二进制位。最小的存储单位是字节。

2. **总线**：总线是用于 CPU 和内存以及其他设备之间的通信，总线主要有三种：

地址总线：用于指定 CPU 将要操作的内存地址。

数据总线：用于读写内存的数据。

控制总线：用于发送和接收信号，比如中断、设备复位等信号，CPU 收到信号后响应，这时也需要控制总线。

1. **输入/输出设备**：输入设备向计算机输入数据，计算机经过计算后，把数据输出给输出设备。比如键盘按键时需要和 CPU 进行交互，这时就需要用到控制总线。
2. **CPU**：中央处理器，类比人脑，作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。CPU 用寄存器存储计算时所需数据，寄存器一般有三种：

通用寄存器：用来存放需要进行运算的数据，比如需进行加法运算的两个数据。

程序计数器：用来存储 CPU 要执行下一条指令所在的内存地址。

指令寄存器：用来存放程序计数器指向的指令本身。

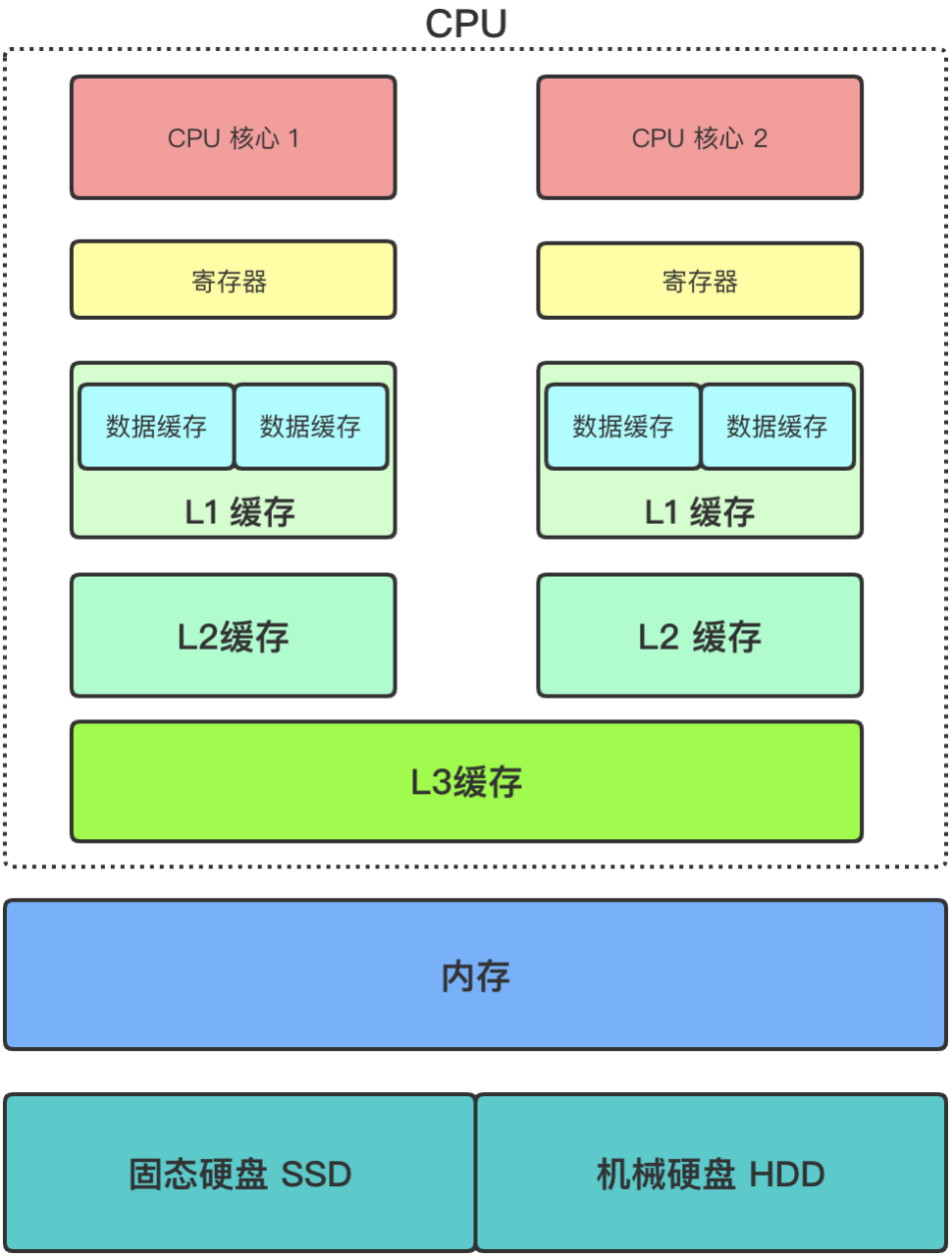
在冯诺伊曼体系下电脑指令执行的过程：

1. CPU 读取程序计数器获得指令内存地址，CPU 控制单元操作地址总线从内存地址拿到数据，数据通过数据总线到达 CPU 被存入指令寄存器。
2. CPU 分析指令寄存器中的指令，如果是计算类型的指令交给逻辑运算单元，如果是存储类型的指令交给控制单元执行。
3. CPU 执行完指令后程序计数器的值通过自增指向下个指令，比如 32 位 CPU 会自增 4。
4. 自增后开始顺序执行下一条指令，不断循环执行直到程序结束。

CPU 位宽：32 位 CPU 一次可操作计算 4 个字节，64 位 CPU 一次可操作计算 8 个字节，这个是硬件级别的。平常我们说的 32 位或 64 位操作系统指的是软件级别的，指的是程序中指令多少位。
线路位宽：CPU 操作指令数据通过高低电压变化进行数据传输，传输时候可以串行传输，也可以并行传输，多少个并行等于多少个位宽。

1.2 CPU 简介

Central Processing Unit 中央处理器，作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。



CPU

1. CPU 核心：一般一个 CPU 会有多个 CPU 核心，平常说的多核是指在一枚处理器中集成两个或多个完整的计算引擎。核跟 CPU 的关系是：核属于 CPU 的一部分。
2. 寄存器：最靠近 CPU 对存储单元，32 位 CPU 寄存器可存储 4 字节，64 位寄存器可存储 8 字节。寄存器访问速度一般是半个 CPU 时钟周期，属于纳秒级别，
3. L1 缓存：每个 CPU 核心都有，用来缓存数据跟指令，访问空间大小一般在 32 ~ 256KB，访问速度一般是 2 ~ 4 个 CPU 时钟周期。

   cat /sys/devices/system/cpu/cpu0/cache/index0/size # L1 数据缓存

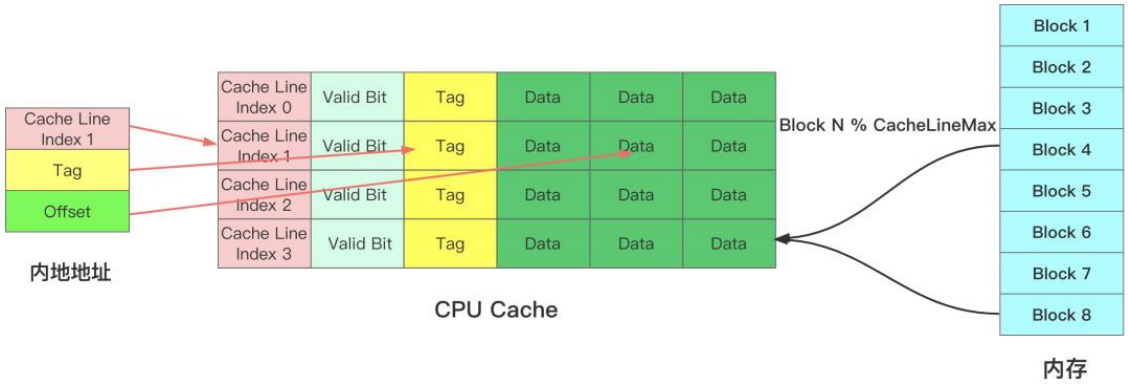
   cat /sys/devices/system/cpu/cpu0/cache/index1/size # L1 指令缓存

4. L2 缓存：每个 CPU 核心都有，访问空间大小在 128KB ~ 2MB，访问速度一般是 10 ~ 20 个 CPU 时钟周期。
   cat /sys/devices/system/cpu/cpu0/cache/index2/size # L2 缓存容量大小
5. L3 缓存：多个 CPU 核心共用，访问空间大小在 2MB ~ 64MB，访问速度一般是 20 ~ 60 个 CPU 时钟周期。
   cat /sys/devices/system/cpu/cpu0/cache/index3/size # L3 缓存容量大小
6. 内存：多个 CPU 共用，现在一般是 4G ~ 512G，访问速度一般是 200 ~ 300 个 CPU 时钟周期。
7. 固体硬盘 SSD：现在台式机主流都会配备，上述的寄存器、缓存、内存都是断电数据立马丢失的，而 SSD 里不会丢失，大小一般是 128G ~ 1T，比内存慢 10 ~ 1000 倍。
8. 机械盘 HDD：很早以前流行的硬盘了，容量可在 512G ~ 8T 不等，访问速度比内存慢 10W 倍不等。
9. 访问数据顺序：CPU 在拿数据处理的时候几乎也是按照上面说得流程来操纵的，只有上面一层找不到才会找下一层。
10. Cache Line : CPU 读取数据时会按照 Cache Line 方式把数据加载到缓存中，每个 Cacheline = 64KB，因为 L1、L2 是每个核独有到可能会触发**伪共享**，就是 所以可能会将数据划分到不同到 CacheLine 中来避免伪共享，比如在 JDK8 新增加的 LongAdder 就涉及到此知识点。

伪共享：缓存系统中是以缓存行（cache line）为单位存储的，当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享。

1. JMM: 数据经过种种分层会导致访问速度在不断提升，同时也带来了各种问题，多个 CPU 同时操作相同数据可能会造成各种 BU 个，需要加锁，这里在 JUC 并发已详细探讨过。

1.3 CPU 访问方式



CPU 访问方式

内存数据映射到 CPU Cache 时通过公式 Block N % CacheLineMax 决定内存 Block 数据放到那个 CPU Cache Line 里。CPU Cache 主要有 4 部分组成。

1. **Cache Line Index**：CPU 缓存读取数据时不是按照字节来读取的，而是按照 CacheLine 方式存储跟读取数据的。
2. **Valid Bit**：有效位标志符，值为 0 时表示无论 CPU Line 中是否有数据，CPU 都会直接访问内存，重新加载数据。
3. **Tag**：组标记，用来标记内存中不同 BLock 映射到相同 CacheLine，用 Tag 来区分不同的内存 Block。
4. **Data**：真实到内存数据信息。

CPU 真实访问内存数据时只需要指定三个部分即可。

1. **Cache Line Index**：要访问到 Cache Line 位置。
2. **Tag**：表示用那个数据块。
3. **Offset**：CPU 从 CPU Cache 读取数据时不是直接读取 Cache Line 整个数据块，而是读取 CPU 所需的数据片段，称为 Word。如何找到 Word 就需要个偏移量 Offset。

1.4 CPU 访问速度

位置	CPU时钟周期
寄存器	0.5
L1	2～4
L2	10～20
L3	20～60
内存	200～300
固态硬盘SSD	比内存慢10～1000倍
机械硬盘HDD	比内存慢10W倍

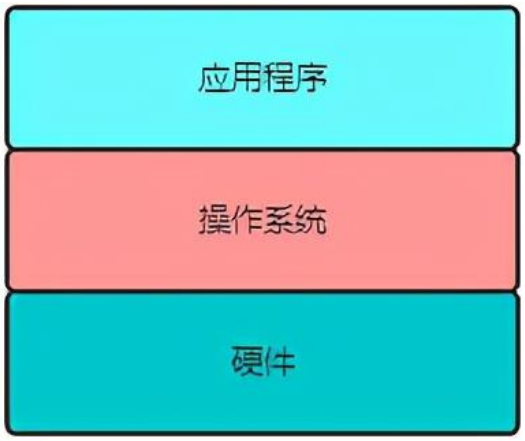
访问耗时对比

如上图所示，CPU 访问速度是逐步变慢，所以 CPU 访问数据时需尽量在距离 CPU 近的高速缓存区访问，根据摩尔定律 CPU 访问速度每 18 个月就会翻倍，而内存的访问每 18 个月也就增长 10% 左右，导致的结果就是 CPU 跟内存访问性能差距逐步变大，那如何尽可能提高 CPU 缓存命中率呢？

1. 数据缓存：遍历数据时候按照内存布局顺序访问，因为 CPU Cache 是根据 Cache Line 批量操作数据的，所以你顺序读取数据会提速，道理跟磁盘顺序写一样。

1. 指令缓存：尽可能的提供有规律的条件分支语句，让 CPU 的分支预测器发挥作用，进一步提高执行的效率，因为 CPU 是自带分支预测器，自动提前将可能需要的指令放到指令缓存区。
2. 线程绑定到 CPU：一个任务 A 在前一个时间片用 CPU 核心 1 运行，后一个时间片用 CPU 核心 2 运行，这样缓存 L1、L2 就浪费了。因此操作系统提供了将进程或者线程绑定到某一颗 CPU 上运行的能力。如 Linux 上提供了 sched_setaffinity 方法实现这一功能，其他操作系统也有类似功能的 API 可用。当多线程同时执行密集计算，且 CPU 缓存命中率很高时，如果将每个线程分别绑定在不同的 CPU 核心上，性能便会获得非常可观的提升。

1.5 操作系统

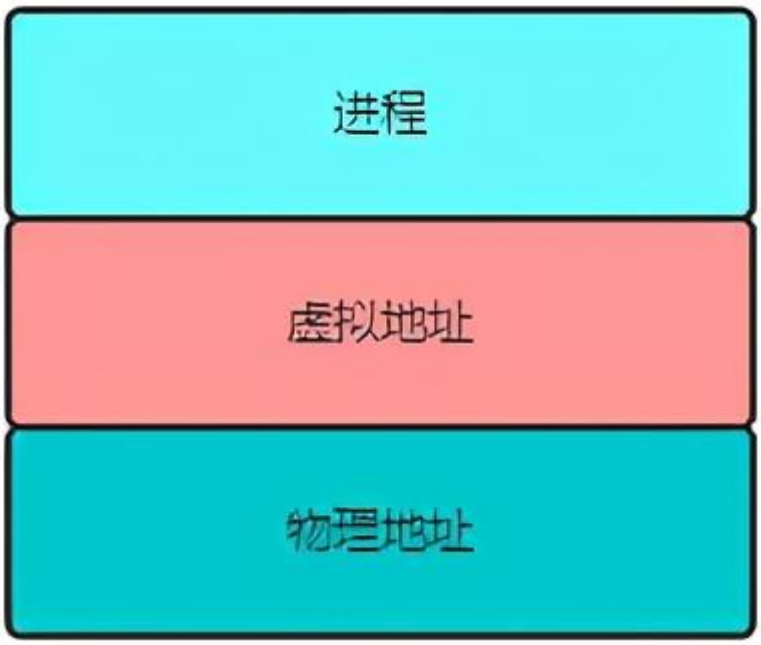


计算机结构

有了冯诺伊曼计算机体系后，电脑想要为用户提供便捷的服务还需要安装个操作系统 Operation System，操作系统是覆盖在硬件上的一层特殊软件，它管理计算机的硬件和软件资源，为其他应用程序提供大量服务。可以理解为操作系统是日常应用程序跟硬件之间的接口。日常你经常在用 Windows/Linux 系统，操作系统给我们提供了超级大的便利，但是你了解操作系统么？操作系统是如何进行**内存管理**、**进程管理**、**文件管理**、**输入输出管理**的呢？

2 内存管理

你的电脑是 32 位操作系统，那可支持的最大内存就是 4G，你有没有好奇为什么可以同时运行 2 个以上的 2G 内存的程序。应用程序不是直接使用的物理地址，操作系统为每个运行的进程分配了一套虚拟地址，每个进程都有自己的虚拟内存地址，进程是无法直接进行物理内存地址的访问的。至于虚拟地址跟物理地址的映射，进程是感知不到的！操作系统自身会提供一套机制将不同进程的虚拟地址和不同内存的物理地址进行映射。

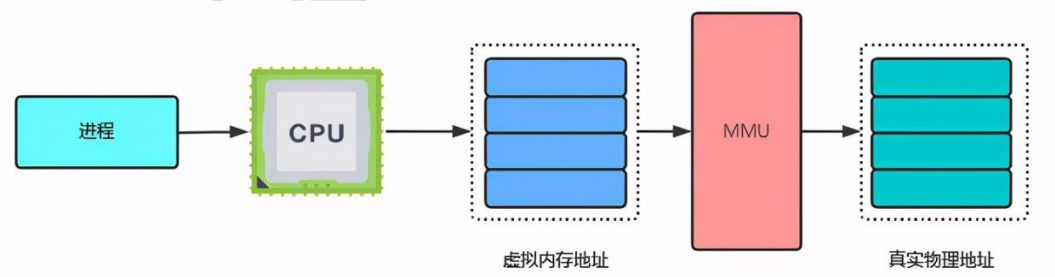


虚拟内存

2.1 MMU

Memory Management Unit 内存管理单元是一种负责处理 CPU 内存访问请求的计算机硬件。它的功能包括**虚拟地址到物理地址的转换、内存保护、中央处理器高速缓存的控制**。现代 CPU 基本上都选择了使用 MMU。

当进程持有虚拟内存地址的时候，CPU 执行该进程时会操作虚拟内存，而 MMU 会自动的将虚拟内存的操作映射到物理内存上。



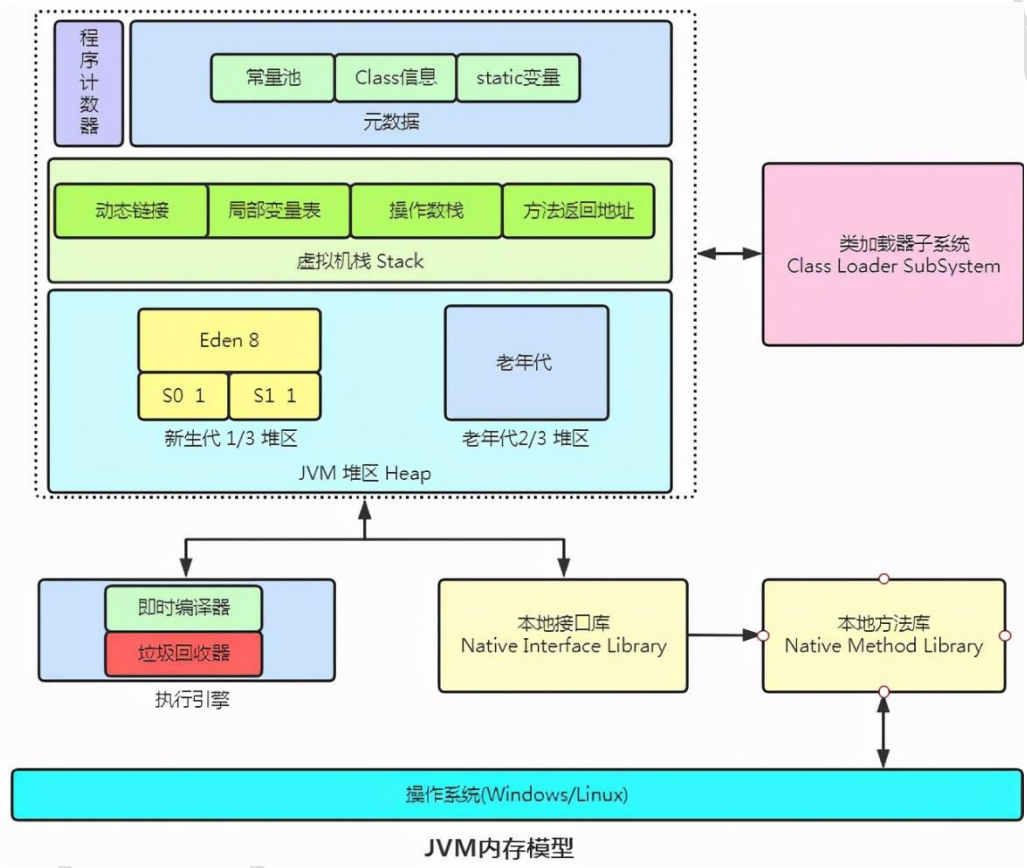
MMU

这里提一下，Java 操作的时候你看到的地址是 JVM 地址，不是真正的物理地址。

2.2 内存管理方式

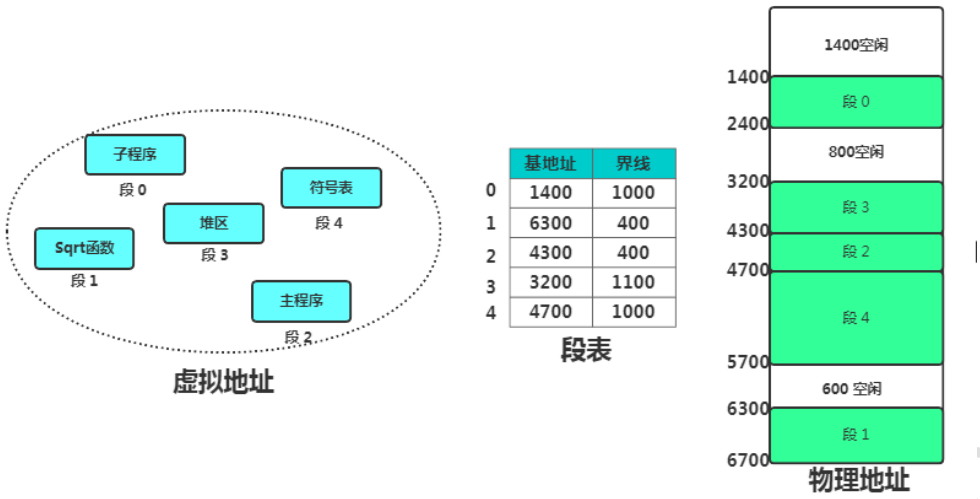
操作系统主要采用内存分段和内存分页来管理虚拟地址与物理地址之间的关系，其中分段是很早前的方法了，现在大部分用的是分页，不过分页也不是完全的分页，是在分段的基础上再分页。

2.2.1 内存分段



JVM 内存模型

我们以上图的 JVM 内存模型举例，程序员会认为我们的代码是由代码段、数据段、栈段、堆段组成。不同的段是有不同的属性的，用户并不关心这些元素所在内存的位置，而分段就是支持这种用户视图的内存管理方案。逻辑地址空间是由一组段构成。每个段都有名称和长度。地址指定了段名称和段内偏移。因此用户段编号和段偏移来指定不同属性的地址。而虚拟内存地址跟物理内存地址中间是通过段表进行映射的，口说无凭，看图吧。



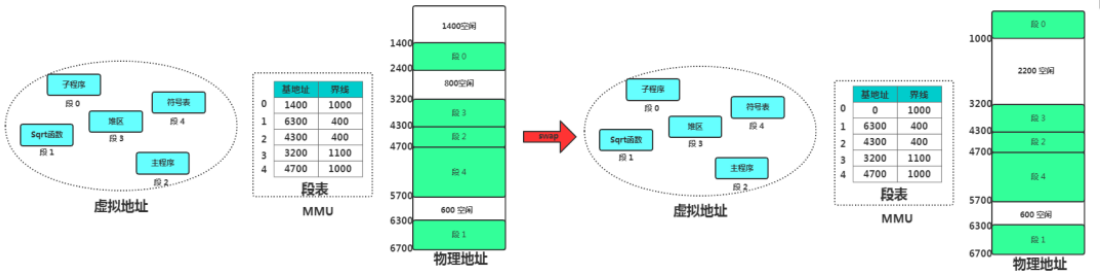
内存分段管理

如上虚拟地址有 5 个段，各段按如图所示来存储。每个段都在段表中有一个条目，它包括段在物理内存内的开始的基地址和该段的界限长度。例如段 2 为 400 字节长，开始于位置 4300。因此对段 2 字节 53 的引用映射成位置 4300 + 53 = 4353。对段 3 字节 852 的引用映射成位置 3200 + 852 = 4052。

分段映射很简单，但是会导致内存碎片跟内存交互效率低。这里先普及下在内存管理中主要有内部内存碎片跟外部内存碎片。

1. **内部碎片**：已经被分配出去的的内存空间不经常使用，并且分配出去的内存空间大于请求所需的内存空间。
2. **外部碎片**：指可用空间还没有分配出去，但是可用空间由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。

以上图为例，现在系统空闲是 1400 + 800 + 600 = 2800。那如果有个程序想要连续的使用 2000，内存分段模式下提供不了啊！上述三个是外部内存碎片。当然可以使用系统的 Swap 空间，先把段 0 写入到磁盘，然后再重新给段 0 分配空间。这样可以实现最终可用，可是但凡涉及到磁盘读写就会导致内存交互效率低。



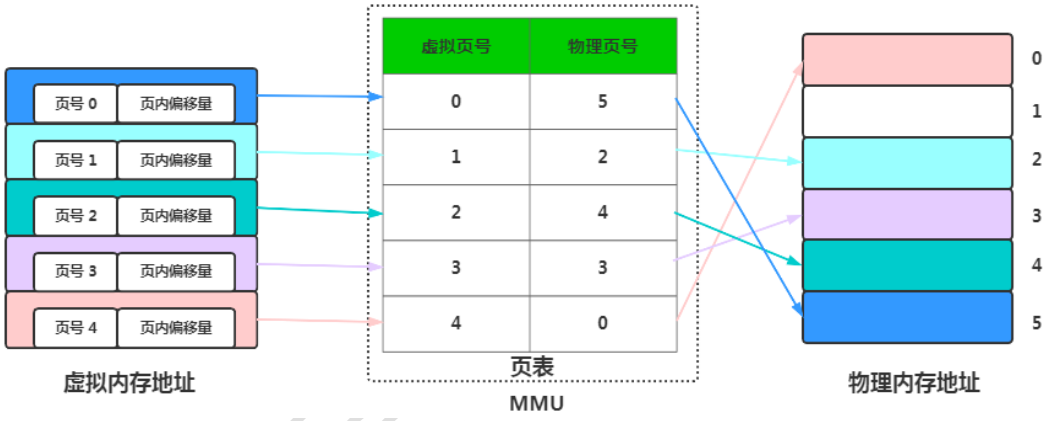
swap 空间利用

2.2.2 内存分页

内存分页，整个虚拟内存和物理内存切成一段段固定尺寸的大小。每个固定大小的尺寸称之为页 Page，在 Linux 系统中 Page = 4KB。然后虚拟内存跟物理内存之间通过页表来实现映射。

采用**内存分页**时内存的释放跟使用都是以页为单位的，也就不会产生内存碎片了。当空间还不够时根据操作系统调度算法，可能将最少用的内存页面 swap-out 换出到磁盘，用时候再 swap-in 换入，尽可能的减少磁盘刷写量，提高内存交互效率。

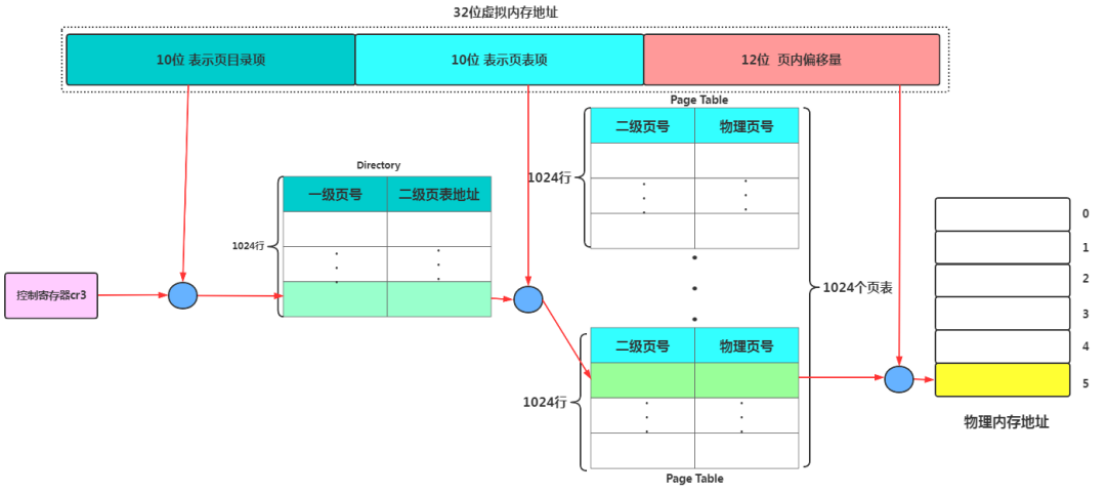
分页模式下虚拟地址主要有页号跟页内偏移量两部分组成。通过页号查询页表找到物理内存地址，然后再配合页内偏移量就找到了真正的物理内存地址。



分页内存寻址

32 位操作系统环境下进程可操作的虚拟地址是 4GB，假设一个虚拟页大小为 4KB，那需要 4GB/4KB = 2^20 个页信息。一行页表记录为 4 字节，2^20 等价于 4MB 页表存储信息。这只是一个进程需要的，如果 10 个、100 个、1000 个呢？仅仅是页表存储都占据超大内存了。

为了解决这个问题就需要用到 多级页表，核心思想就是**局部性**分配。在 32 位的操作系统中将将 4G 空间分为 1024 行页目录项目(4KB)，每个页目录项又对应 1024 行页表项。如下图所示：



32 位系统二级分页

控制寄存器 cr3 中存放了页目录的物理地址，通过 cr3 寄存器可以找到页目录，而 32 位线性地址中的 Directory 部分决定页目录中的目录项，而页目录项中存放了要找的页表的物理基地址，再结合线性地址中的中间 10 位页表项，就可以找到页框的页表项。线性地址中的 Offset 部分占 12 位，因此页框的物理地址 + 线性地址 Offset 部分 = 页框中的任何一个字节。

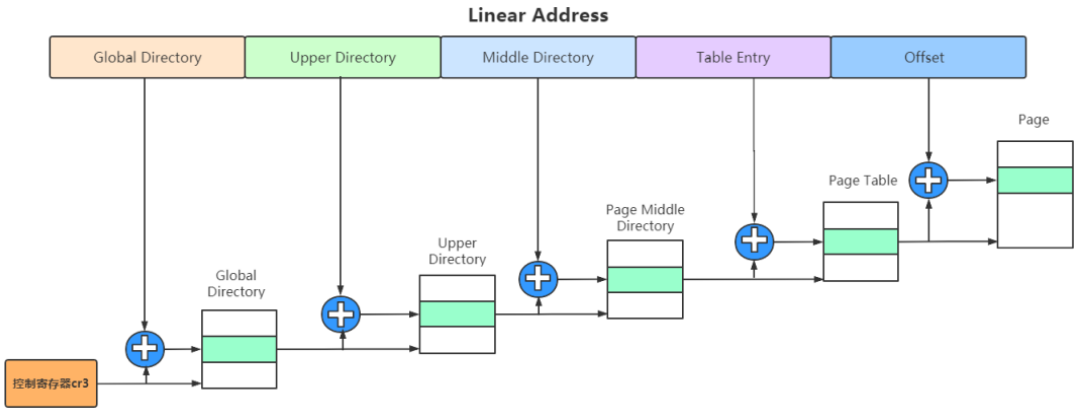
分页后一级页就等价于 4G 虚拟地址空间，并且如果一级页表中那些地址没有就不需要再创建二级页表了！核心思想就是按需创建，当系统给每个进程分配 4G 空间，进程不可能占据全部内存的，如果一级目录页只有 10%用到了，此时页表空间 = 一级页表 4KB + 0.1 * 4MB 。这比单独的每个进程占据 4M 好用多了！

多层分页的弊端就是访问时间的**增加**。

1. 使用页表时读取内存中一页内容需要 2 次访问内存，访问页表项 + 并读取的一页数据。
2. 使用二级页表的话需要三次访问，访问页目录项 + 访问页表项 + 访问并读取的一页数据。访存次数的增加也就意味着访问数据所花费的总时间增加。

而对于 64 位系统，二级分页就无法满足了，Linux 从 2.6.11 开始采用四级分页模型。

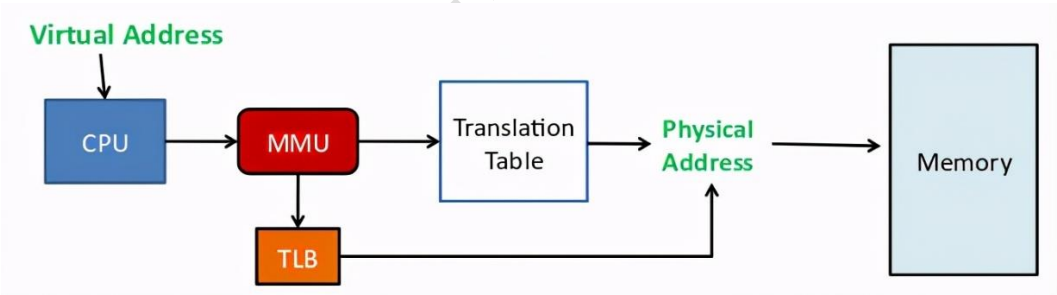
1. Page Global Directory 全局页目录项
2. Page Upper Directory 上层页目录项
3. Page Middle Directory 中间页目录项
4. Page Table Entry 页表项
5. Offset 偏移量。



64 位寻址

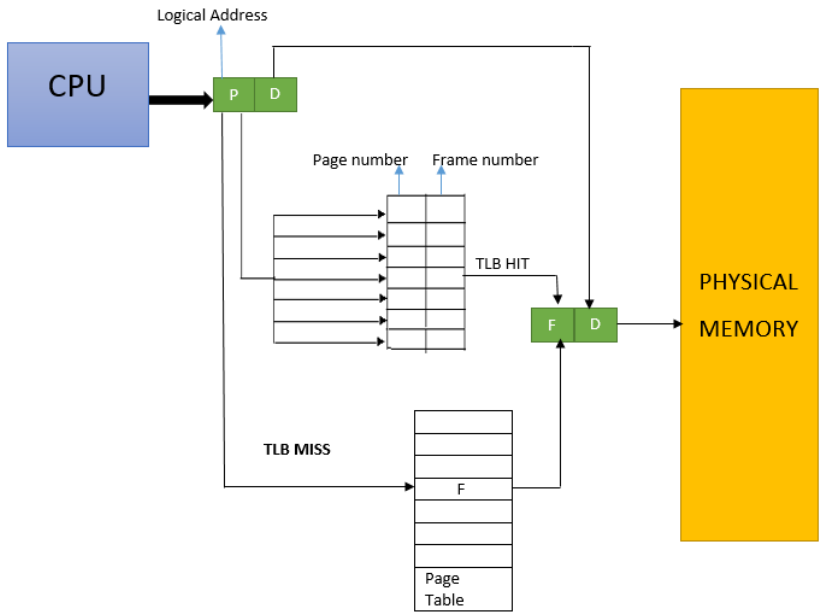
2.2.2 TLB

Translation Lookaside Buffer 可翻译为地址转换后援缓冲器，简称为快表，属于 CPU 内部的一个模块，TLB 是 MMU 的一部分，实质是 cache，它所缓存的是最近使用的数据的页表项（虚拟地址到物理地址的映射）。他的出现是为了加快访问数据（内存）的速度，减少重复的页表查找。当然它不是必须要有的，但有它，速度就更快。



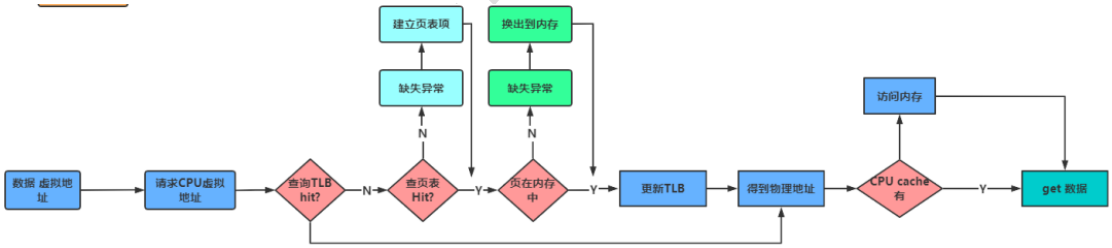
TLB

TLB 很小，因此缓存的东西也不多。主要缓存最近使用的数据的数据映射。TLB 结构如下图：



TLB 查询

如果一个需要访问内存中的一个数据，给定这个数据的虚拟地址，查询 TLB，发现有 hit，直接得到物理地址，在内存根据物理地址取数据。如果 TLB 没有这个虚拟地址 miss，那么只能费力的通过页表来查找了。日常 CPU 读取一个数据的流程如下：



CPU 读取数据流程图

当进程地址空间进行了上下文切换时，比如现在是进程 1 运行，TLB 中放的是进程 1 的相关数据的地址，突然切换到进程 2，TLB 中原有的数据不是进程 2 相关的，此时 TLB 刷新数据有两种办法。

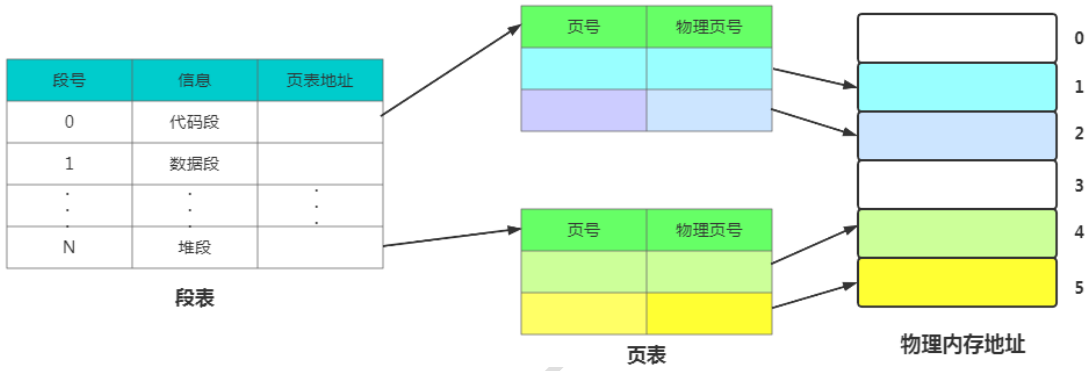
1. **全部刷新**：很简单，但花销大，很多不必刷新的数据也进行刷新，增加了无畏的花销。
2. **部分刷新**：根据标志位，刷新需要刷新的数据，保留不需要刷新的数据。

2.2.3 段页式管理

内存分段跟内存分页不是对立的，这俩可以组合起来在同一个系统中使用的，那么组合起来后通常称为段页式内存管理。段页式内存管理实现的方式：

1. 先对数据不同划分出不同的段，也就是前面说的分段机制。
2. 然后再把每一个段进行分页操作，也就是前面说的分页机制。
3. 此时 地址结构 = 段号 + 段内页号 + 页内位移。

每一个进程有一张段表，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。



段页式管理

同时我们经常看到两个专业词逻辑地址跟线性地址。

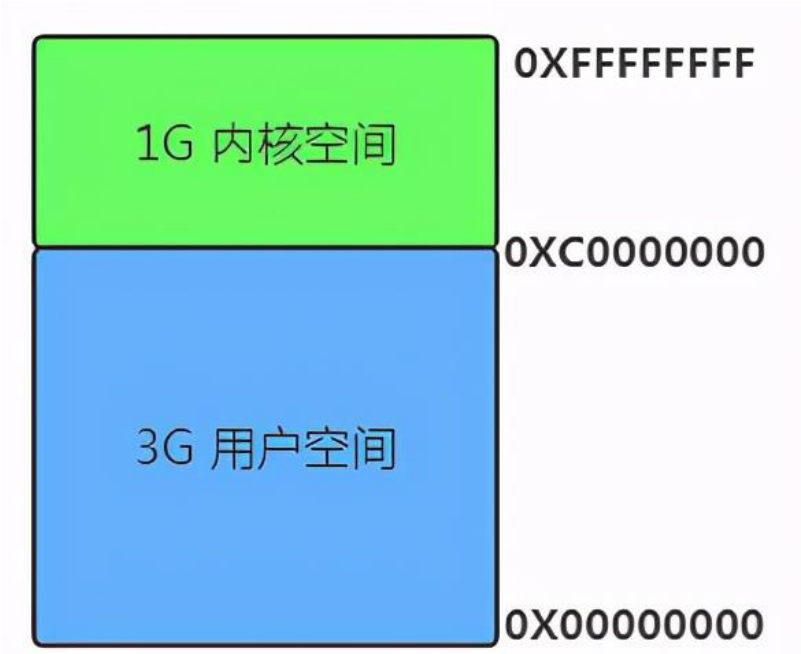
1. 逻辑地址：指的是没被段式内存管理映射的地址。
2. 线性地址：通过段式内存管理映射且页式内存管理转换前的地址，俗称虚拟地址。

目前 Intel X86 CPU 采用的是内存分段 + 内存分页的管理方式，其中分页的意思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。

X86 内存管理方式

2.2.4 Linux 内存管理

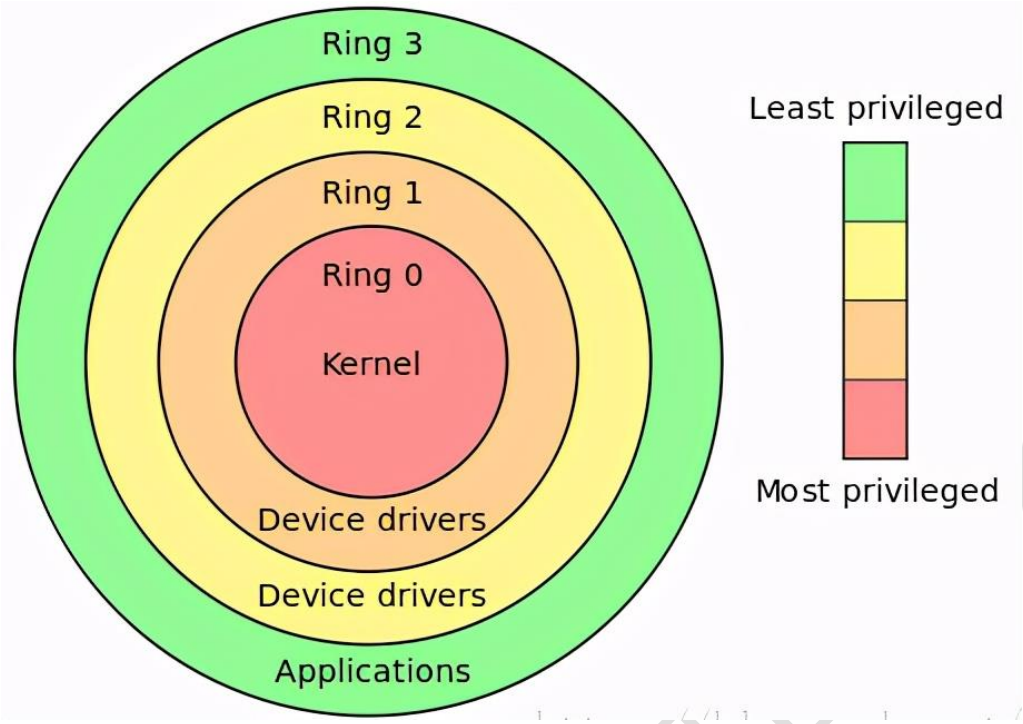
先说结论：Linux 系统基于 X86 CPU 而做的操作系统，所以也是用的段页式内存管理方式。



我们知道 32 位的操作系统可寻址范围是 4G，操作系统会将 4G 的可访问内存空间分为**用户空间**跟**内核空间**。

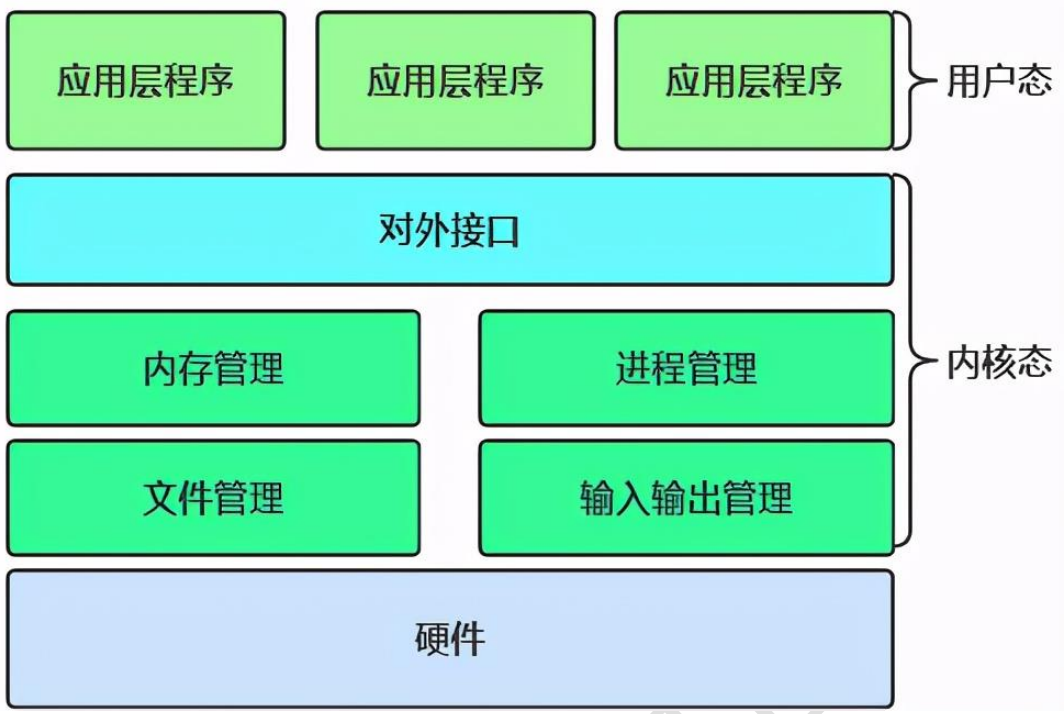
1. 内核空间：操作系统内核访问的区域，独立于普通的应用程序，是受保护的内存空间。内核态下 CPU 可执行任何指令，可自由访问任何有效地址。
2. 用户空间：普通应用程序可访问的内存区域。被执行代码会受到 CPU 众多限制，进程只能访问映射其地址空间的页表项中规定的在用户态下可访问页面的虚拟地址。

那为啥要搞俩空间呢?现在嵌入式环境跟以前的 WIN98 系统是没有区分俩空间的，须知俩空间是 CPU 分的，而操作系统是在上面运行的，单一用户、单一任务服务的操作系统，是没有分所谓用户态和内核态的必要。用户态和内核态是因为有多用户，多任务的需求，然后在 CPU 硬件厂商配合之后，产生的一个操作系统解决多用户多任务需求的方案。方案就是**限制**，通过硬件手段（也只能硬件手段才能做到），限制某些代码，使其无法控制整个物理硬件，进而使各个不同用户，不同任务的代码，无权修改整个物理硬件，再进而保护操作系统的核心底层代码和其他用户的数据不被无意或者有意地破坏和盗取。



后来研究者根据 CPU 的运行级别，分成了 Ring0~Ring3 四个级别。Ring0 是最高级别，Ring1 次之，Rng2 更次之，拿 Linux+x86 来说， 操作系统内核的代码运行在最高运行级别 Ring0 上，可以使用特权指令，控制中断、修改页表、访问设备等。 应用程序的代码运行在最低运行级别上 Ring3 上，不能做受控操作，只能访问用户被分配的空间。如果要做访问磁盘跟写文件等操作，那就要通过执行系统调用函数，执行系统调用的时候，CPU 的运行级别会发生从 Ring3 到 Ring0 的切换，并跳转到系统调用对应的内核代码位置执行，这样内核就为你完成了设备访问，完成之后再从 Ring0 返回 Ring3。这个过程也称作**用户态和内核态的切换**。

用户态想要使用计算机设备或 IO 需通过系统调用完成 sys call，系统调用就是让内核来做这些操作。而系统调用是影响整个当前进程上下文的，ＣＰＵ提供了个软中断来是实现保护线程，获取系统调用号跟参数，交给内核对应系统调用函数执行。



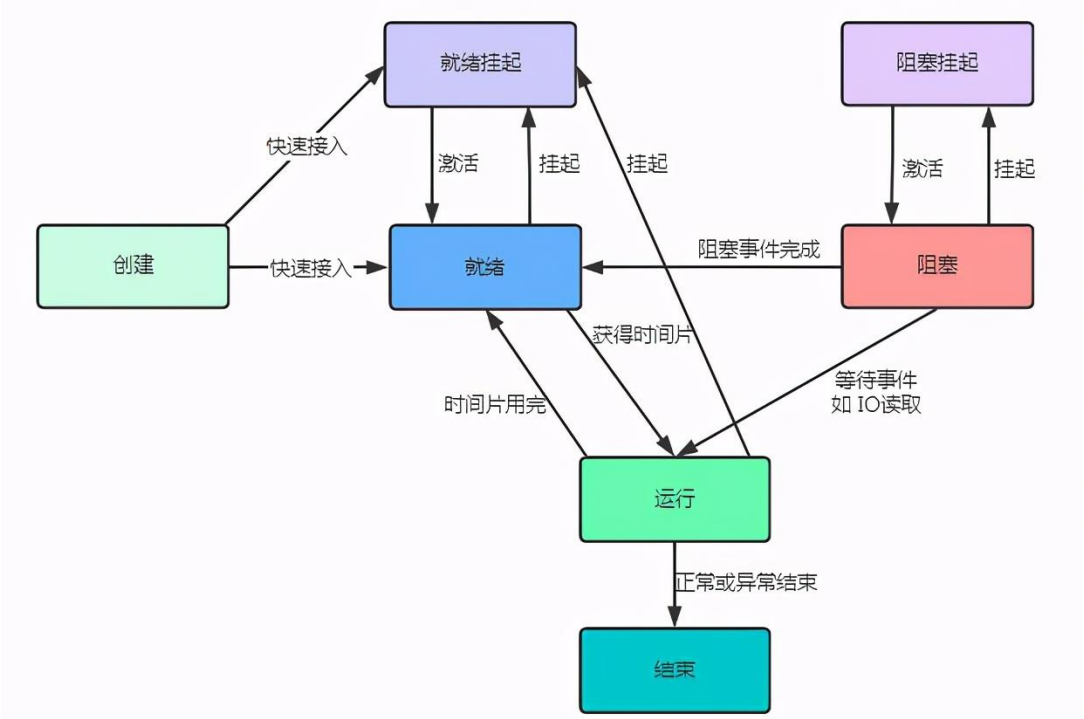
Linux 系统结构

可以看到每个应用程序都各自有独立的虚拟内存地址，但每个虚拟内存中对应的内核地址其实是相同的一大块，这样当进程切换到内核态后可以很方便地访问内核空间内存。比如 Java 代码创建线程 new Thread 调用 start 方法后跟踪 JVM 源码你会发现是调用 pthread_create 来创建线程的，这就涉及到了用户态到内核态的切换。

3 进程管理

3.1 进程基础知识

进程是程序的一次执行，是一个程序及其数据在机器上顺序执行时所发生的活动，是具有独立功能的程序在一个数据集合上的一次运行过程，是系统进行资源分配和调度的一个基本单位。进程的调度状态如下：



状态变化图

重点说下挂起跟阻塞：

1. 阻塞一般是当系统执行 IO 操作时，此时进程进入阻塞状态，等待某个事件的返回。
2. 挂起是指进程没有占有物理内存，被写到磁盘上了。这时进程状态是挂起状态。

阻塞挂起：进程被写入硬盘并等待某个事件的出现。

就绪挂起：进程被写入硬盘，进入内存可直接进入就绪状态。

3.2 PCB

为了描述跟控制进程的运行，系统为每个进程定义了一个数据结构——进程控制块 Process Control Block，它是进程实体的一部分，是操作系统中最重要的记录型数据结构。

PCB 的作用是**使一个在多道程序环境下不能独立运行的程序，成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程** ：

1. 作为独立运行基本单位的标志
2. 实现间断性的运行方式
3. 提供进程管理所需要的信息
4. 提供进程调度所需要的信息

5. 实现与其他进程的同步与通信

3.2.1 PCB 信息

PCB 为实现上述功能，内部包含众多信息：

1. **进程标识符**：用于唯一地标识一个进程，一个进程通常有两种标识符：

内部进程标识符：标识各个进程，每个进程都有一个并且唯一的标识符，设置内部标识符主要是为了方便系统使用。

外部进程标识符：它由创建者提供，可设置用户标识，以指示拥有该进程的用户。往往是由用户进程在访问该进程时使用。一般为了描述进程的家族关系，还应设置父进程标识及子进程标识。

1. **处理机状态**：由各种寄存器组成。包含许多信息都放在寄存器中，方便程序 restart。

通用寄存器、指令计数器、程序状态字 PSW、用户栈指针等信息。

1. **进程调度信息**

进程状态：指明进程的当前状态，作为进程调度和对换时的依据。

进程优先级：用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机

进程调度所需的其它信息：与所采用的进程调度算法有关，如进程已等待 CPU 的时间总和、进程已执行的时间总和等。

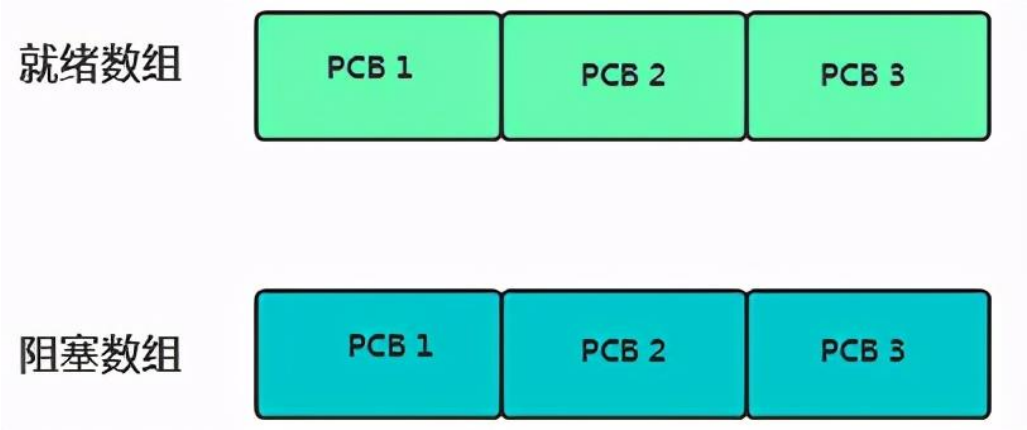
事件：指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

1. **资源清单**

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。

3.2.2 PCB 组织方式

操作系统中有太多 PCB，如何管理是个问题，一般有如下方式。

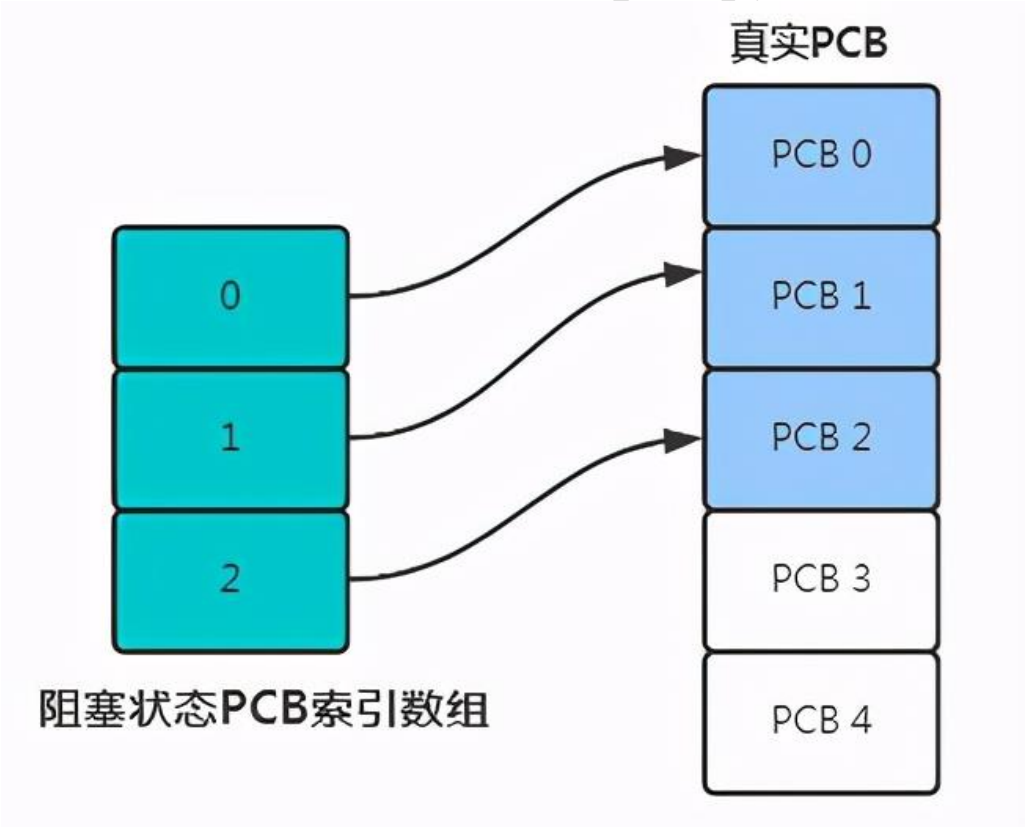


线下数组

1. **线性方式**:

将系统所有 PCB 都组织在一张线性表中，将该表首地址存在内存的一个专用区域

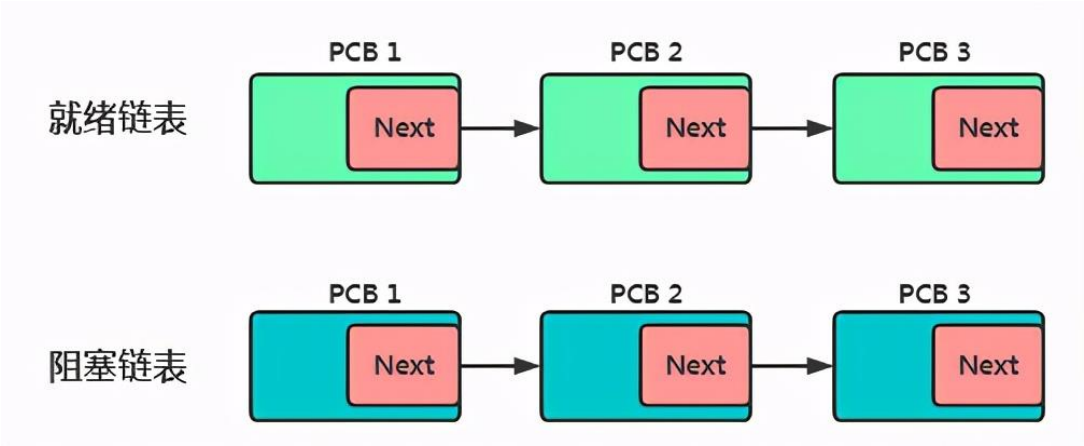
实现简单，开销小，但是每次都需要扫描整张表，适合进程数目不多的系统



索引方式

1. **索引方式**:

将同一状态的进程组织在一个索引表中，索引表项指向相应的 PCB，不同状态对应不同的索引表。



链表方式

1. **链接方式**:

把同一状态的 PCB 链接成一个队列，形成就绪队列、阻塞队列、空白队列等。对其中的就绪队列常按进程优先级的高低排列，优先级高排在队前。

因为进程创建、销毁、调度频繁，所以**一般采用此模式**。

3.3 进程控制

进程控制是进程管理最基本的功能，主要包括创建新进程，终止已完成的进程，将发生异常的进程置于阻塞状态，将进程唤醒等。

3.3.1 进程创建

父进程可创建子进程，父进程终止后子进程也会被终止。子进程可继承父进程所有资源，子进程终止需将自己所继承的资源归还父进程。接下来看下创建的大致流程。

1. 为新进程分配唯一进件标识号，然后创建一个空白 PCB，需注意 PCB 数量是有限的，所以可能会创建失败。
2. 尝试为新进程分配所需资源，如果资源不足进程会进入等待状态。
3. 初始化 PCB，有如下几个操作。

标识信息：将系统分配的标识符和父进程标识符填入新 PCB

处理机状态信息：使程序计数器指向程序入口地址，使栈指针指向栈顶

处理机控制信息：将进程设为就绪/静止状态，通常设为最低优先级

1. 如果进程调度队列能接纳新进程，就将进程插入到就绪队列，等待被调度运行。

3.3.2 进程终止

进程终止情况一般分为正常结束、异常结束、外界干预三种。

1. 正常结束
2. 异常结束

越界错：访问的存储区越出该进程的区域

保护错：试图访问不允许访问的资源，或以不适当的方式访问（写只读）

非法指令：试图执行不存在的指令（可能是程序错误地转移到数据区，数据当成了指令）

特权指令出错：用户进程试图执行一条只允许 OS 执行的指令

运行超时：执行时间超过指定的最大值

等待超时：进程等待某件事超过指定的最大值

算数运算错：试图执行被禁止的运算（被 0 除）

I/O 故障

1. 外界干预

操作员或 OS 干预（死锁）

父进程请求，子进程完成父进程指定的任务时

父进程终止，所有子进程都应该结束

终止过程：

1. 根据被终止进程的标识符，从 PCB 集合中检索出该 PCB，读取进程状态
2. 若处于执行状态则立即终止执行，将 CPU 资源分配给其他进程。
3. 若进程有子孙进程则将其所有子孙进程终止。
4. 全部资源还给父进程或操作系统。
5. 该进程的 PCB 从所在队列/链表中移出。

3.3.3 进程阻塞

意思是该进程执行半路被阻塞，必须由某个事件进程唤醒该进程。常见的就是 IO 读取操作。常见阻塞时机/事件如下：

1. 请求共享资源失败，系统无足够资源分配
2. 等待某种操作完成
3. 新数据尚未到达（相互合作的进程）
4. 等待新任务

阻塞流程：

1. 找到要被阻塞进程标识号对应的 PCB。
2. 将该进程由运行状态转换为阻塞状态。
3. 将该 进程 PCB 插入的阻塞队列中去。

3.3.4 进程唤醒

唤醒 原语 wake up，一般和阻塞成对使用。唤醒过程如下：

1. 从阻塞队列找到所需 PCB。
2. PCB 从阻塞队列溢出，然后变为就绪状态。
3. 从阻塞队列溢出该 PCB 然后插入到就绪状态队列等待被分配 CPU 资源。

3.4 进程调度

进程数一般会大于 CPU 个数，进程状态切换主要由调度程序进行调度。一般情况下 CPU 调度时主要分为抢占式调度跟非抢占式调度。

1. 非抢占式：让进程运行直到结束或阻塞的调度方式， 容易实现，适合专用系统。
2. 抢占式：每个进程获得时间片才可以被 CPU 调度运行， 可防止单一进程长时间独占 CPU 系统开销大。

3.4.1 进程调度原则

1. **CPU 利用率**

CPU 利用率 = 忙碌时间 / 总时间。

调度程序应该尽量让 CPU 始终处于忙碌的状态，这可提高 CPU 的利用率。比如当发生 IO 读取时候，不要傻傻等待，去执行下别的进程。

1. **系统吞吐量**

系统吞吐量 = 总共完成多少个作业 / 总共花费时间。

长作业的进程会占用较长的 CPU 资源导致降低吞吐量，相反短作业的进程会提升系统吞吐量。

1. **周转时间**

周转时间 = 作业完成时间 - 作业提交时间。

平均周转时间 = 各作业周转时间和 / 作业数

带权周转时间 = 作业周转时间 / 作业实际运行时间

平均带权周转时间 = 各作业带权周转时间之和 / 作业数

尽可能使周转时间降低。

1. **等待时间**

指的是进程在等待队列中等待的时间，一般也需要尽可能短。

响应时间
响应时间 = 系统第一次响应时间 - 用户提交时间，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。

3.4.2 调度算法

FCFS 算法

1. First Come First Severd 先来先服务算法，遵循先来后端原则，每次从就绪队列拿等待时间最久的，运行完毕后再拿下一个。
2. 该模式对长作业有利，适用 CPU 繁忙型作业的系统，不适用 I/O 型作业，因为会导致进程 CPU 利用率很低。

SJF 算法

1. Shortest Job First 最短作业优先算法，该算法会优先选择运行所需时间最短的进程执行，可提高吞吐量。
2. 跟 FCFS 正好相反，对长作业很不利。

SRTN 算法

1. Shortest Remaining Time Next 最短剩余时间优先算法，可以认为是 SJF 的抢占式版本，当一个新就绪的进程比当前运行进程具有更短完成时间时，系统抢占当前进程，选择新就绪的进程执行。
2. 有最短的平均周转时间，但不公平，源源不断的短任务到来，可能使长的任务长时间得不到运行。

HRRN 算法

1. Highest Response Ratio Next 最高响应比优先算法，为了平衡前面俩而生，按照响应优先权从高到低依次执行。属于前面俩的折中权衡。
2. 优先权 = (等待时间 + 要求服务时间) / 要求服务时间

RR 算法

1. Round Robin 时间片轮转算法，操作系统设定了个时间片 Quantum，时间片导致每个进程只有在该时间片内才可以运行，这种方式导致每个进程都会均匀的获得执行权。
2. 时间片一般 20ms~50ms，如果太小会导致系统频繁进行上下文切换，太大又可能引起对短的交互请求的响应变差。

HPF 算法

1. Highest Priority First 最高优先级调度算法，从就绪队列中选择最高优先级的进程先执行。
2. 优先级的设置有初始化固定死的那种，也有在代码运转过程中根据等待时间或性能动态调整 这两种思路。
3. 缺点是可能导致低优先级的一直无法被执行。

MFQ 算法

1. Multilevel Feedback Queue 多级反馈队列调度算法 ，可以认为是 RR 算法 跟 HPF 算法 的综合体。
2. 系统会同时存在多个就绪队列，每个队列优先级从高到低排列，同时优先级越高获得是时间片越短。
3. 新进程会先加入到最高优先级队列，如果新进程优先级高于当前在执行的进程，会停止当前进程转而去执行新进程。新进程如果在时间片内没执行完毕需下移到次优先级队列。
   多级反馈队列调度算法

3.5 线程

3.5.1 线程定义

早期操作系统是没有线程概念的，线程是后来加进来的。为啥会有线程呢？那是因为以前在多进程阶段，经常会涉及到进程之间如何通讯，如何共享数据的问题。并且进程关联到 PCB 的生命周期，管理起来开销较大。为了解决这个问题引入了线程。

线程是进程当中的一个执行流程。同一个进程内的多个线程之间可以共享进程的代码段、数据段、打开的文件等资源。同时每个线程又都有一套独立的寄存器和栈来确保线程的控制流是独立的。

进程有个 PCB 来管理，同理操作系统通过 Thread Control Block 线程控制块来实现线程的管控。

3.5.2 线程优缺点

优点

1. 一个进程中可以同时存在 1~N 个线程，这些线程可以并发的执行。
2. 各个线程之间可以共享地址空间和文件等资源。

缺点

1. 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。
2. 多线程编程，让人头大的东西。
3. 线程执行开销小，但不利于资源的隔离管理和保护，而进程正相反。

3.5.3 进程跟线程关联

进程：

1. 是系统进行资源分配和调度的一个独立单位.
2. 是程序的一次执行，每个进程都有自己的地址空间、内存、数据栈及其他辅助记录运行轨迹的数据

线程：

1. 是进程的一个实体，是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位

2. 所有的线程运行在同一个进程中，共享相同的运行资源和环境
3. 线程一般是并发执行的，使得实现了多任务的并行和数据共享。

进程线程区别：

1. 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。
2. 线程的划分尺度小于进程(资源比进程少)，使得多线程程序的并发性高。
3. 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
4. 资源分配给进程，同一进程的所有线程共享该进程的所有资源。
5. CPU 分配资源给进程，但真正在 CPU 上运行的是线程。
6. 线程不能够独立执行，必须依存在进程中。

线程快在哪儿？

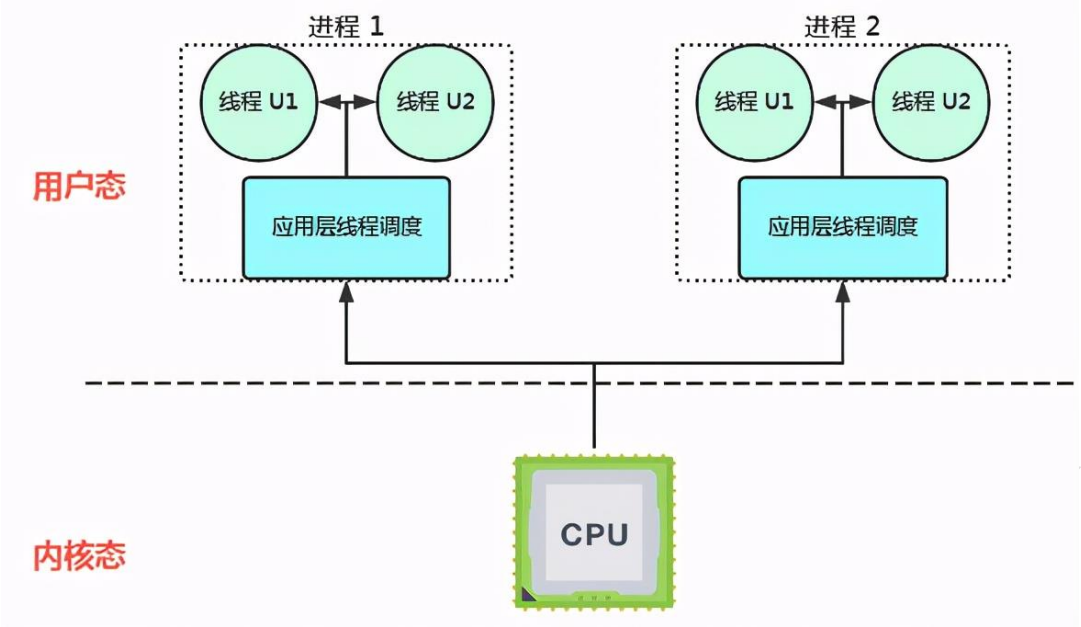
1. 线程创建的时有些资源不需要自己管理，直接从进程拿即可，线程管理寄存器跟栈的生命周期即可。
2. 同进程内多线程共享数据，所以进程数据传输可以用 zero copy 技术，不需要经过内核了。
3. 进程使用一个虚拟内存跟页表，然后多线程共用这些虚拟内存，如果同进程内两个线程进行上下文切换比进程提速很多。

3.5.4 线程实现

在前面的内存管理中说到了内核态跟用户态。相对应的线程的创建也分为用户态线程跟内核态线程。

3.5.4.1 用户态线程

在用户空间实现的线程，由用户态的线程库来完成线程的管理。操作系统按进程维度进行调度，**当线程在用户态创建时应用程序在用户空间内要实现线程的创建、维护和调度。操作系统对线程的存在一无所知！**操作系统只能看到进程看不到线程。所有的线程都是在用户空间实现。在操作系统看来，每一个进程只有一个线程。



用户态线程

好处：

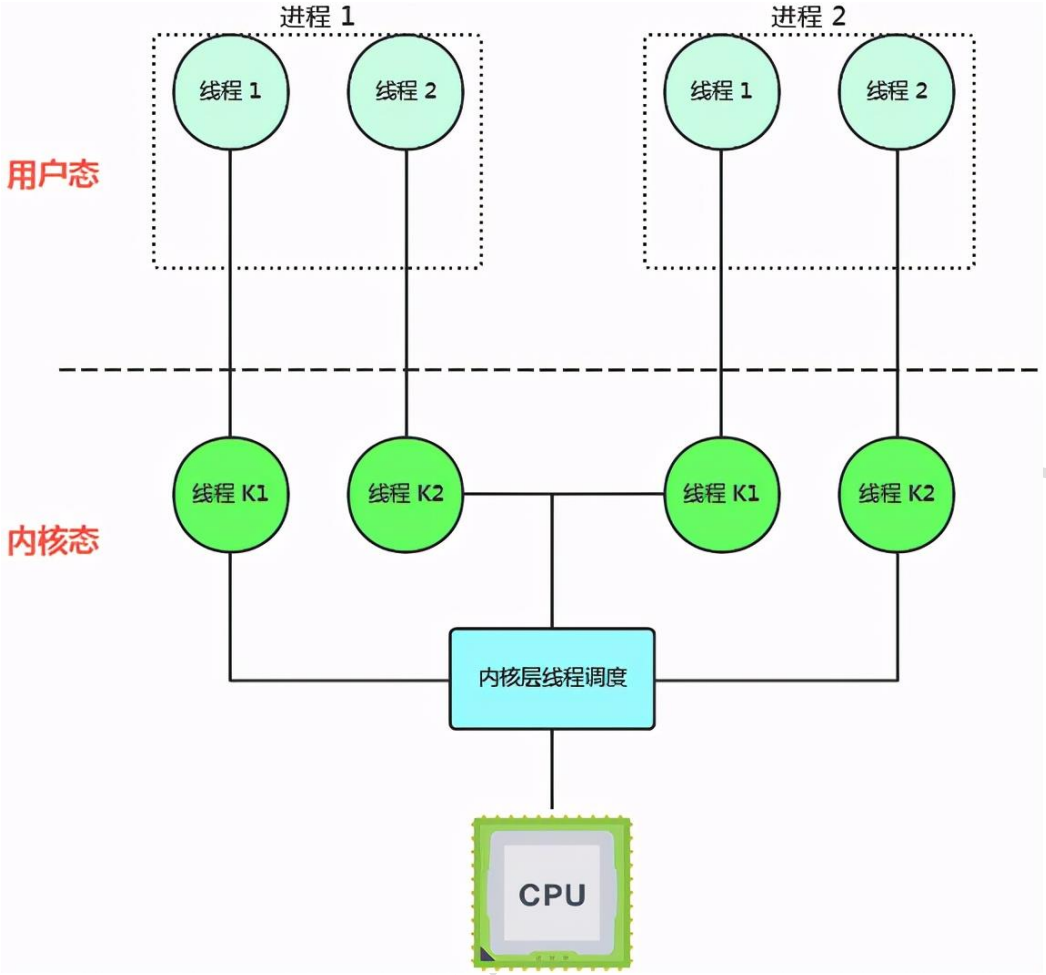
1. 及时操作系统不支持线程模式也可以通过用户层库函数来支持线程模式，TCB 由用户级线程库函数来维护。
2. 使用库函数模式实现线程可以避免用户态到内核态的切换。

坏处：

1. CPU 不知道线程存在，CPU 的时间片切换是以进程为维度的，某个线程因为 IO 等操作导致线程阻塞，操作系统会阻塞整个进程，即使这个进程中其它线程还在工作。
2. 用户态线程没法打断正在运行中的线程，除非线程主动交出 CPU 使用权。

3.5.4.2 内核态线程

在内核中实现的线程，是由内核管理的线程，线程对应的 TCB 在操作系统里，这样线程的创建、终止和管理都是由操作系统负责。内线程模式下一个用户线程对应一个内核线程。



内核态线程

注意：Linux 中的 JVM 从 1.2 版以后是基于 pthread 实现的，所以现在 Java 中线程的本质就是操作系统中的线程。

优点：

1. 一个进程中某个线程阻塞不会影响其他内核线程运行。
2. 用户态模式一个时间片分给多个线程，内核态模式直接分配给线程的时间片增加。

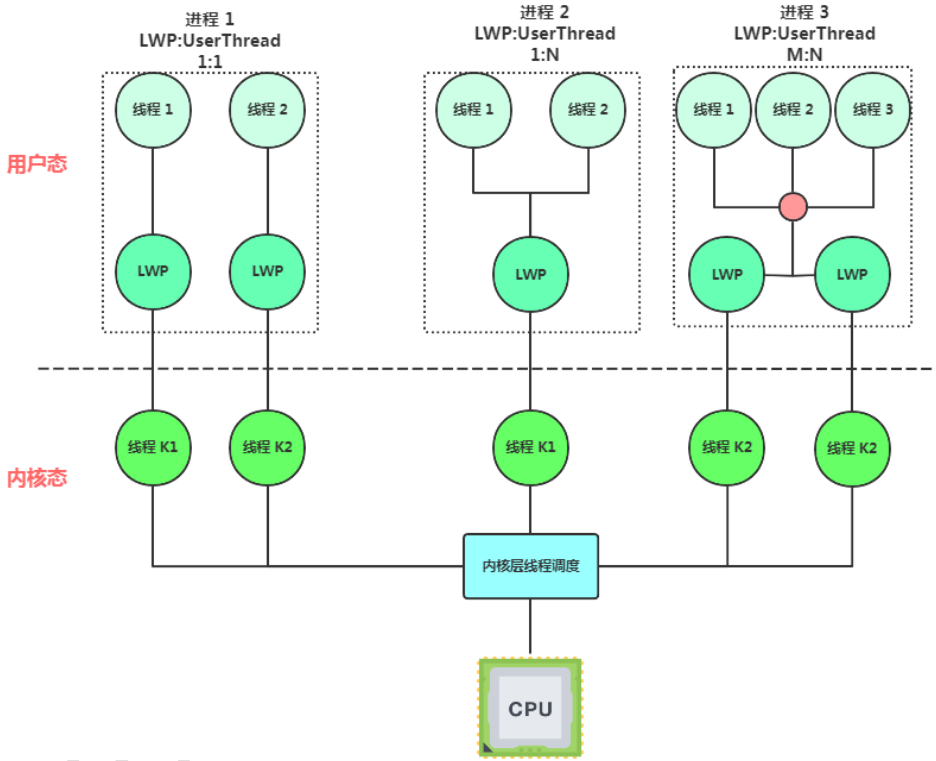
缺点：

1. 内核级线程调度开销较大。调度内核线程的代价可能和调度进程差不多昂贵，代价要比用户级线程大很多。一个线程默认栈=1M，线程多了会导致内存消耗很大。
2. 线程表是存放在操作系统固定的表格空间或者堆栈空间里，所以内核级线程的数量是有限的。

3.4.4.3 轻量级进程

最初的进程定义都包含程序、资源及其执行三部分，其中程序通常指代码，资源在操作系统层面上通常包括内存资源、IO 资源、信号处理等部分，而程序的执行通常理解为执行上下文，包括对 CPU 的占用，后来发展为线程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。

Light-weight process **轻量级进程是内核支持的用户线程**，它是基于内核线程的高级抽象，系统只有先支持内核线程才能有 LWP。一个进程可有 1~N 个 LWP，每个 LWP 是跟内核线程一对一映射的，也就是 LWP 都是由一个内核线程支持。



LWP 模式

轻量级进程本质还是进程，只是跟普通进程相比 LWP 跟其他进程共享大部分逻辑地址空间跟系统资源，LWP 轻量体现在它只有一个最小的执行上下文和调度程序所需的统计信息。他是进程的执行部分，只带有执行相关的信息。

Linux 特性：

1. Linux 中没有真正的线程，因为 Linux 并没有为线程准备特定的数据结构。在内核看来只有进程而没有线程，在调度时也是当做进程来调度。Linux 所谓的线程其实是与其他进程共享资源的进程。但 windows 中确实有线程。
2. Linux 中没有的线程，线程是由进程来模拟实现的。
3. 所以在 Linux 中在 CPU 角度看，进程被称作轻量级进程 LWP。

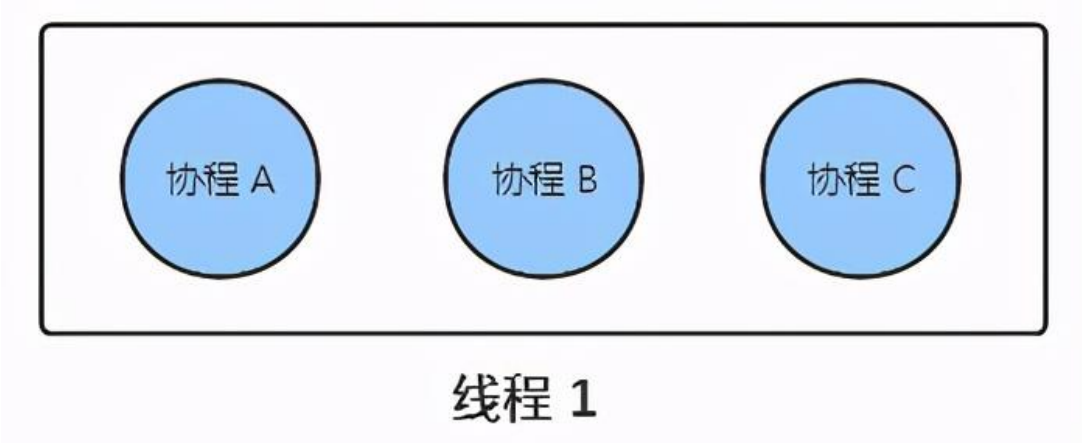
3.5.5 协程

3.5.5.1 协程定义

大多数 web 服务跟互联网服务本质上大部分都是 IO 密集型服务，IO 密集型服务的瓶颈不在 CPU 处理速度，而在于尽可能快速的完成高并发、多连接下的数据读写。以前有两种解决方案：

1. 多进程：存在频繁调度切换问题，同时还会存在每个进程资源不共享的问题，需要额外引入进程间通信机制来解决。
2. 多线程：高并发场景的大量 IO 等待会导致多线程被频繁挂起和切换，非常消耗系统资源，同时多线程访问共享资源存在竞争问题。

此时协程出现了，协程 Coroutines 是一种比线程更加轻量级的微线程。类比一个进程可以拥有多个线程，一个线程也可以拥有多个协程。可以简单的把协程理解成子程序调用，每个子程序都可以在一个单独的协程内执行。



协程

协程运行在线程之上，当一个协程执行完成后，可以选择主动让出，让另一个协程运行在当前线程之上。**协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程**，而且协程的切换在用户态完成，切换的代价比线程从用户态到内核态的代价小很多，一般在 Python、Go 中会涉及到协程的知识，尤其是现在高性能的脚本 Go。

3.5.5.2 协程注意事项

协程运行在线程之上，并且协程调用了一个阻塞 IO 操作，此时操作系统并不知道协程的存在，它只知道线程，因此在协程调用阻塞 IO 操作时，操作系统会让线程进入阻塞状态，当前的协程和其它绑定在该线程之上的协程都会陷入阻塞而得不到调度。

因此在协程中不能调用导致线程阻塞的操作，比如打印、读取文件、Socket 接口等。协程只有和异步 IO 结合起来才能发挥最大的威力。并且**协程只有在 IO 密集型的任务中才会发挥作用**。

3.6 进程通信

进程的用户地址空间是相互独立的，不可以互相访问，但内核空间是进程都共享的，所以进程之间要通信必须通过内核。**进程间通信主要通过管道、消息队列、共享内存、信号量、信号、Socket 编程**。

3.6.1 管道

管道主要分为匿名管道跟命名管道两种，可以实现数据的单向流动性。**使用起来很简单，但是管道这种通信方式效率低，不适合进程间频繁地交换数据**。

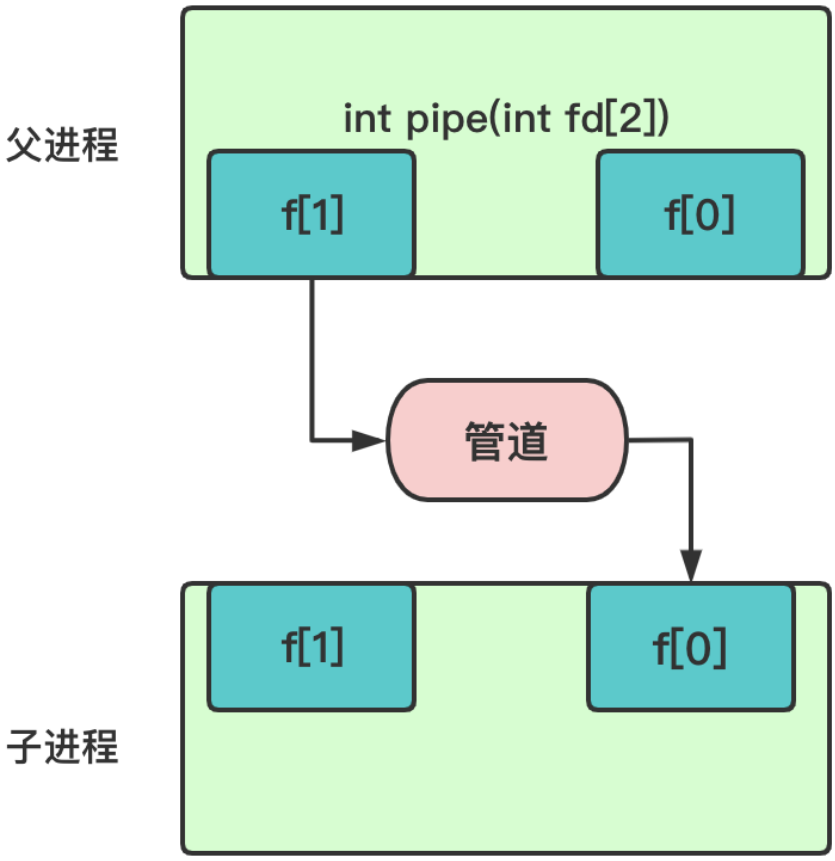
匿名管道：

1. 日常 Linux 系统中的|就是匿名管道。指令的前一个输入是后一个指令的输出。

命名管道：

1. 一般通过 mkfifo SoWhatPipe 创建管道。通过 echo "sw" > SoWhatPipe 跟 cat < SoWhatPipe 实现输入跟输出。

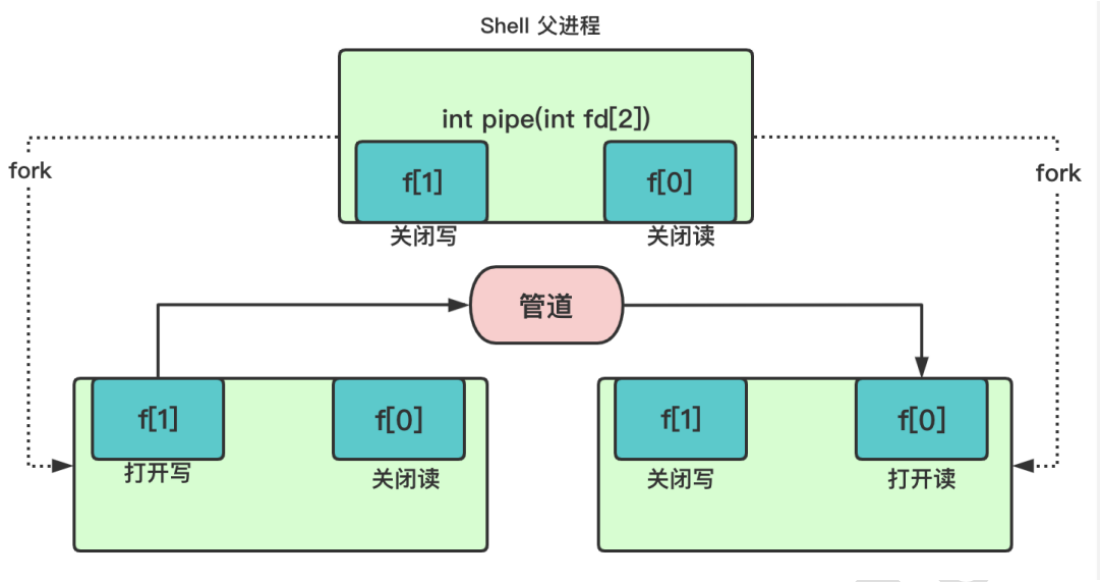
匿名管道的实现依赖 int pipe(int fd[2])函数，其中 fd[0]是读取断描述符，fd[1]是管道写入端描述符。它的本质就是在内核中创建个属于内存的缓存，从一端输入无格式数据一端输出无格式数据，需注意管道传输大小是有限的。



管道通信底层

匿名管道的通信范围是存在父子关系的进程。由于管道没有实体，也就是没有管道文件，不会涉及到文件系统。只能通过 fork 子进程来复制父进程 fd 文件描述符，父子进程通过共用特殊的管道文件实现跨进程通信，并且因为管道只能一端写入，另一端读出，所以通常父子进程遵从如下要求：

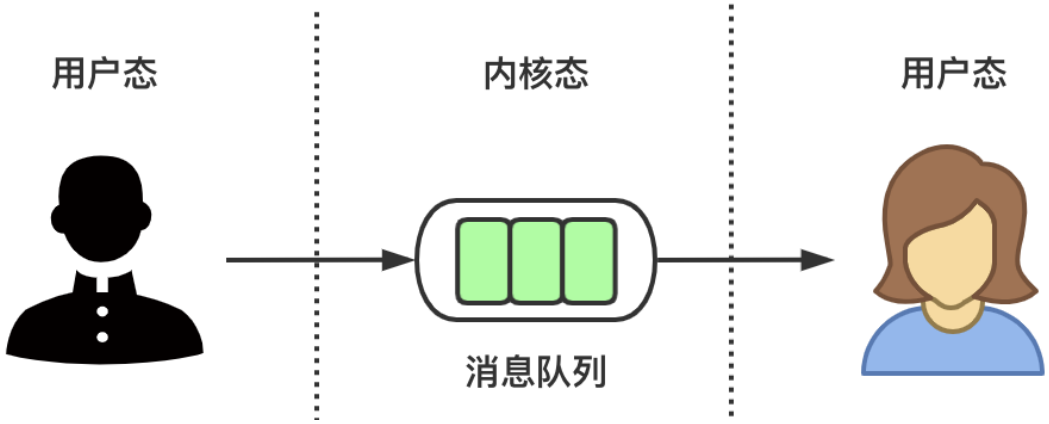
1. 父进程关闭读取的 fd[0]，只保留写入的 fd[1]。
2. 子进程关闭写入的 fd[1]，只保留读取的 fd[0]。



shell 管道通信

需注意 Shell 执行匿名管道 a | b 其实是通过 Shell 父进程 fork 出了两个子进程来实现通信的，而 ab 之间是不存在父子进程关系的。而命名管道是可以直接在不想关进程间通信的，因为有管道文件。

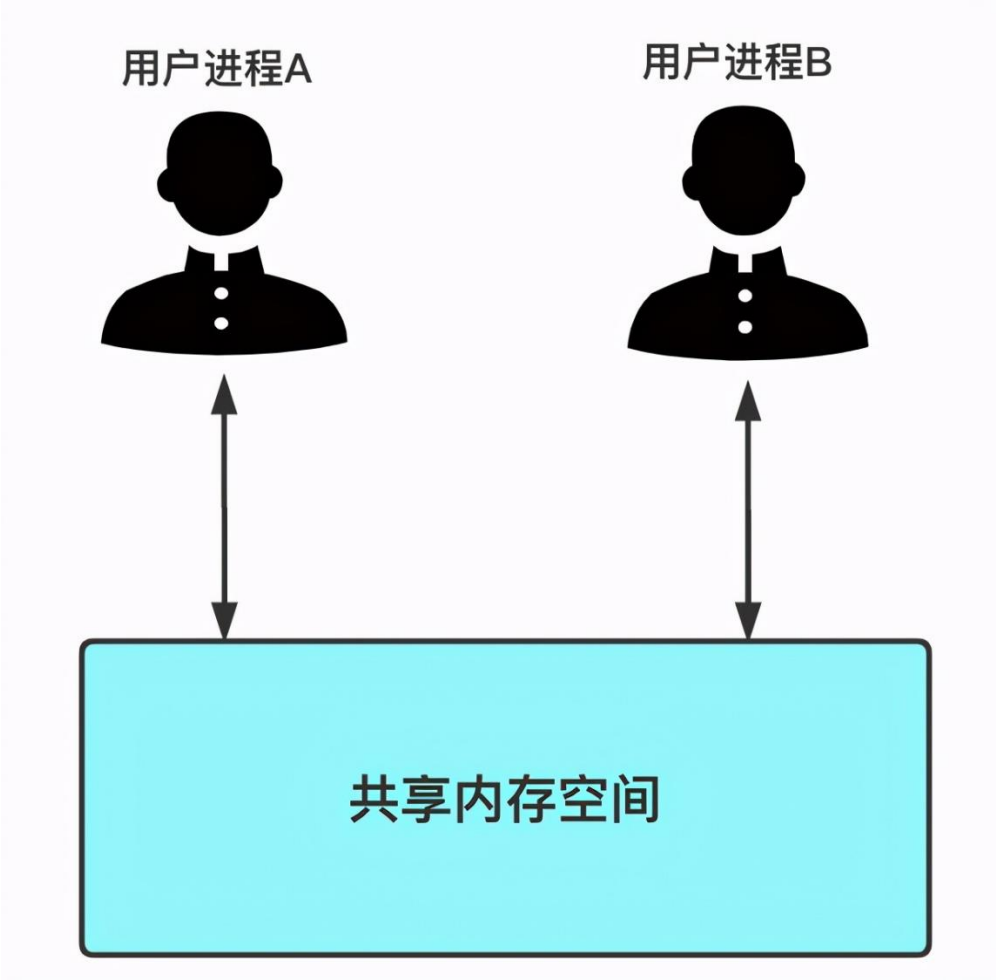
3.6.2 消息队列



消息队列

消息队列是保存在**内核**中的消息链表，**会涉及到用户态跟内核态到来回切换**，双方约定好消息体到数据结构，然后发送数据时将数据分成一个个独立的数据单元消息体，需注意消息队列及单个消息都有上限，日常我们到 RabbitMQ、Redis 都涉及到消息队列。

3.6.3 共享内存



共享空间

现代操作系统对内存管理采用的是虚拟内存技术，也就是每个进程都有自己独立的虚拟内存空间，不同进程的虚拟内存映射到不同的物理内存中。所以，即使进程 A 和进程 B 虚拟地址是一样的，真正访问的也是不同的物理内存地址，该模式不涉及到用户态跟内核态来回切换，JVM 就是用的共享内存模式。并且并发编程也是个难点。

3.6.4 信号量

既然共享内存容易造成数据紊乱，那为了简单的实现共享数据在任意时刻只能被一个进程访问，此时需要信号量。

信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的数据。

信号量表示资源的数量，核心点在于原子性的控制一个数据的值，控制信号量的方式有 PV **两种原子操作：**

1. P 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。
2. V 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。

3.6.5 信号

对于异常状态下进程工作模式需要用到信号工作方式来通知进程。比如 Linux 系统为了响应各种事件提供了很多异常信号 kill -l，**信号是进程间通信机制中唯一的异步通信机制**，可以在任何时候发送信号给某一进程。比如：

1. kill -9 1412 ，表示给 PID 为 1412 的进程发送 SIGKILL 信号，用来立即结束该进程。
2. 键盘 Ctrl+C 产生 SIGINT 信号，表示终止该进程。
3. 键盘 Ctrl+Z 产生 SIGTSTP 信号，表示停止该进程，但还未结束。

有信号发生时，进程一般有三种方式响应信号：

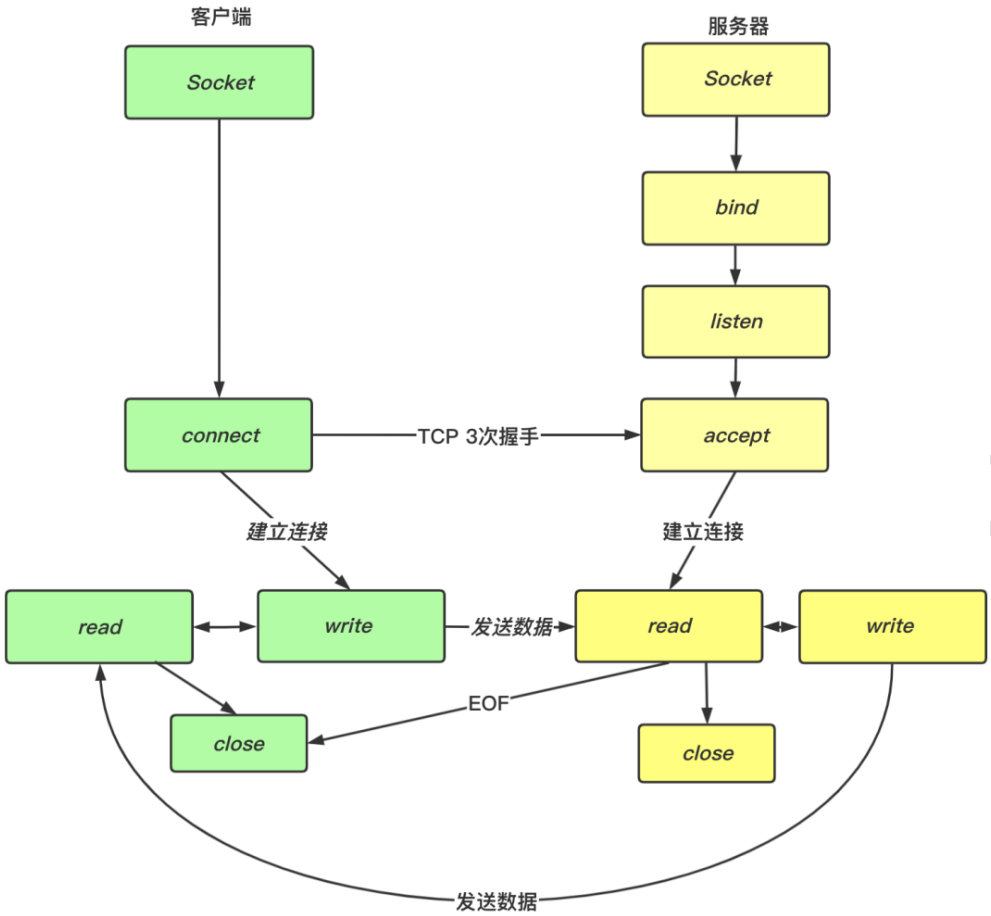
1. 执行默认操作：Linux 操作系统为众多信号配备了专门的处理操作。
2. 捕捉信号：给捕捉到的信号配备专门的信号处理函数。
3. 忽略信号：专门用来忽略某些信号，但 SIGKILL 和 SEGSTOP 是无法被忽略的，为了能在任何时候结束或停止某个进程而存在。

3.6.6 Socket 编程

前面提到的管道、消息队列、共享内存、信号量和信号都是在同一台主机上进行进程间通信，**那要想跨网络与不同主机上的进程之间通信，就需要 Socket 通信**。

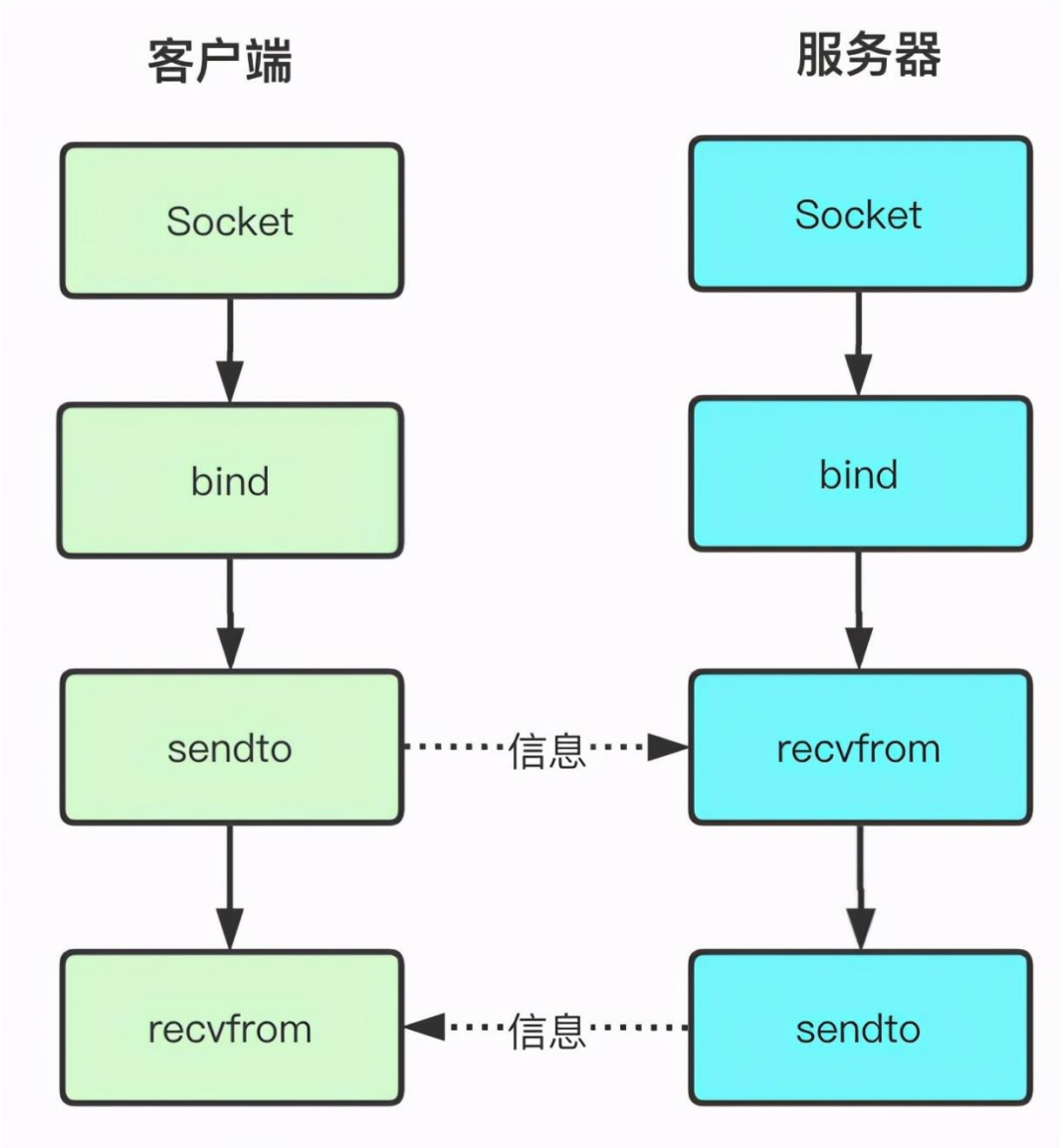
```
int socket(int domain, int type, int protocal)
```

上面是 socket 编程的核心函数，可以指定 IPV4 或 IPV6 类型，TCP 或 UDP 类型。比如 **TCP** 协议通信的 socket 编程模型如下：



Socket 编程

1. 服务端和客户端初始化 socket，得到文件描述符。
2. 服务端调用 bind，将绑定在 IP 地址和端口。
3. 服务端调用 listen，进行监听。
4. 服务端调用 accept，等待客户端连接。
5. 客户端调用 connect，向服务器端的地址和端口发起连接请求。
6. 服务端 accept 返回用于传输的 socket 的文件描述符。
7. 客户端调用 write 写入数据，服务端调用 read 读取数据。
8. 客户端断开连接时，会调用 close，那么服务端 read 读取数据的时候，就会读取到了 EOF，待处理完数据后，服务端调用 close，表示连接关闭。
9. 服务端调用 accept 时，连接成功会返回一个已完成连接的 socket，后续用来传输数据。服务端有俩 socket，一个叫作监听 socket，一个叫作已完成连接 socket。
10. 成功连接建立之后双方开始通过 read 和 write 函数来读写数据。



UDP 传输

UDP 比较简单，属于类似广播性质的传输，不需要维护连接。但也需要 bind，每次通信时调用 sendto 和 recvfrom 都要传入目标主机的 IP 地址和端口。

3.7 多线程编程

既然多进程开销过大，那平常我们经常使用到的就是多线程编程了。期间可能涉及到内存模型、JMM、Volatile、临界区等等。这些在 Java 并发编程专栏有讲。

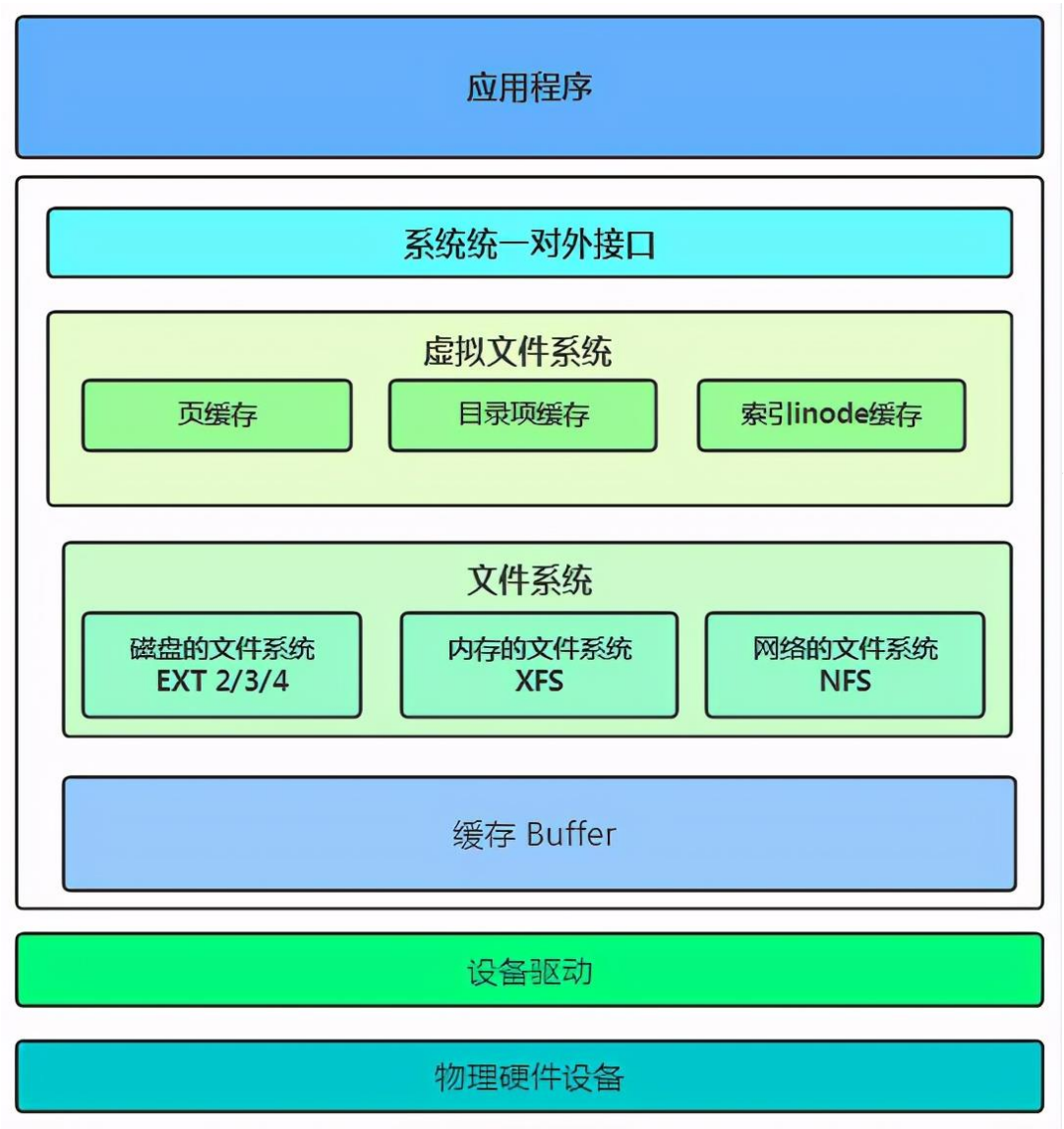
4 文件管理

4.1 VFS 虚拟文件系统

文件系统在操作系统中主要负责将文件数据信息存储到磁盘中，起到持久化文件的作用。文件系统的基本组成单元就是文件，文件组成方式不同就会形成不同的文件系统。

文件系统有很多种而不同的文件系统应用到操作系统后需要提供统一的对外接口，此时用到了一个设计理念没有什么是加一层解决不了的，在用户层跟不同的文件系统之间加入一个虚拟文件系统层 Virtual File System。

虚拟文件系统层定义了一组所有文件系统都支持的数据结构和标准接口，这样程序员不需要了解文件系统的工作原理，只需要了解 VFS 提供的统一接口即可。



虚拟文件系统

日常的文件系统一般有如下三种：

1. 磁盘文件系统：就是我们常见的 EXT 2/3/4 系列。
2. 内存文件系统：数据没存储到磁盘，占用内存数据，比如/sys、/proc。进程中的一些数据映射到/proc 中了。
3. 网络文件系统：常见的网盘挂载 NFS 等，通过访问其他主机数据实现。

4.2 文件组成

以 Linux 系统为例，在 Linux 系统中一切皆文件，Linux 文件系统会为每个文件分配索引节点 inode 跟目录项 directory entry 来记录文件内容跟目录层次结构。

4.2.1 inode

要理解 inode 要从文件储存说起。文件存储在硬盘上，硬盘的最小存储单位叫做扇区。每个扇区储存 512 字节。操作系统读取硬盘的时候，不会一个个扇区的读取，这样效率太低，一般一次性连续读取 8 个扇区(4KB)来当做一块，这种由多个扇区组成的**块**，是文件存取的最小单位。

文件数据都储存在块中，我们还必须找到一个地方储存文件的元信息，比如 inode 编号、文件大小、创建时间、修改时间、磁盘位置、访问权限等。几乎除了文件名以为的所有文件元数据信息都存储在一个叫叫索引节点 inode 的地方。可通过 stat 文件名查看 inode 信息

每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件。Unix/Linux 系统内部不使用文件名，而使用 inode 号码来识别文件，用户可通过 ls -i 查看每个文件对应编号。对于系统来说文件名只是 inode 号码便于识别的别称或者绰号。特殊名字的文件不好删除时可以尝试用 inode 号删除，移动跟重命名不会导致文件 inode 变化，当用户尝试根据文件名打开文件时，实际上系统内部将这个过程分成三步：

1. 系统找到这个文件名对应的 inode 号码。
2. 通过 inode 号码，获取 inode 信息，进行权限验证等操作。
3. 根据 inode 信息，找到文件数据所在的 block，读出数据。

需注意 inode 也会消耗硬盘空间，硬盘格式化后会被分成**超级块**、**索引节点区**和**数据块区**三个区域：

1. 超级块区：用来存储文件系统的详细信息，比如块大小，块个数等信息。一般文件系统挂载后就会将数据信息同步到内存。
2. 索引节点区：用来存储索引节点 inode table。每个 inode 一般为 128 字节或 256 字节，一般每 1KB 或 2KB 数据就需设置一个 inode。一般为了加速查询会把索引数据缓存到内存。
3. 数据块区：真正存储磁盘数据的地方。df -i # 查看每个硬盘分区的 inode 总数和已经使用的数量

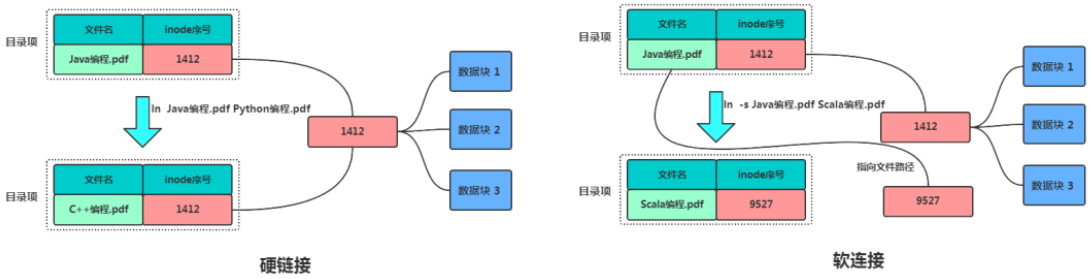
   sudo dumpe2fs -h /dev/hda | grep "Inode size" # 查看每个 inode 节点的大小

4.2.2 目录

Unix/Linux 系统中**目录 directory 也是一种文件**，打开目录实际上就是打开目录文件。目录文件内容就是一系列目录项的列，目录项的内容包含**文件的名字、文件类型、索引节点指针以及与其他目录项的层级关系**。

为避免频繁读取磁盘里的目录文件，内核会把已经读过的目录文件用目录项这个数据结构缓存在内存，方便用户下次读取目录信息，目录项可包含目录或文件，不要惊讶于可以保存目录，目录格式的目录项里面保存的是目录里面一项一项的文件信息。

4.2.3 软连接跟硬链接



软连接跟硬链接

硬链接：老文件 A 被创建若干个硬链接 B、C 后。A、B、C 三个文件的 inode 是相同的，所以不能跨文件系统。同时只有 ABC 全部删除，系统才会删除源文件。
软链接：相当于基于老文件 A 新建了个文件 B，该文件 B 有新的 inode，不过文件 B 内容是老文件 A 的路径。所以软链接可以跨文件系统。当老文件 A 删除后，文件 B 仍然存在，不过找不到指定文件了。

[sowhat@localhost ~]$ ln [选项] 源文件 目标文件
选项：
-s：建立软链接文件。如果不加 "-s" 选项，则建立硬链接文件；
-f：强制。如果目标文件已经存在，则删除目标文件后再建立链接文件；

4.3 文件存储

说文件存储前需了解**文件系统操作基本单位是数据块**，而平常用户操作字节到数据块之间是需要转换的，当然这些文件系统都帮我们对接好了。接下来看文件系统是如何按照数据块， 文件在磁盘中存储时候主要分为连续空间存储跟非连续空间存储

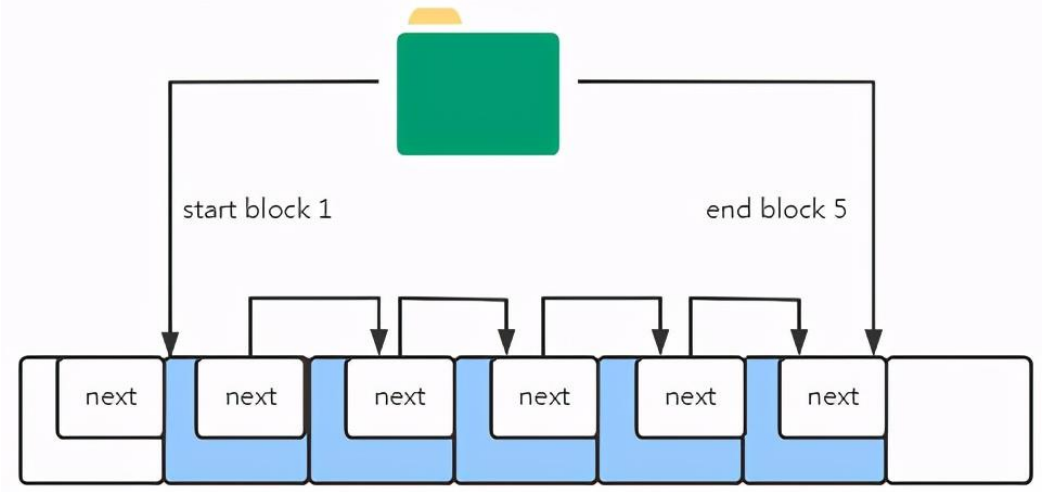
4.3.1 连续空间存储

1. 实现：连续空间存储的意思就跟数组存储一样，找个连续的空间一次性把数据存储进去，**文件头**存储起始位置跟数据长度即可。
2. 优势：读写效率高，磁盘寻址一次即可。
3. 劣势：容易产生空间碎片，并且文件扩容不方便。
   连续存储

4.3.2 非连续空间存储之链表

隐式链表

1. 实现：文件头包含 StartBlock、EndBlock。每个 BLock 有隐藏的 next 指针，跟单向链表一样。
2. 缺点：只能通过链式不断往下查找数据，不支持快速直接访问。



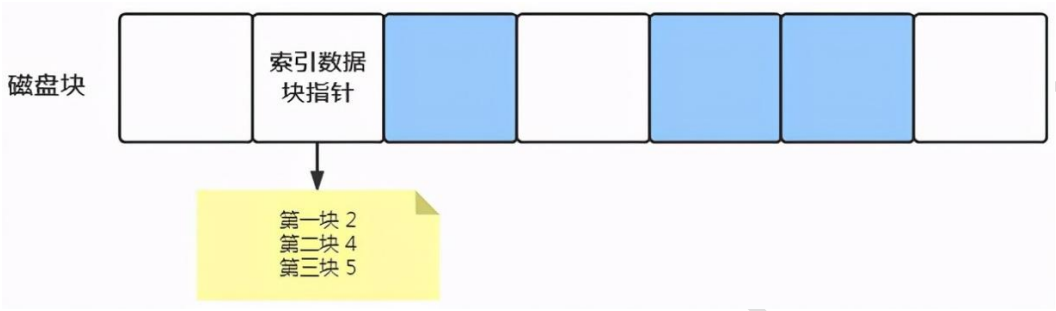
隐式链表

显式链表

1. 实现：把每个 Block 中的 next 指针存储到内存文件分配表中，通过遍历数组方式实现拿到全部数据。
2. 缺点：前面说 1KB 就有个 inode 指针，如果磁盘数据很大那就需要很大的**文件分配表**来存储映射关系了，
   显示链表

4.3.3 非连续空间存储之索引

1. 实现：整个文件类型一本新华字典，真实的数据块在词典实际位置存储着，但文件所需数据块的索引位置会被汇总起来形成目录索引放在字典前头。
2. 优势：不会产生碎片，文件可动态扩容，并且支持顺序跟随机读写。
3. 劣势：可能一个小文件都要占用一个目录索引，文件过大导致索引指针一个容不下，可能还需要有多级索引或索引+链表模式。



索引存储

这些存储方式各有利弊，所以操作系统才存储的时候一般是根据文件的大小进行动态的变化存储方式的，跟 STL 中的快排底层 = 快排 + 插入排序 + 堆排 一样的道理。

4.3.4 空闲空间管理

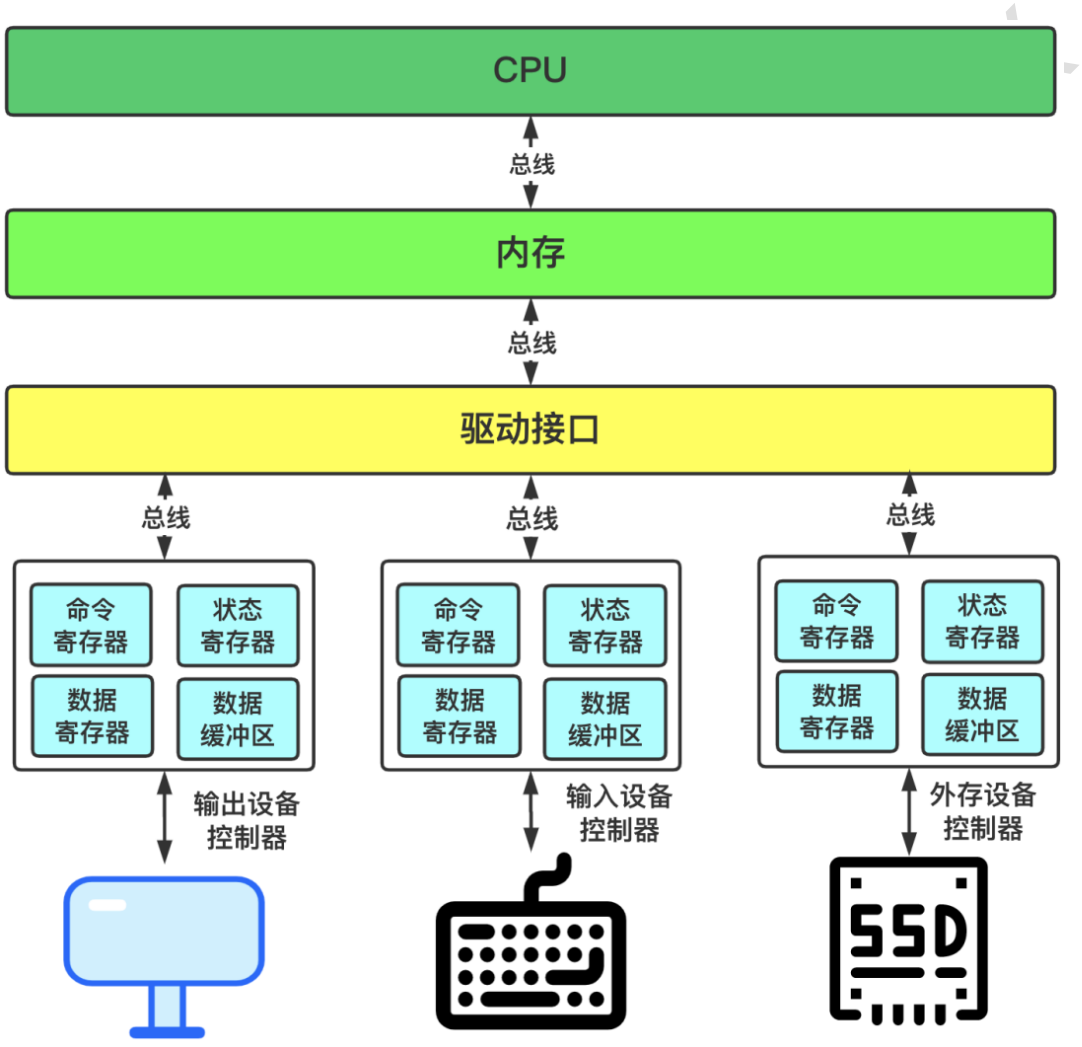
为了避免用户存储数据时候遍历全部磁盘空间来寻找可以数据块，一般有如下几种记录方法。

1. 空闲表：动态的维护一个空闲数据块列表，每行存储空闲块的开始位置跟空闲长度。适合少量有少量空闲数据块时。
   空闲表
2. 空闲链表：将空闲的数据库用 next 指针串联起来，缺点是不能随机访问。
   空闲链表
3. 位图法：利用 Bit 的 01 表示数据块可用跟不可用，简单方便，**inode 跟空闲数据库都用的此方法**。
   位图法

5 输入输出管理

5.1 设备控制器跟驱动程序

5.1.1 设备控制器



设备控制器

操作系统为统一管理众多的设备并且屏蔽设备之间的差异，给每个设备都安装了个小 CPU 叫**设备控制器**。每个设备控制器都知道自己对应外设的功能跟用法，并且每个**设备控制器**都有独有的寄存器用来跟 CPU 通信。

1. 读设备寄存器值了解设备状态，是否可以接收新指令。

2. 操作系统给设备寄存器写入一些指令可以实现发送数据、接收数据等等操作。

控制器一般分为**数据寄存器、命令寄存器跟状态寄存器**，CPU 通过读、写设备控制器中的寄存器来便捷的控制设备：

1. 数据寄存器：CPU 向 I/O 设备写入需要传输的数据，比如打印 what，CPU 就要先发送一个 w 字符给到对应的 I/O 设备。
2. 命令寄存器：CPU 发送命令来告诉 I/O 设备要进行输入/输出操作，于是就会交给 I/O 设备去工作，任务完成后，会把状态寄存器里面的状态标记为完成。
3. 状态寄存器：用来告诉 CPU 现在已经在工作或工作已经完成，只有状态寄存标记成已完成，CPU 才能发送下一个字符和命令。

同时输入输出设备可分为块设备跟字符设备。

1. 块设备：用来把数据存储在固定大小的块中，每个块有自己的地址，硬盘、U 盘等是常见的块设备。块设备一般数据传输较大为避免频繁 IO，控制器中有个可读写等**数据缓冲区**。Linux 操作系统为屏蔽不同块设备带来的差异引入了**通用块层**，**通用块层**是处于文件系统和磁盘驱动中间的一个块设备抽象层，主要提供如下俩功能：

向上为文件系统和应用程序，提供访问块设备的标准接口，向下把各种不同的磁盘设备抽象为统一的块设备，并在内核层面提供一个框架来管理这些设备的驱动程序。

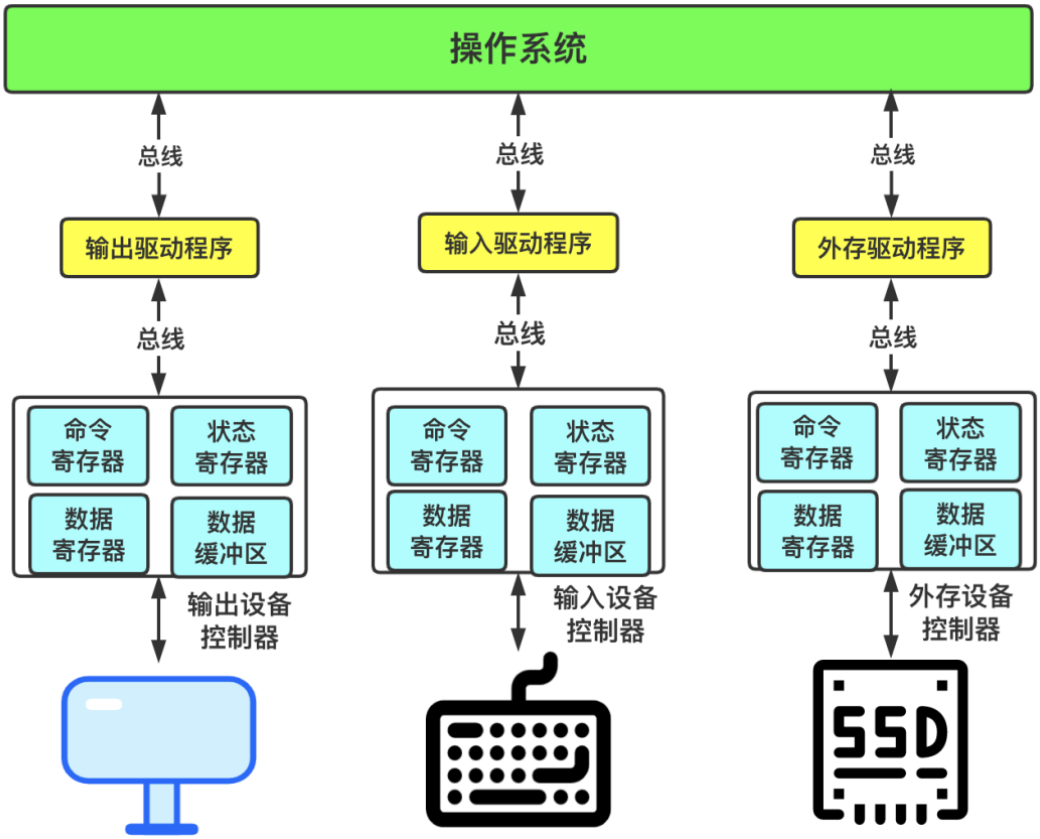
通用层还会给文件系统和应用程序发来的 I/O 进行**调度**，主要目的是为了提高磁盘读写的效率。

1. 字符设备：以字符为单位发送或接收一个字符流，字符设备是不可寻址的，也没有任何寻道操作，鼠标是常见的字符设备。

CPU 一般通过 **IO 端口**跟**内存映射 IO** 来跟设备的控制寄存器和数据缓冲区进行通信

1. IO 端口：每个控制寄存器被分配一个 I/O 端口，可以通过特殊的汇编指令操作这些寄存器，比如 in/out 类似的指令。
2. 内存映射 IO：将所有控制寄存器映射到内存空间中，这样就可以像读写内存一样读写数据缓冲区。

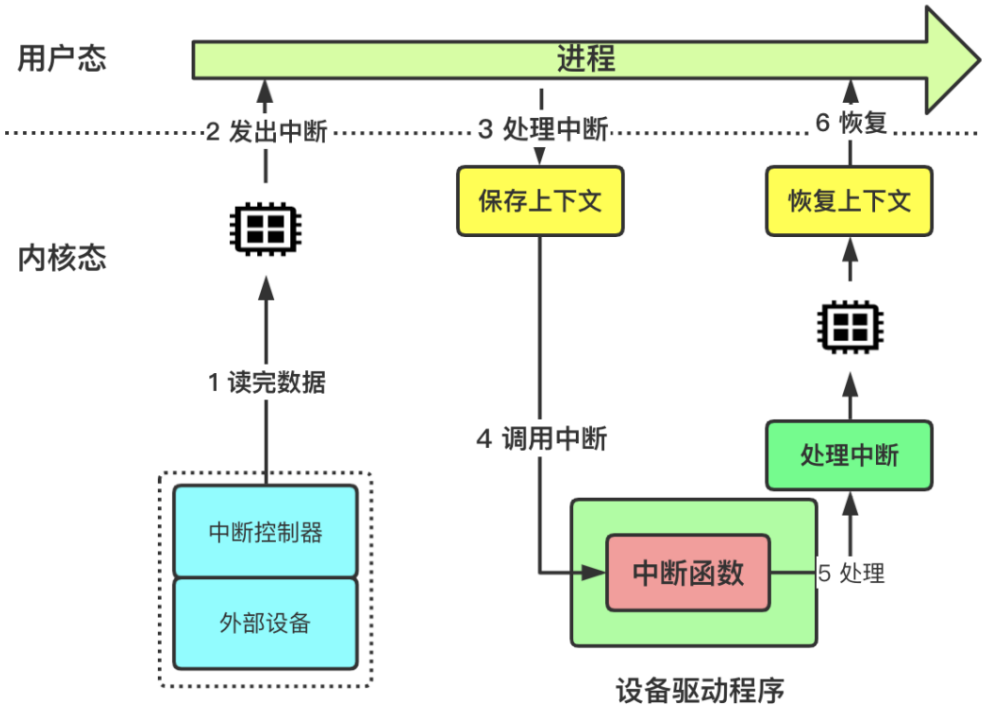
5.1.2 驱动接口



驱动程序

设备控制器屏蔽了设备细节，但每种设备的控制器的寄存器、缓冲区等使用模式都是不同的，它属于硬件。在操作系统图范畴内为了屏蔽设备控制器的差异，引入了**设备驱动程序，不同设备到驱动程序会提供统一接口给操作系统来调用**，这样操作系统内核会像调用本地代码一样使用设备驱动程序接口。

设备发出 IO 请求就是在**设备驱动程序**中来响应到，它会根据中断类型调用响应到中断处理程序进行处理。



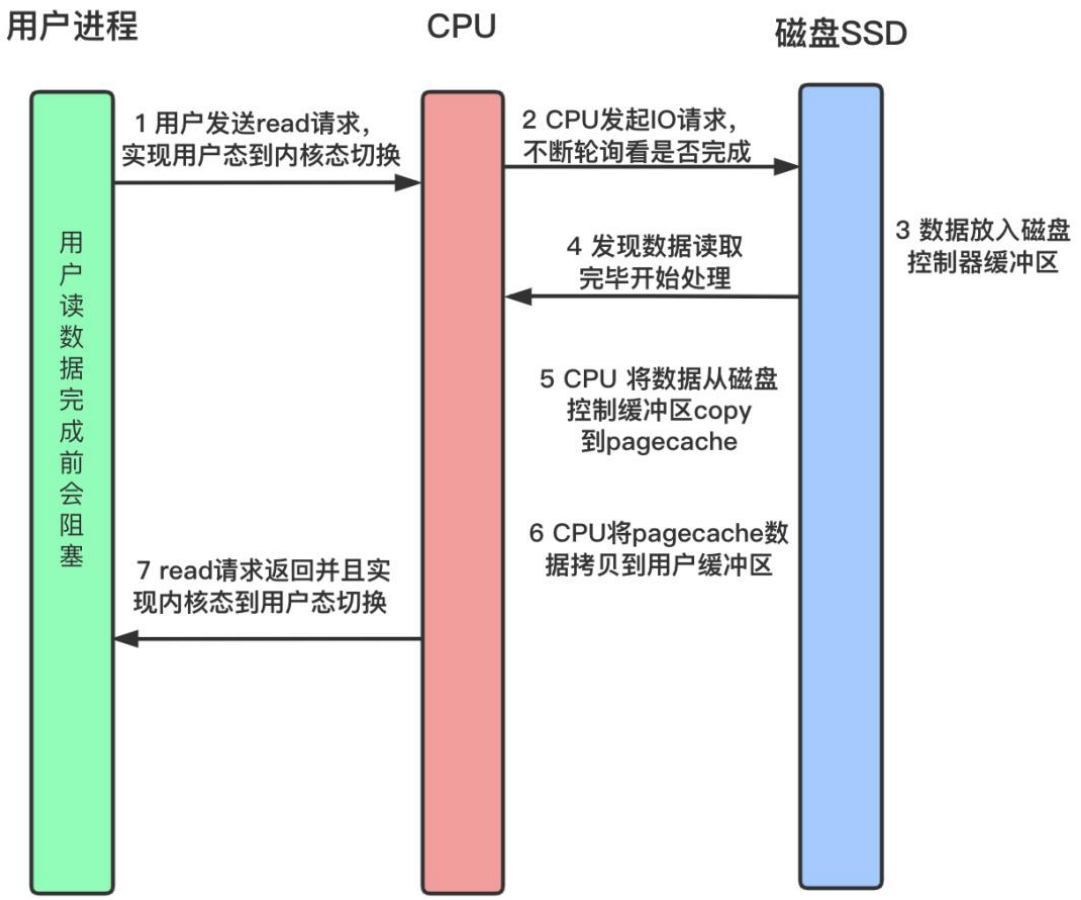
中断请求流程

5.2 IO 控制

CPU 发送指令让那个设备控制器去读写数据，完毕后如何通知 CPU 呢？

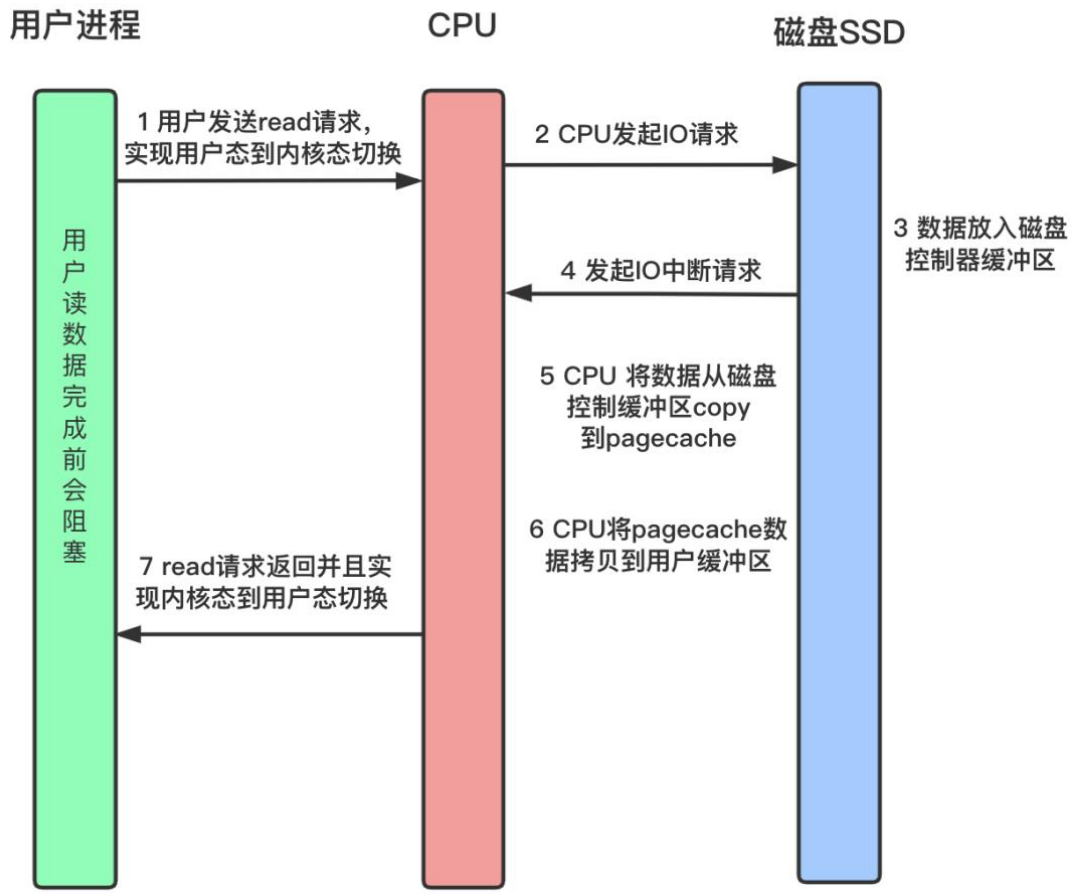
5.2.1 轮询模式

控制器中有个**状态寄存器**，CPU 不断**轮**询查看寄存器状态，该模式会傻瓜式的一直占用 CPU。


用户进程
CPU
磁盘SSD
用户读数据完成前会阻塞
1 用户发送read请求，实现用户态到内核态切换
2 CPU发起IO请求，不断轮询看是否完成
3 数据放入磁盘控制器缓冲区
4 发现数据读取完毕开始处理
5 CPU 将数据从磁盘控制缓冲区copy到pagecache
6 CPU将pagecache数据拷贝到用户缓冲区
7 read请求返回并且实现内核态到用户态切换

轮询模式

5.2.2 IO 中断请求



中断模式

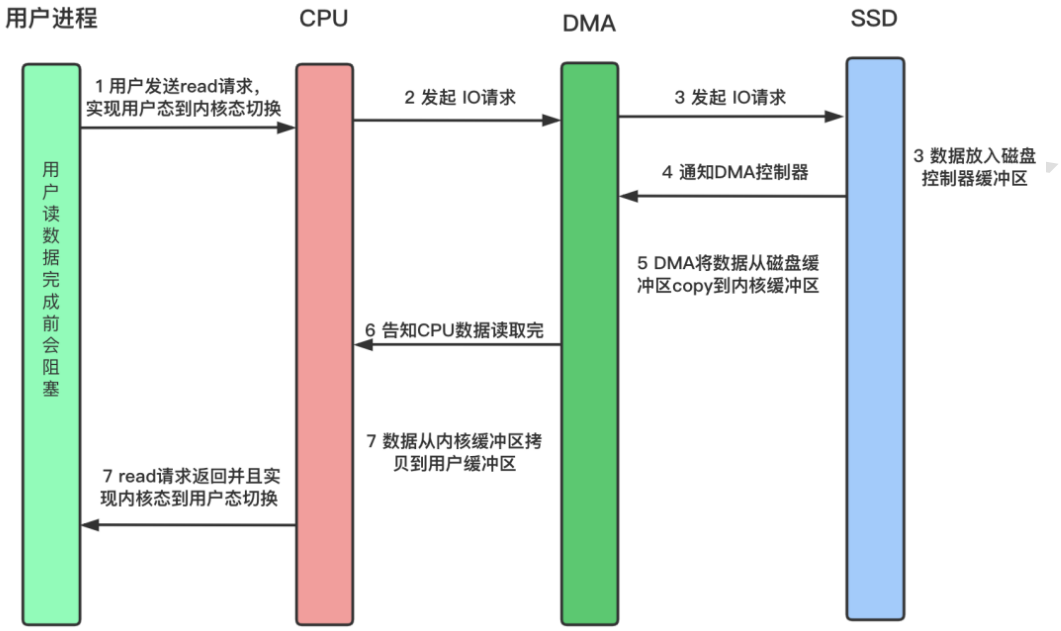
控制器有个中断控制器，当设备完成任务后触发中断到中断控制器，中断控制器就通知 CPU 来处理中断请求。中断有两种，一种是**软中断**，比如代码调用 INT 指令触发。一种是**硬件中断**，硬件通过中断控制器触发的。但中断方式对于频繁读写磁盘数据的操作就不太友好了，会频繁打断 CPU。

这里说下磁盘高速缓存 PageCache，它是用来缓存最近被 CPU 访问的数据到内存中，并且还具有预读功能，可能你读前 16KB 数据，已经把后 16KB 数据给你缓存好了。

pagecache ：页缓存，当进程需读取磁盘文件时，linux 先分配一些内存，将数据从磁盘读区到内存中，然后再将数据传给进程。当进程需写数据到磁盘时，linux 先分配内存接收用户数据，然后再将数据从内存写到磁盘。同时 pagecache 由于大小受限，所以一般只缓存最近被访问的数据，数据不足时还需访问磁盘。

5.2.3 DMA 模式

Direct Memory Access 直接内存访问，在硬件 DMA 控制器的支持下，**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，让 CPU 去处理别的事**。



DMA 模式

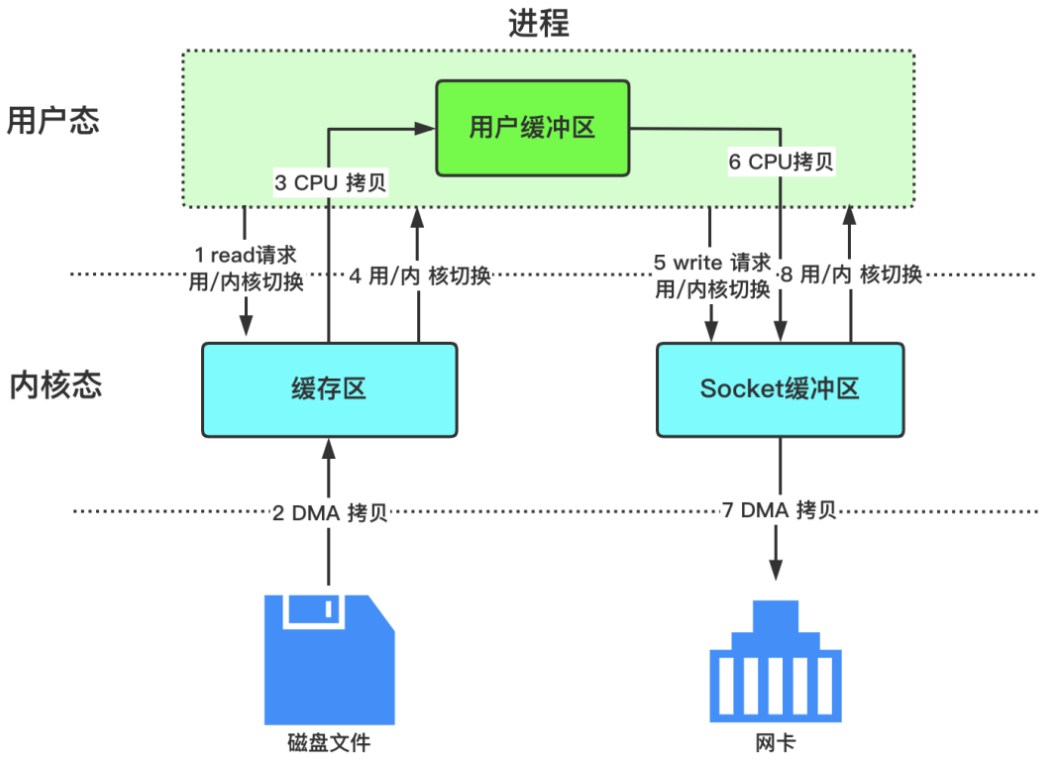
可以发现整个数据传输过程中 CPU 是不会直接参与数据搬运工作，由 DMA 来直接负责数据读取工作，现如今每个 IO 设备一般都自带 DMA 控制器。读数据时候仅仅在传送开始跟结束时需要 CPU 干预。

5.2.4 Zero Copy

Zero Copy 全程不会通过 CPU 来搬运数据，所有的数据都是通过 DMA 来进行传输的，中间只需要经过 2 次上下文切换跟 2 次 DMA 数据拷贝，相比最原始读写方式至少速度翻倍。其实在 Kafka 中已经讲过 Zero Copy 了。

5.2.4.1 老版本读写

老版本的简单读写操作中间不对数据做任何操作。期间会发生 **4 次用户态跟内核态的切换。2 次 DMA 数据拷贝，2 次 CPU 数据拷贝**。

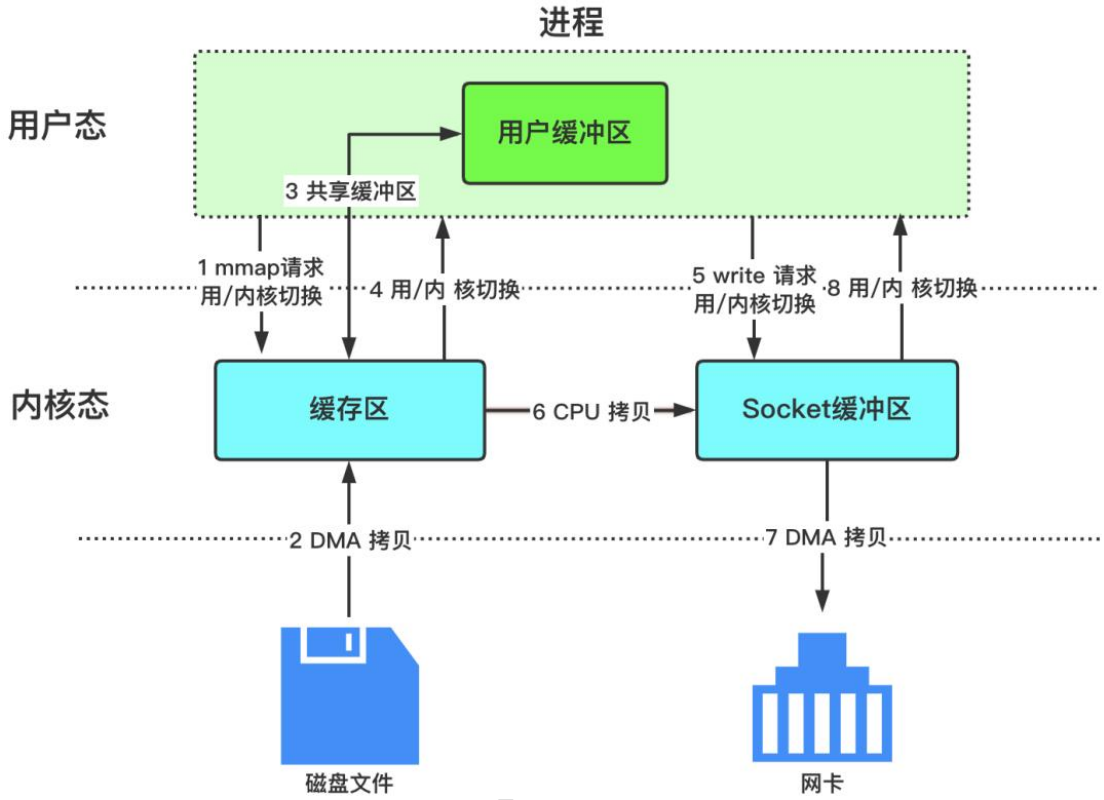


老式读写

提速方法就是需减少用户态与内核态的上下文切换和内存拷贝的次数。数据传输时从内核的读缓冲区拷贝到用户的缓冲区，再从用户缓冲区拷贝到 socket 缓冲区的这个过程是没有必要的。接下来

接下来按照三个版本说下 Zero Copy 发展史。

5.2.4.2 mmap 跟 write



mmap + write

思路就是用 **mmap** 替代 read 函数，mmap 调用时会**直接把内核缓冲区里的数据映射到用户空间**，此时减少了一次数据拷贝，但仍然需要通过 CPU 把内核缓冲区的数据拷贝到 socket 缓冲区里，而且仍然需要 4 次上下文切换，因为系统调用还是 2 次。

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

5.2.4.3 sendfile

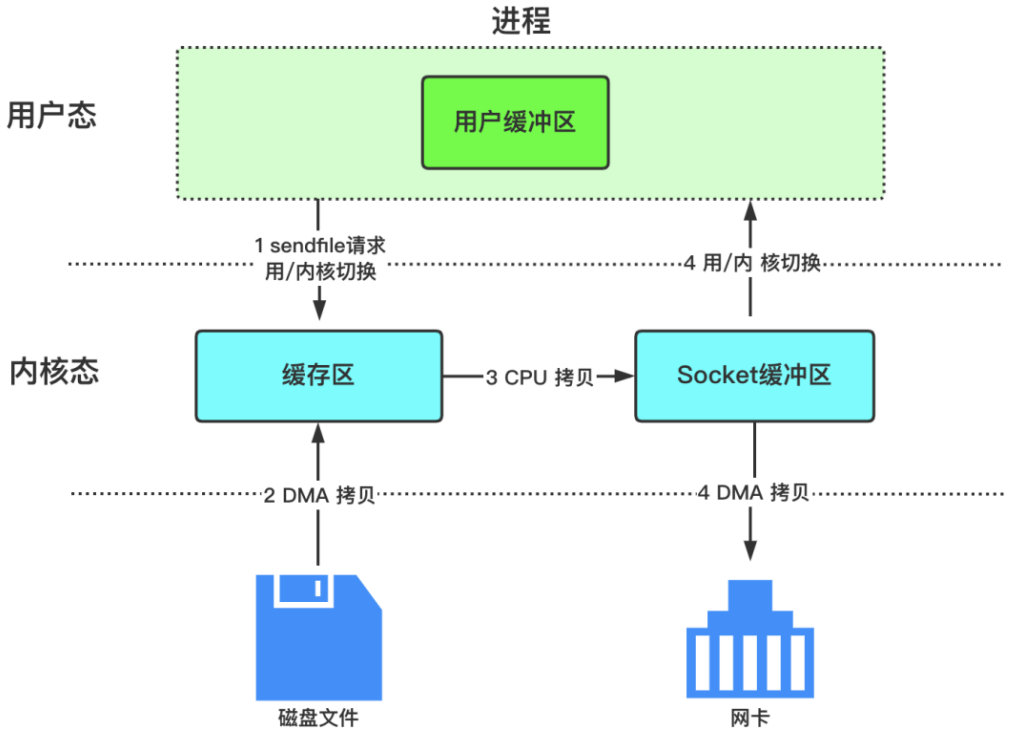
Linux 内核版本 2.1 版本提供了函数 **sendfile()**。

```
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

out_fd : 目的文件描述符
in_fd:源文件描述符
offset:源文件内偏移量

count:打算复制数据长度
ssize_t:实际上复制数据的长度

可以发现一个 sendfile = read + write，避免了 2 次用户态跟内核态来回切换，并且可以直接把内核缓冲区里的数据拷贝到 socket 缓冲区里，这样就只有 2 次上下文切换，和 3 次数据拷贝。



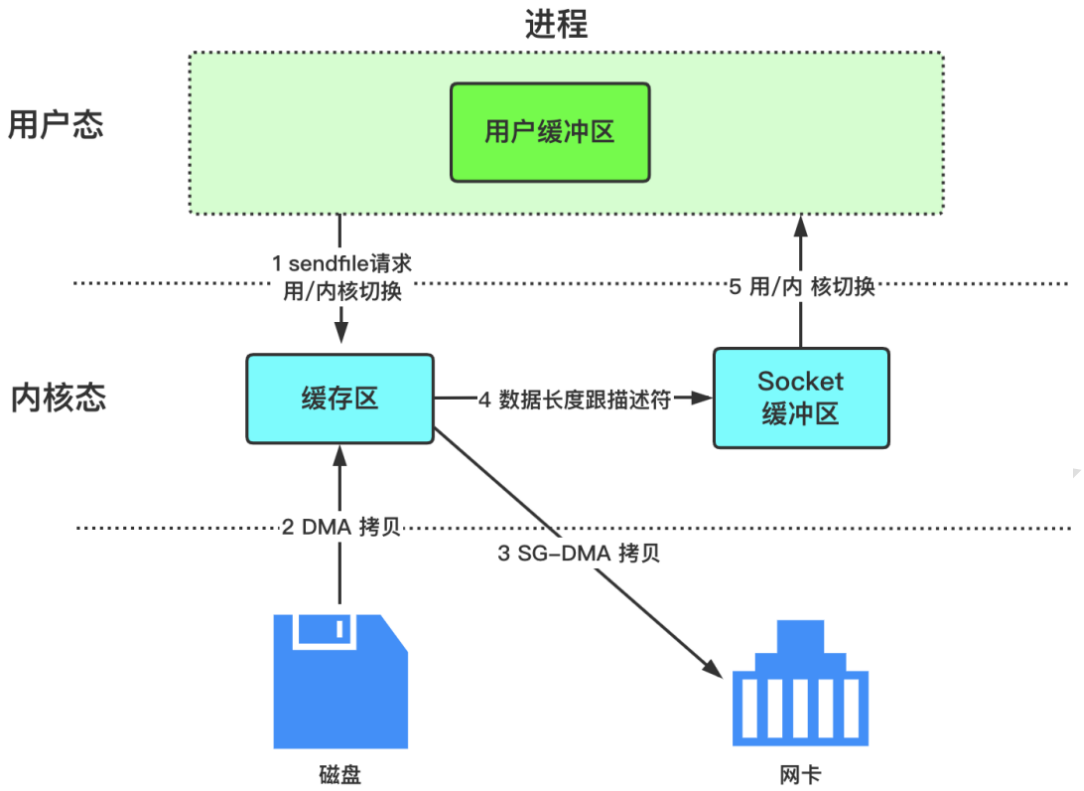
sendfile 模式

5.2.4.4 真正的零拷贝

Linux 内核 2.4 如果网卡支持 SG-DMA 技术，可以减少通过 CPU 把内核缓冲区里的数据拷贝到 socket 缓冲区的过程。

```
$ ethtool -k eth0 | grep scatter-gather
scatter-gather: on
```

SG-DMA 技术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。



编辑

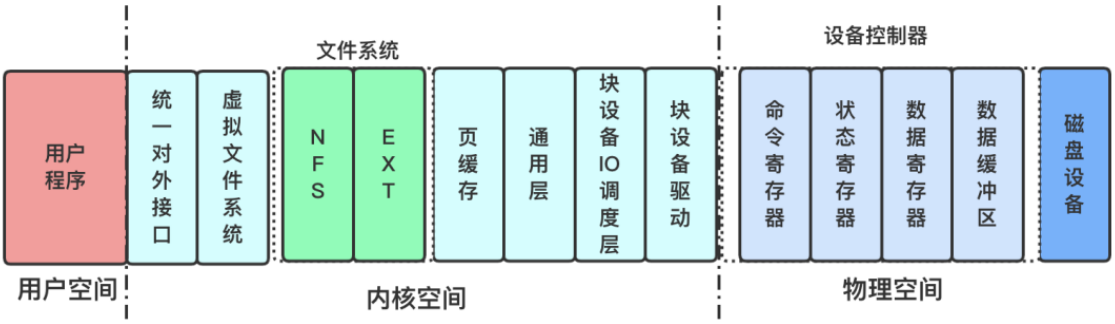
ZeroCopy

5.2.4.5 文件传输规则

不要以为会了 Zero Copy 后，无论大小文件都用 Zero Copy。实际工作中一般小文件采用 Zero Copy 技术，而大文件会用异步 IO。至于为啥，且看如下分析：

前面说的数据从磁盘读到内核缓冲区就是读到 PageCache 中，PageCache 具有缓存跟预读功能。但当传输超大文件时 PageCache 会不失效，因为大文件会快速占满 PageCache 区，但这些文件又只是一次访问，会造成其他热点小文件无法使用 PageCache，所以索性不用 PageCache，使用异步 IO 的了。至于异步 IO 是啥呢？下文在说。

5.3 IO 分层



编辑

IO 分层

Linux 存储系统的 I/O 由上到下可以分为**文件系统层、通用块层、设备层**。

1. 文件系统层向上为应用程序统一提供了标准的文件访问接口，向下会通过通用块层来存储和管理磁盘数据。
2. 通用块层包括块设备的 I/O 队列和 I/O 调度器，通过 IO 调度器处理 IO 请求。
3. 设备层包括硬件设备、设备控制器和驱动程序，负责最终物理设备的 I/O 操作。

Linux 系统中的 IO **读取提速：**

1. 为提高文件访问效率会使用页缓存、索引节点缓存、目录项缓存等多种缓存机制，目的是为了减少对块设备的直接调用。
2. 为了提高块设备的访问效率， 会使用缓冲区，来缓存块设备的数据。