# 二、硬件结构

常见的**寄存器**种类：

* 通用寄存器，用来存放需要进行运算的数据，比如需要进行加和运算的两个数据。
* 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」，注意不是存储了下一条要执行的指令，此时指令还在内存中，程序计数器只是存储了下一条指令「的地址」。
* 指令寄存器，用来存放当前正在执行的指令，也就是指令本身，指令被执行完成之前，指令都存储在这里。

**总线**是用于 CPU 和内存以及其他设备之间的通信，总线可分为 3 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 收到信号后自然进行响应，这时也需要控制总线；

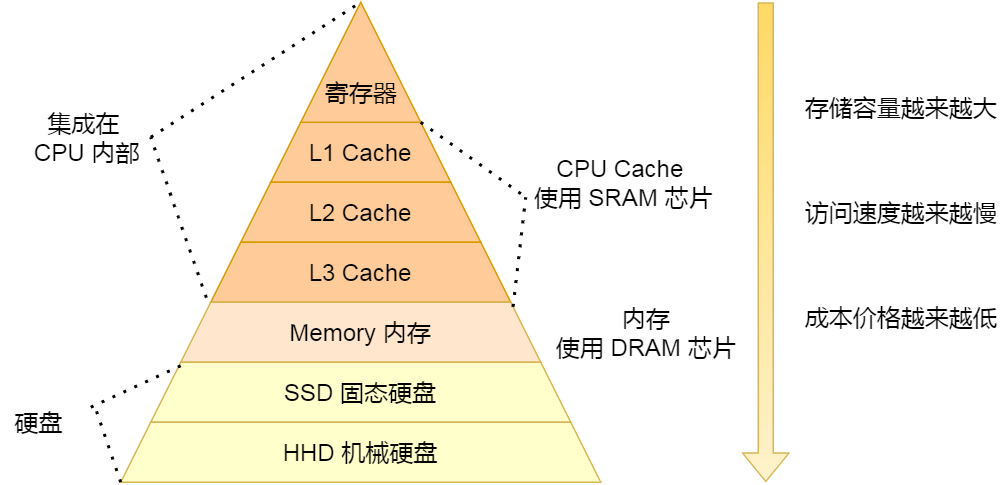
CPU 要读写内存数据的时候，一般需要通过下面这三个总线：

* 首先要通过「地址总线」来指定内存的地址；
* 然后通过「控制总线」控制是读或写命令；
* 最后通过「数据总线」来传输数据；

CPU 执行程序的过程如下：

* 第一步，CPU 读取「程序计数器」的值，这个值是指令的内存地址，然后 CPU 的「控制单元」操作「地址总线」指定需要访问的内存地址，接着通知内存设备准备数据，数据准备好后通过「数据总线」将指令数据传给 CPU，CPU 收到内存传来的数据后，将这个指令数据存入到「指令寄存器」。
* 第二步，「程序计数器」的值自增，表示指向下一条指令。这个自增的大小，由 CPU 的位宽决定，比如 32 位的 CPU，指令是 4 个字节，需要 4 个内存地址存放，因此「程序计数器」的值会自增 4；
* 第三步，CPU 分析「指令寄存器」中的指令，确定指令的类型和参数，如果是计算类型的指令，就把指令交给「逻辑运算单元」运算；如果是存储类型的指令，则交由「控制单元」执行；

## 存储器的层次结构



当 CPU 需要访问内存中某个数据的时候，如果寄存器有这个数据，CPU 就直接从寄存器取数据即可，如果寄存器没有这个数据，CPU 就会查询 L1 高速缓存，如果 L1 没有，则查询 L2 高速缓存，L2 还是没有的话就查询 L3 高速缓存，L3 依然没有的话，才去内存中取数据。

## CPU Cache 的数据写入

数据写入 Cache 之后，内存与 Cache 相对应的数据将会不同，这种情况下 Cache 和内存数据都不一致了，于是我们肯定是要把 Cache 中的数据同步到内存里的。

问题来了，那在什么时机才把 Cache 中的数据写回到内存呢？为了应对这个问题，下面介绍两种针对写入数据的方法：

* 写直达（Write Through）
* 写回（Write Back）

### [#](https://xiaolincoding.com/os/1_hardware/cpu_mesi.html#%E5%86%99%E7%9B%B4%E8%BE%BE)写直达

**把数据同时写入内存和 Cache 中**

* 如果数据已经在 Cache 里面，先将数据更新到 Cache 里面，再写入到内存里面；
* 如果数据没有在 Cache 里面，就直接把数据更新到内存里面。

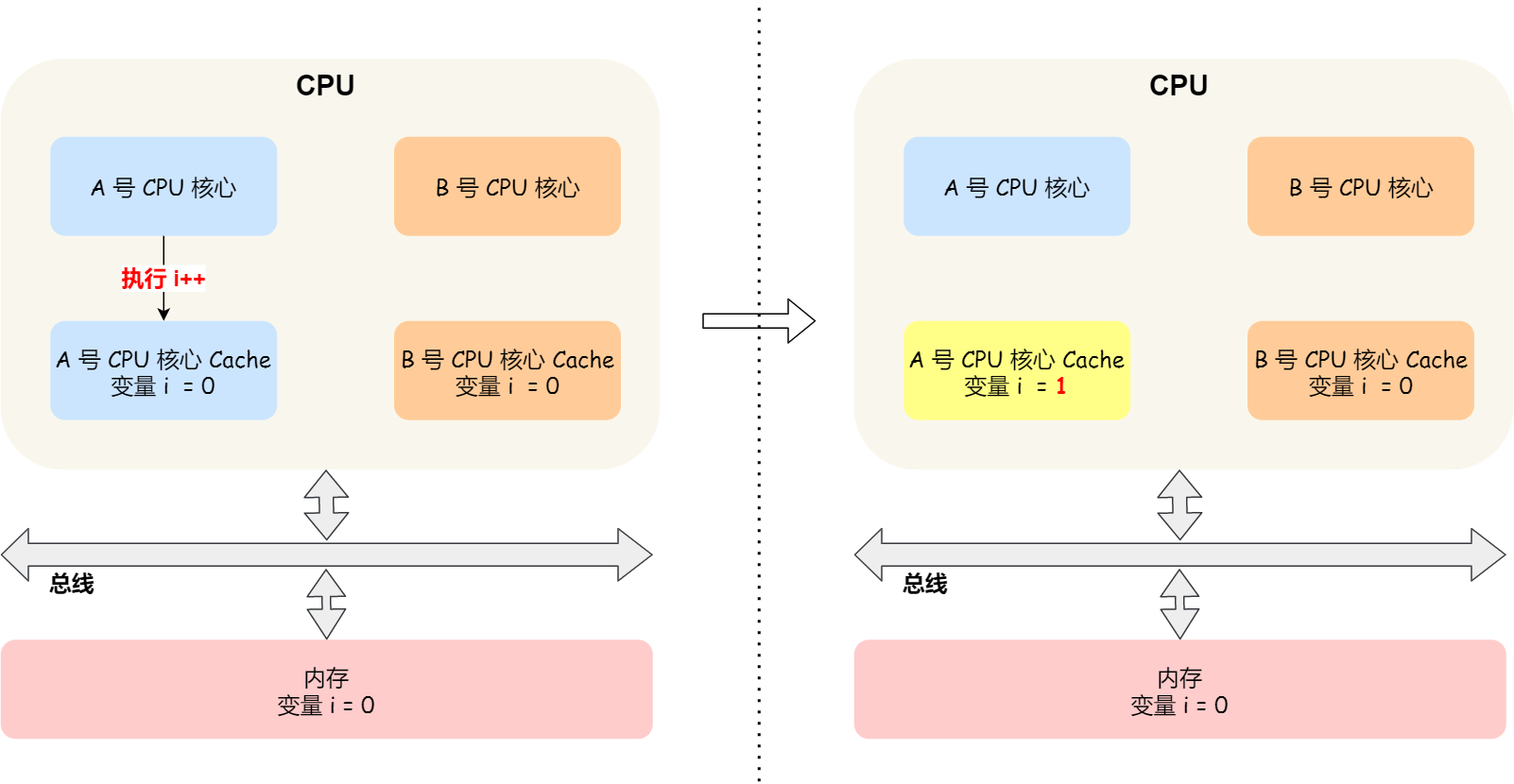
### 写回

**当发生写操作时，新的数据仅仅被写入 Cache Block 里，只有当修改过的 Cache Block「被替换」时才需要写到内存中**

* 如果当发生写操作时，数据已经在 CPU Cache 里的话，则把数据更新到 CPU Cache 里，同时标记 CPU Cache 里的这个 Cache Block 为脏（Dirty）的，这个脏的标记代表这个时候，我们 CPU Cache 里面的这个 Cache Block 的数据和内存是不一致的，这种情况是不用把数据写到内存里的；
* 如果当发生写操作时，数据所对应的 Cache Block 里存放的是「别的内存地址的数据」的话，就要检查这个 Cache Block 里的数据有没有被标记为脏的：
  + 如果是脏的话，我们就要把这个 Cache Block 里的数据写回到内存，然后再把当前要写入的数据，先从内存读入到 Cache Block 里（注意，这一步不是没用的），然后再把当前要写入的数据写入到 Cache Block，最后也把它标记为脏的；
  + 如果不是脏的话，把当前要写入的数据先从内存读入到 Cache Block 里，接着将数据写入到这个 Cache Block 里，然后再把这个 Cache Block 标记为脏的就好了

## 缓存一致性问题

 CPU 都是多核的，由于 L1/L2 Cache 是多个核心各自独有的，那么会带来多核心的**缓存一致性（Cache Coherence）** 的问题



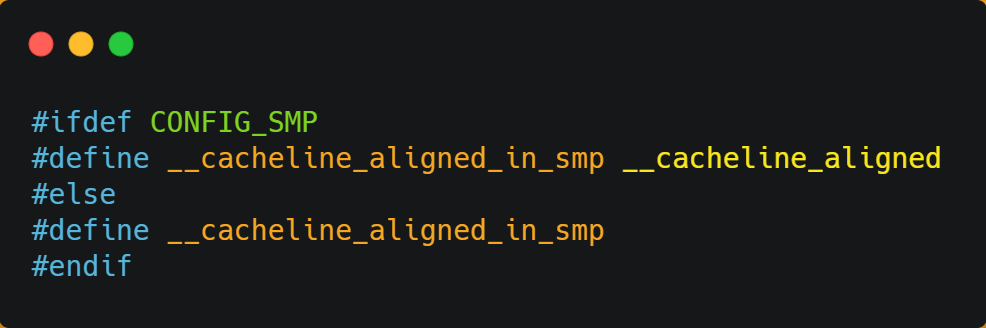
要保证做到下面这 2 点：

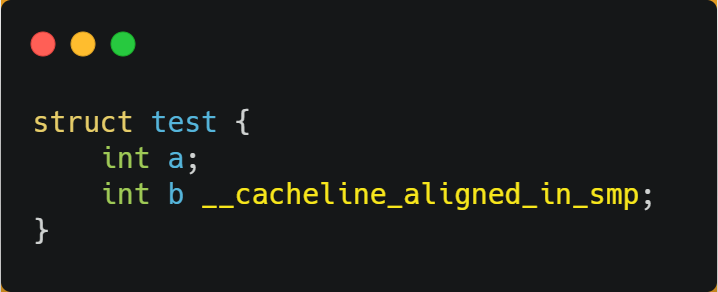
* 第一点，某个 CPU 核心里的 Cache 数据更新时，必须要传播到其他核心的 Cache，这个称为写传播（*Write Propagation*）；
* 第二点，某个 CPU 核心里对数据的操作顺序，必须在其他核心看起来**顺序是一样**的，这个称为事务的串行化（*Transaction Serialization*）。

### 伪共享

多个线程同时读写同一个 Cache Line 的不同变量时，而导致 CPU Cache 失效的现象称为**伪共享（False Sharing）**。

对于多个线程共享的热点数据，即经常会修改的数据，应该避免这些数据刚好在同一个 Cache Line 中，否则就会出现为伪共享的问题。





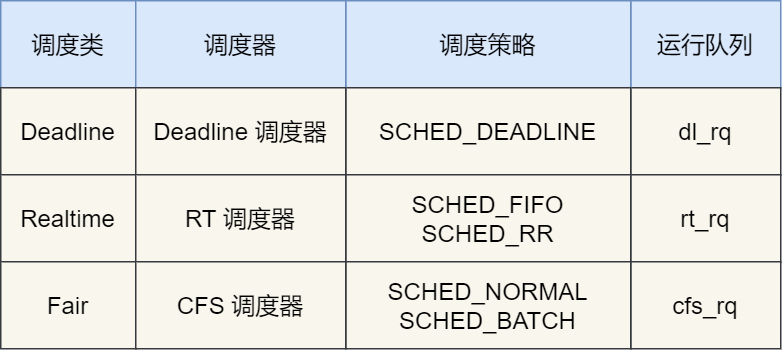
针对在同一个 Cache Line 中的共享的数据，如果在多核之间竞争比较严重，为了防止伪共享现象的发生，可以采用上面的宏定义使得变量在 Cache Line 里是对齐的。

## CPU 如何选择线程的？

在 Linux 内核中，进程和线程都是用 task\_struct 结构体表示的，区别在于线程的 task\_struct 结构体里部分资源是共享了进程已创建的资源，比如内存地址空间、代码段、文件描述符等，所以 Linux 中的线程也被称为轻量级进程

Linux 内核里的调度器，调度的对象就是 task\_struct

### 调度类



Deadline 和 Realtime是应用于实时任务的：

* *SCHED\_DEADLINE*：是按照 deadline 进行调度的，距离当前时间点最近的 deadline 的任务会被优先调度；
* *SCHED\_FIFO*：对于相同优先级的任务，按先来先服务的原则，但是优先级更高的任务，可以抢占低优先级的任务，也就是优先级高的可以「插队」；
* *SCHED\_RR*：对于相同优先级的任务，轮流着运行，每个任务都有一定的时间片，当用完时间片的任务会被放到队列尾部，以保证相同优先级任务的公平性，但是高优先级的任务依然可以抢占低优先级的任务；

Fair 调度类是应用于普通任务，都是由 CFS 调度器管理的，分为两种调度策略：

* *SCHED\_NORMAL*：普通任务使用的调度策略；
* *SCHED\_BATCH*：后台任务的调度策略，不和终端进行交互，因此在不影响其他需要交互的任务，可以适当降低它的优先级。

CFS：**完全公平调度（Completely Fair Scheduling）**

这个算法的理念是想让分配给每个任务的 CPU 时间是一样，于是它为每个任务安排一个虚**拟运行时间 vruntime**，如果一个任务在运行，其运行的越久，该任务的 vruntime 自然就会越大，而没有被运行的任务，vruntime 是不会变化的。

那么，**在 CFS 算法调度的时候，会优先选择 vruntime 少的任务**，以保证每个任务的公平性。

普通任务之间还是有优先级区分的，所以在计算虚拟运行时间 vruntime 还要考虑普通任务的**权重值**

### CPU 运行队列

每个 CPU 都有自己的**运行队列（Run Queue, rq）**，包含三个运行队列，Deadline 运行队列 dl\_rq、实时任务运行队列 rt\_rq 和 CFS 运行队列 cfs\_rq，其中 cfs\_rq 是用红黑树来描述的，按 vruntime 大小来排序的，最左侧的叶子节点，就是下次会被调度的任务。

这几种调度类是有优先级的，优先级如下：**Deadline > Realtime > Fair**，这意味着 Linux 选择下一个任务执行的时候，会按照此优先级顺序进行选择，也就是说先从 dl\_rq 里选择任务，然后从 rt\_rq 里选择任务，最后从 cfs\_rq 里选择任务。因此，**实时任务总是会比普通任务优先被执行**。

### 调整优先级

普通任务之间：调整nice值

普通任务也可以改变调度策略变成实时任务

# 2.6 什么是软中断？

## 中断是什么？

在计算机中，中断是系统用来响应**硬件设备**请求的一种机制，操作系统收到硬件的中断请求，会打断正在执行的进程，然后调用内核中的中断处理程序来响应请求。

中断是一种**异步**的事件处理机制，可以提高系统的并发处理能力。操作系统收到了中断请求，会打断其他进程的运行，所以**中断请求的响应程序，也就是中断处理程序，要尽可能快的执行完，这样可以减少对正常进程运行调度的影响。**

中断处理程序在响应中断时，可能还会「**临时关闭中断**」，这意味着，如果**当前中断处理程序没有执行完之前，系统中其他的中断请求都无法被响应**，也就说中断有可能会丢失，所以中断处理程序要短且快。

## 什么是软中断？

Linux 系统**为了解决中断处理程序执行过长和中断丢失的问题，将中断过程分成了两个阶段，分别是「上半部和下半部分」**。

* **上半部用来快速处理中断**，一般会**暂时关闭中断请求**，主要负责处理跟硬件紧密相关或者时间敏感的事情。**也就是硬中断**，主要是负责耗时短的工作，特点是**快速执行**。
* **下半部用来延迟处理上半部未完成的工作**，一般以「内核线程」的方式运行。**也就是软中断**，主要是负责上半部未完成的工作，通常都是耗时比较长的事情，特点是**延迟执行**；

举例：

网卡收到网络包后，通过 DMA 方式将接收到的数据写入内存，接着会通过**硬件中断**通知内核有新的数据到了，于是内核就会调用对应的中断处理程序来处理该事件，这个事件的处理也是会分成上半部和下半部。

上部分要做的事情很少，会先禁止网卡中断，避免频繁硬中断，而降低内核的工作效率。接着，内核会触发一个**软中断**，把一些处理比较耗时且复杂的事情，交给「软中断处理程序」去做，也就是中断的下半部，其主要是需要从内存中找到网络数据，再按照网络协议栈，对网络数据进行逐层解析和处理，最后把数据送给应用程序。

还有一个区别，**硬中断（上半部）是会打断 CPU 正在执行的任务，然后立即执行中断处理程序，而软中断（下半部）是以内核线程的方式执行**，并且每一个 CPU 都对应一个软中断内核线程，名字通常为「ksoftirqd/CPU 编号」，比如 0 号 CPU 对应的软中断内核线程的名字是 ksoftirqd/0。软中断不只是包括硬件设备中断处理程序的下半部，一些内核自定义事件也属于软中断，比如**内核调度**等、**RCU 锁**（内核里常用的一种锁）等。

# 3.1 Linux 内核 vs Windows 内核

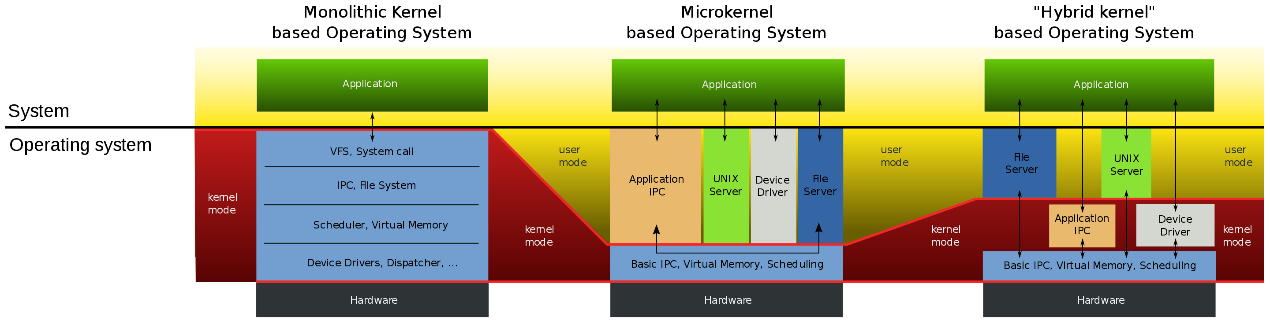
## Linux 的设计

Linux 内核设计的理念主要有这几个点：

* MultiTask，多任务
* SMP，对称多处理
* ELF，可执行文件链接格式
* Monolithic Kernel，宏内核

**宏内核**的特征是系统内核的所有模块，比如进程调度、内存管理、文件系统、设备驱动等，都运行在内核态。

**微内核**架构的内核只保留最基本的能力，比如进程调度、虚拟机内存、中断等，把一些应用放到了用户空间，比如**驱动程序、文件系统**等。这样服务与服务之间是隔离的，单个服务出现故障或者完全攻击，也不会导致整个操作系统挂掉，提高了操作系统的稳定性和可靠性。



Linux 的内核设计是采用了**宏内核**，Window 的内核设计则是采用了**混合内核**

鸿蒙是**微内核**

# 四、内存管理

# 4.1 为什么要有虚拟内存？

## 虚拟内存

**操作系统会提供一种机制，将不同进程的虚拟地址和不同内存的物理地址映射起来。**

如果程序要访问虚拟地址的时候，**由操作系统转换成不同的物理地址**，这样不同的进程运行的时候，写入的是不同的物理地址，这样就不会冲突了。

于是，这里就引出了两种地址的概念：

* 我们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*）。

虚拟地址会通过 CPU 芯片中的**内存管理单元（MMU）**的映射关系，来转换变成物理地址

操作系统是如何管理虚拟地址与物理地址之间的关系？

**①内存分段 ②内存分页**

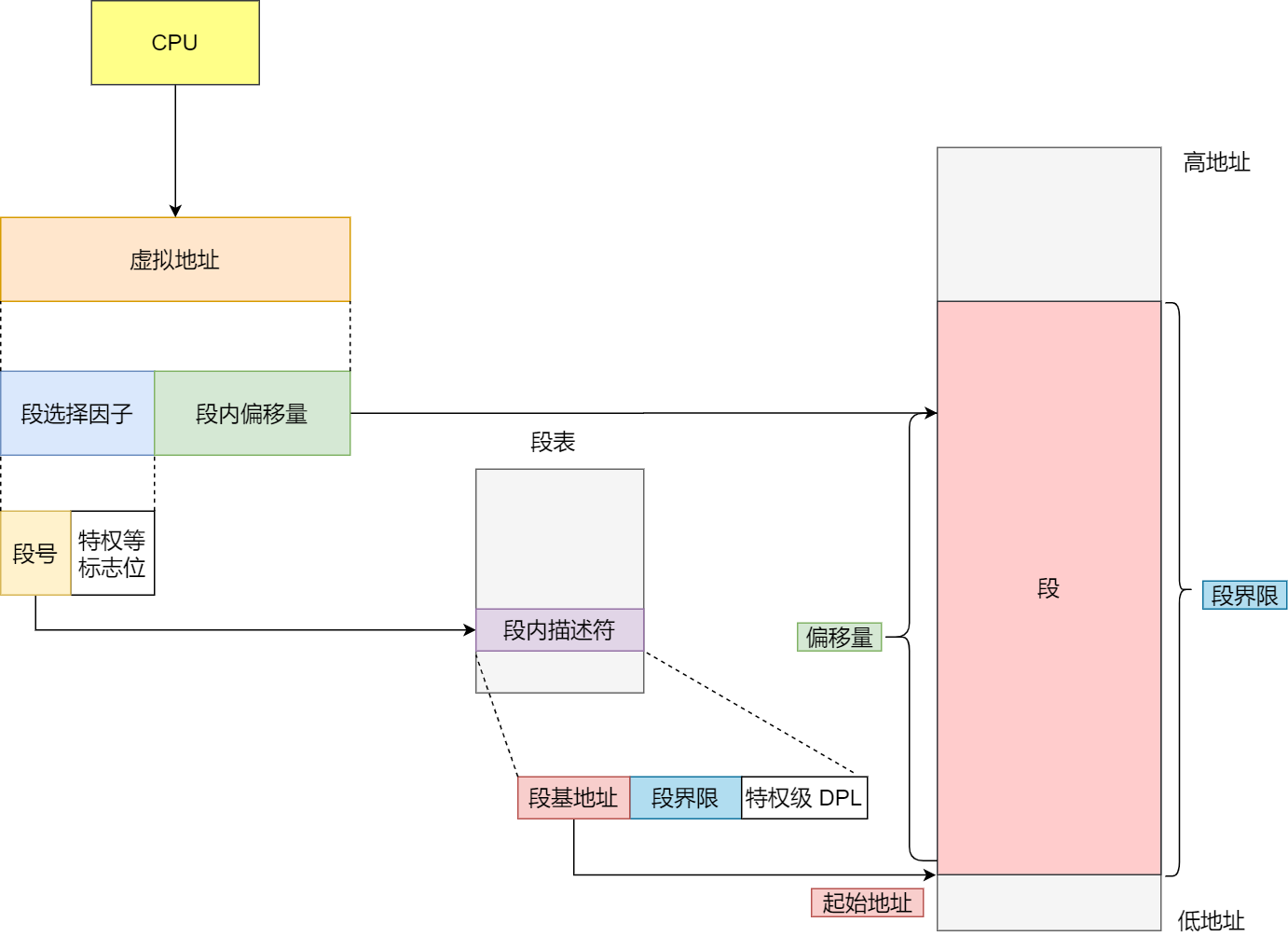
## 虚拟内存有什么作用：

* 第一，虚拟内存可以**使进程的运行内存超过物理内存大小**，因为程序运行符合局部性原理，CPU 访问内存会有很明显的重复访问的倾向性，对于那些没有被经常使用到的内存，我们可以把它换出到物理内存之外，比如硬盘上的 swap 区域。
* 第二，由于每个进程都有自己的页表，所以每个进程的**虚拟内存空间就是相互独立的**。进程也没有办法访问其他进程的页表，所以这些页表是私有的，这就解决了多进程之间地址冲突的问题。
* 第三，页表里的**页表项中除了物理地址之外，还有一些标记属性的比特，比如控制一个页的读写权限，标记该页是否存在等**。在内存访问方面，操作系统提供了更好的安全性。

## 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：**段选择因子**和**段内偏移量**。



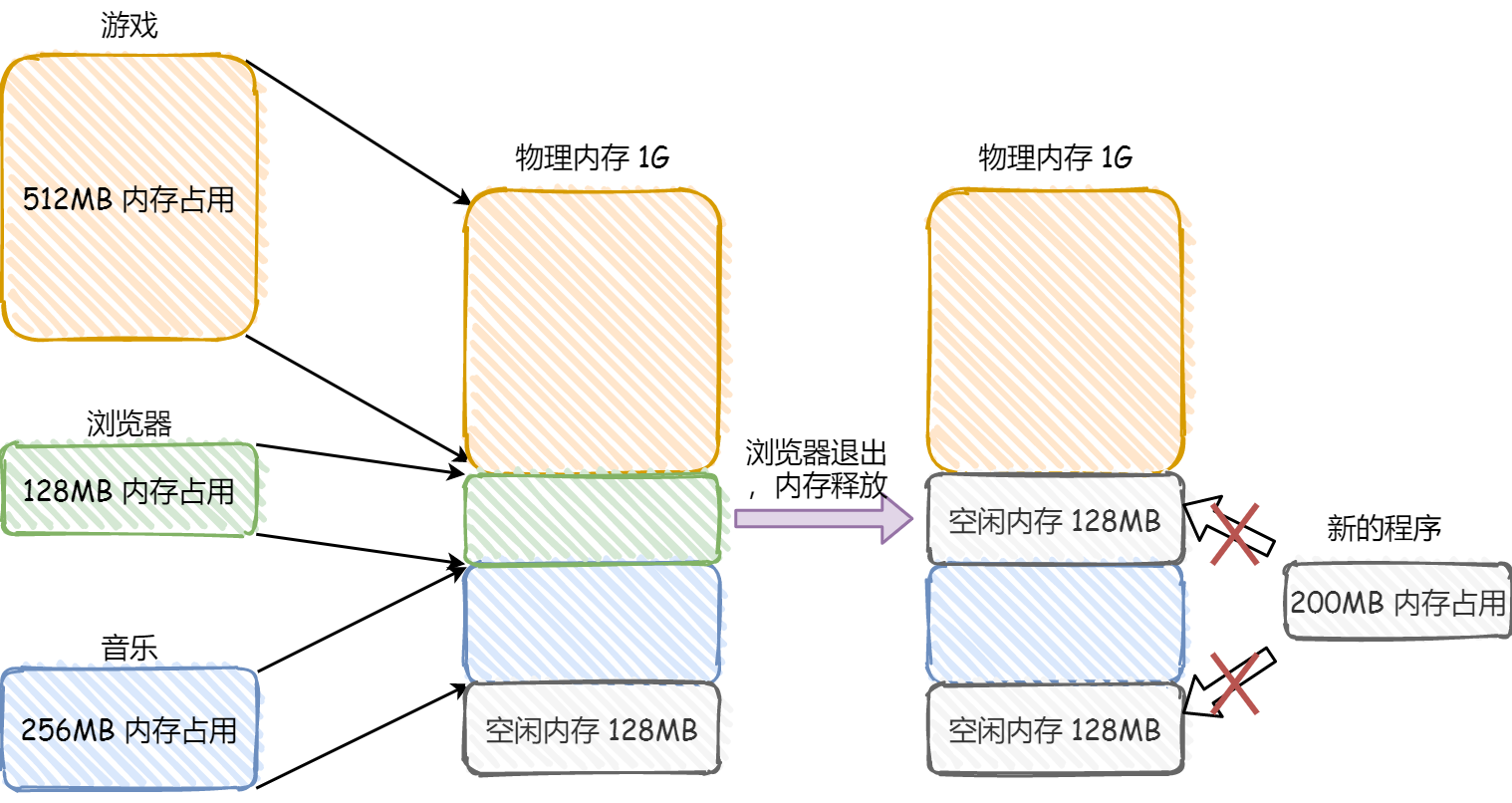
缺点：

* 第一个就是**内存碎片**的问题。
* 第二个就是**内存交换的效率低**的问题。

内存分段会出现内存碎片吗？

内存碎片主要分为，**内部**内存碎片和**外部**内存碎片。

内存分段管理可以做到段根据实际需求分配内存，所以有多少需求就分配多大的段，所以**不会出现内部内存碎片**。但是由于每个段的长度不固定，所以多个段未必能恰好使用所有的内存空间，会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，所以**会出现外部内存碎片**的问题。



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。

在 Linux 系统里，也就是我们常看到的 Swap 空间，这块空间是从硬盘划分出来的，用于内存与硬盘的空间交换。

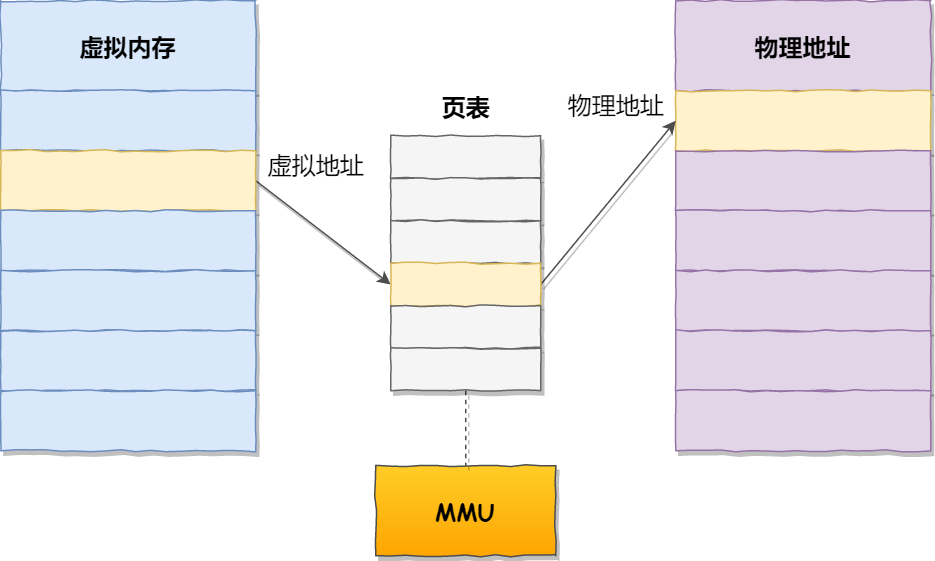
分段为什么会导致内存交换效率低的问题？

**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

**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。

虚拟地址与物理地址之间通过**页表**来映射，如下图：



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。

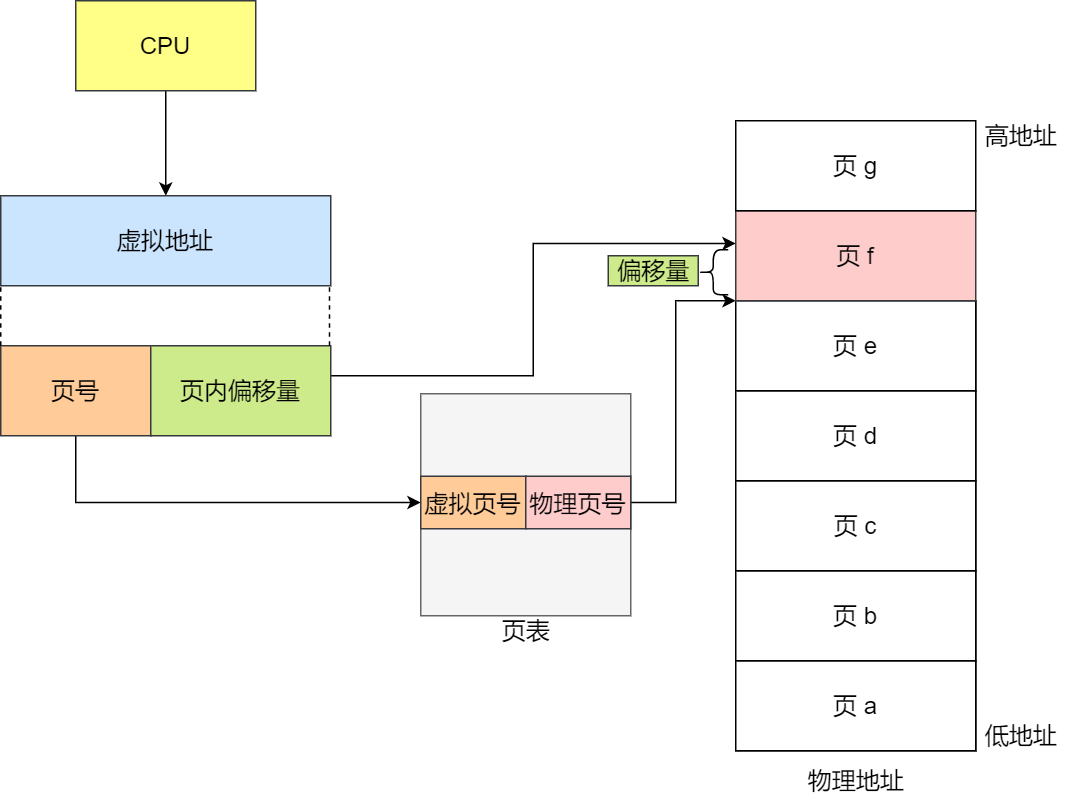
**内存分页时页与页之间是紧密排列的，所以不会有外部碎片。**但是，因为内存分页机制分配内存的**最小单位是一页**，即使程序不足一页大小，我们最少只能分配一个页，所以页内会出现内存浪费，所以针对**内存分页机制会有内部内存碎片**的现象。

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。**

在进行虚拟内存和物理内存的页之间的映射之后，并不真的把页加载到物理内存里，而是**只有在程序运行中，需要用到对应虚拟内存页里面的指令和数据时，再加载到物理内存里面去。**

分页机制下，虚拟地址和物理地址是如何映射的？

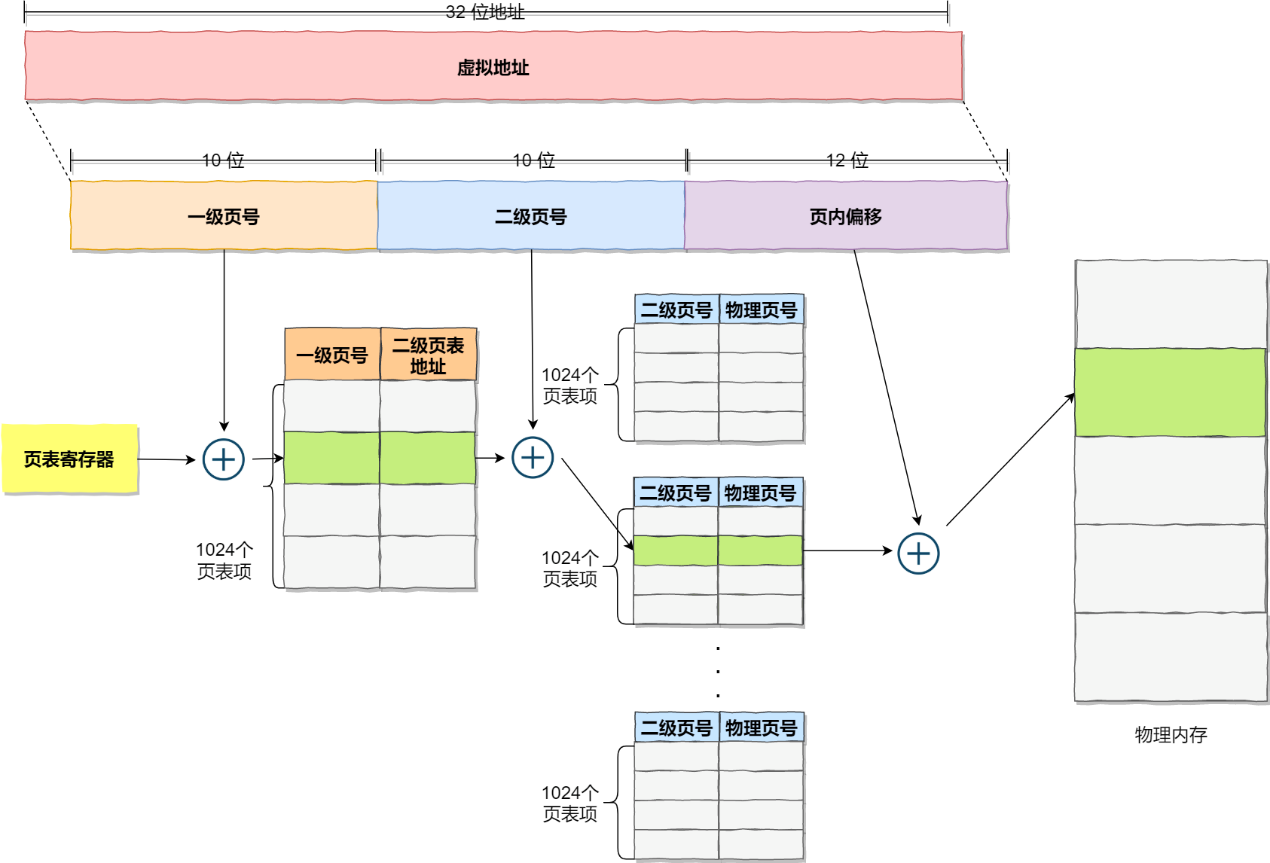
分页机制下，虚拟地址分为两部分，**页号**和**页内偏移**。页号作为页表的索引，**页表**包含物理页每页所在**物理内存的基地址**，这个基地址与页内偏移的组合就形成了物理内存地址，见下图。



简单的分页的缺陷

在 32 位的环境下，虚拟地址空间共有 4GB，假设一个页的大小是 4KB（2^12），那么就需要大约 100 万 （2^20）个页，每个「页表项」需要 4 个字节大小来存储，那么整个 4GB 空间的映射就需要有 4MB 的内存来存储页表。每个进程都是有自己的虚拟地址空间的，也就说都有自己的页表。那么100 个进程就需要 400MB 的内存来存储页表

### 多级页表

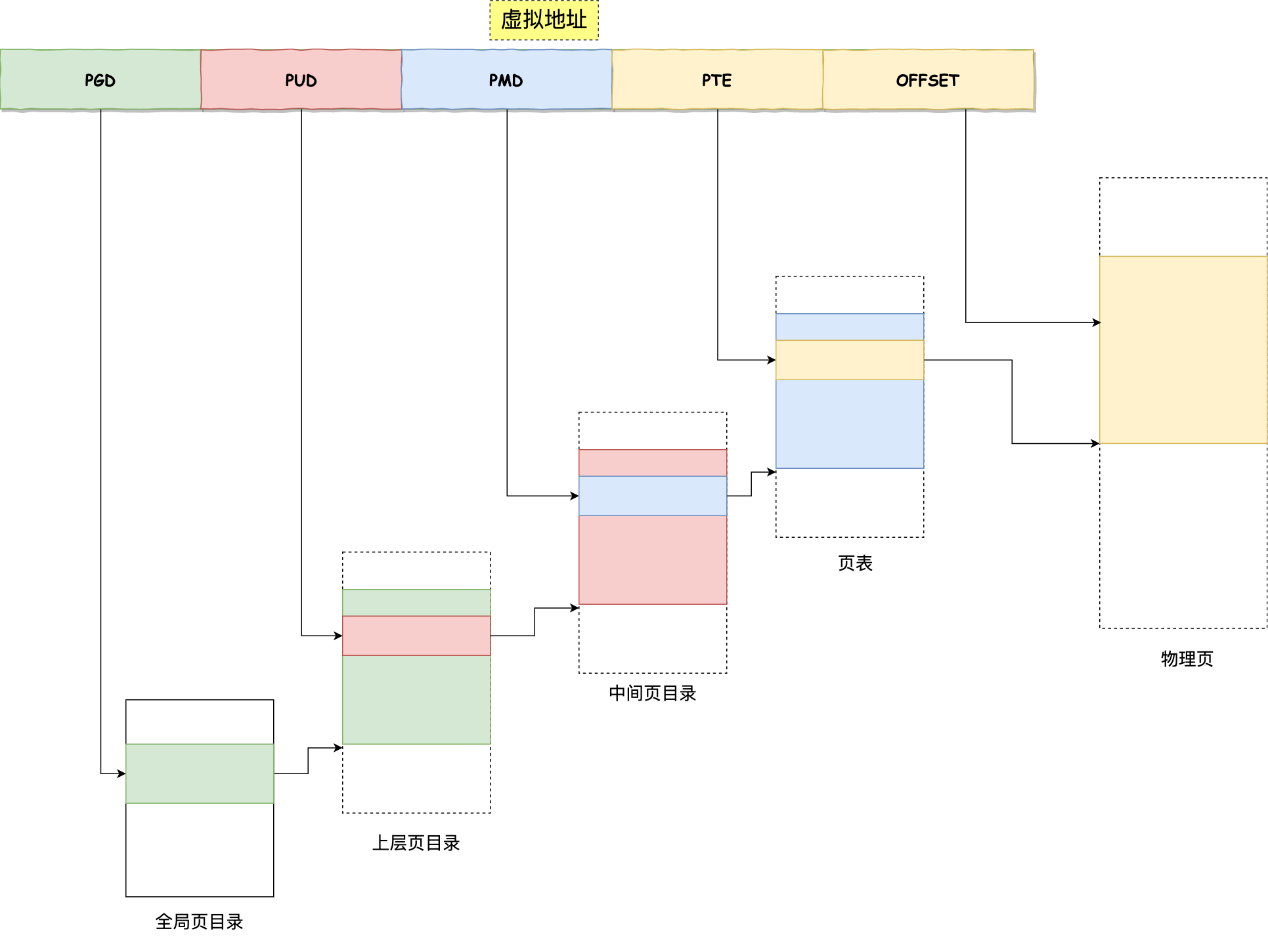


**如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。**假设只有 20% 的一级页表项被用到了，那么页表占用的内存空间就只有 4KB（一级页表） + 20% \* 4MB（二级页表）= 0.804MB

为什么不分级的页表就做不到这样节约内存

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**（第一级）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）。

64位系统是四级页表：



### TLB

通常称为**页表缓存、转址旁路缓存、快表**等。

把最常访问的几个页表项存储到访问速度更快的硬件

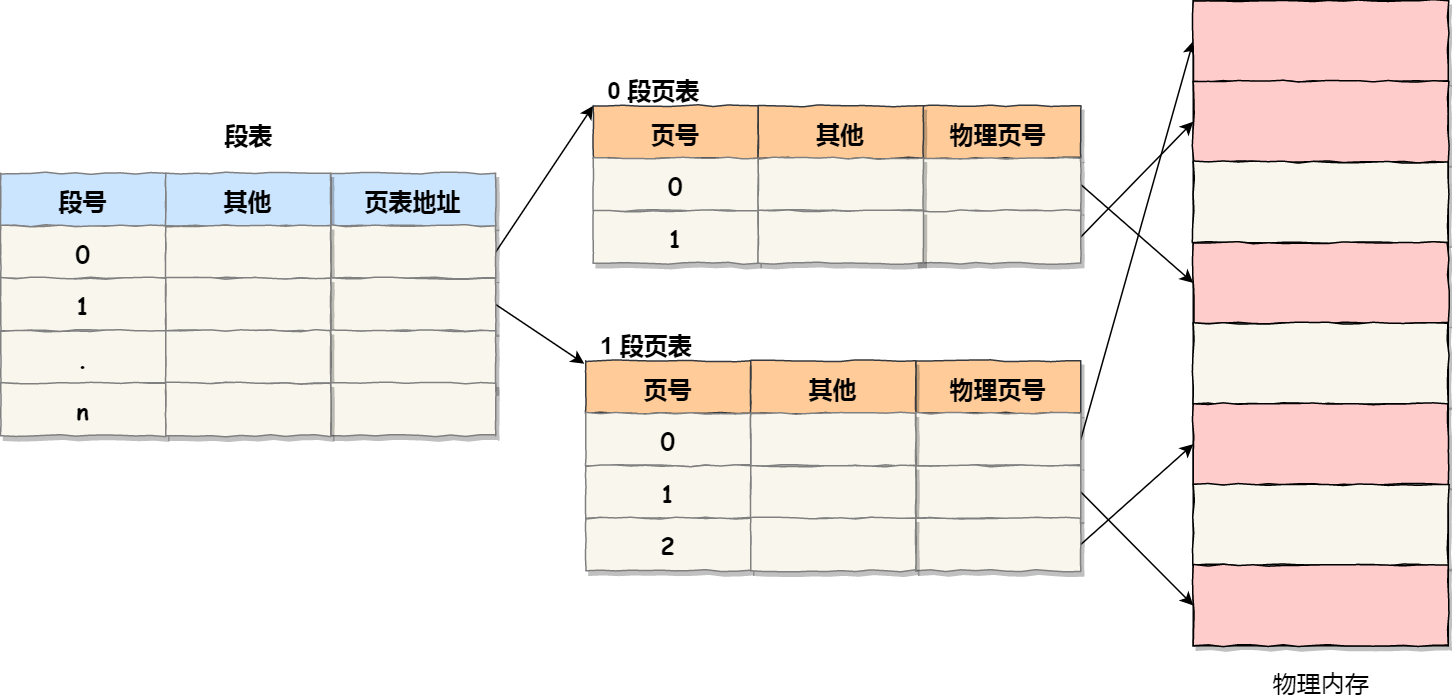
有了 TLB 后，那么 CPU 在寻址时，**会先查 TLB，如果没找到，才会继续查常规的页表**。TLB 的命中率其实是很高的，因为程序最常访问的页就那么几个。

## 段页式内存管理

段页式内存管理实现的方式：

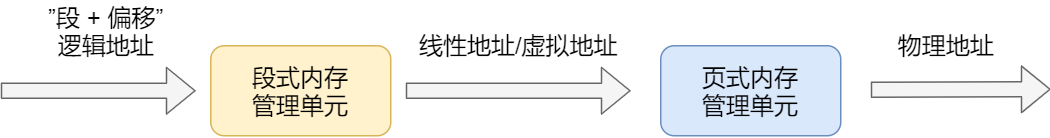
* 先将程序划分为多个有逻辑意义的段，也就是前面提到的分段机制；
* 接着再把每个段划分为多个页，也就是对分段划分出来的连续空间，再划分固定大小的页；

地址结构由**段号、段内页号和页内位移**三部分组成



## Linux 内存布局

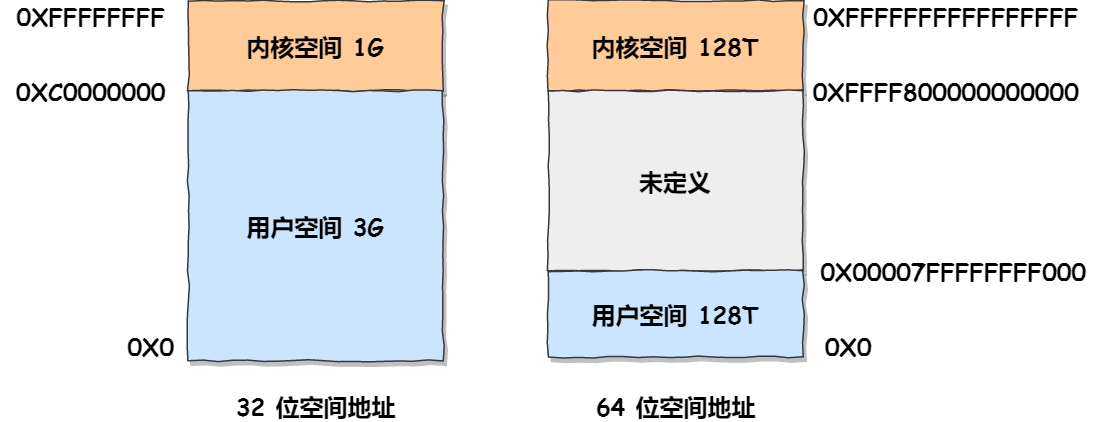
由于早期处理器的原因，**Linux 内存主要采用的是页式内存管理，但同时也不可避免地涉及了段机制**。



事实上，**Linux 内核所采取的办法是使段式映射的过程实际上不起什么作用。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

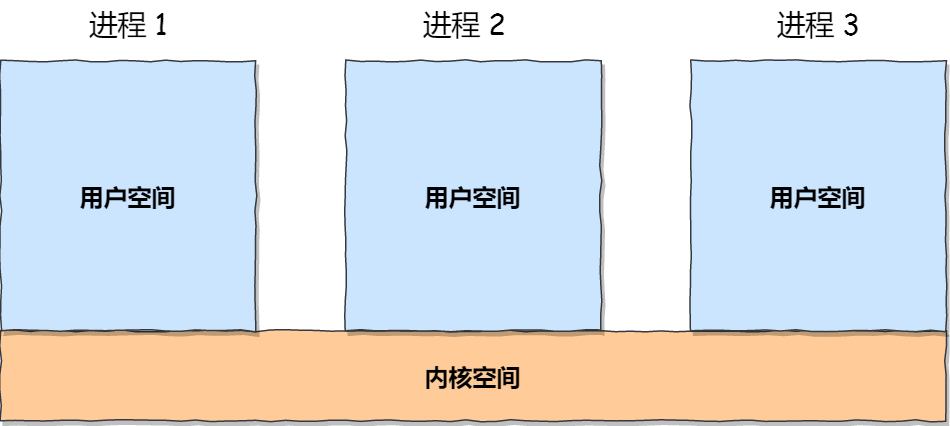
Linux 操作系统中，虚拟地址空间的内部又被分为**内核空间和用户空间**两部分，不同位数的系统，地址空间的范围也不同。比如最常见的 32 位和 64 位系统，如下所示：



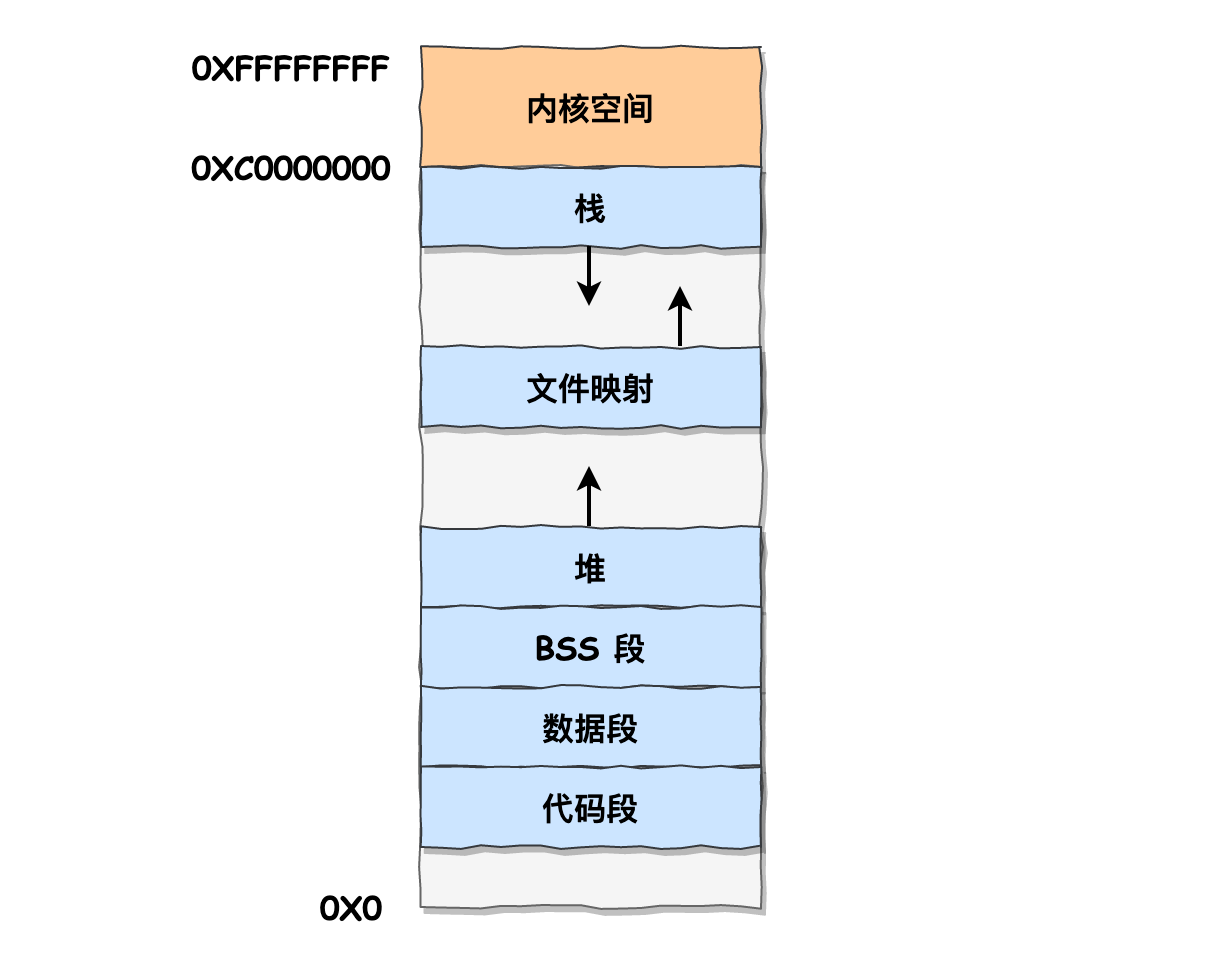
**内核空间与用户空间的区别：**

* 进程在用户态时，只能访问用户空间内存；
* 只有进入内核态后，才可以访问内核空间的内存；

**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**。



即**每个进程的用户空间不同，但内核空间相同**



**用户空间**内存，从**低到高**分别是 6 种不同的内存段：

* 代码段，包括二进制可执行代码；
* 数据段，包括已初始化的静态常量和全局变量；
* BSS 段，包括未初始化的静态变量和全局变量；
* 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 (opens new window)](http://lishiwen4.github.io/linux/linux-process-memory-location)）；
* 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；

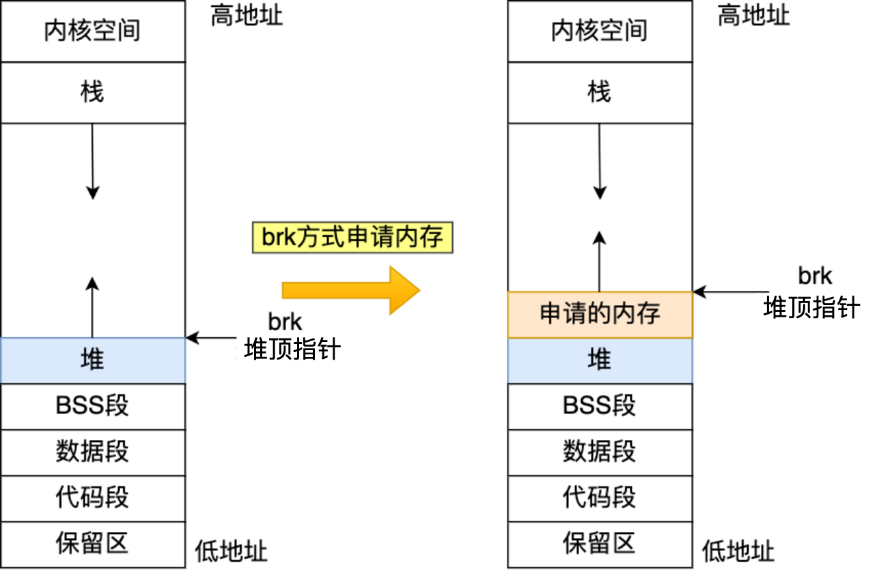
代码段下面还有一段内存空间的（灰色部分），这一块区域是「**保留区**」，之所以要有保留区这是因为在大多数的系统里，我们认为**比较小数值的地址不是一个合法地址**

# 4.2 malloc 是如何分配内存的？

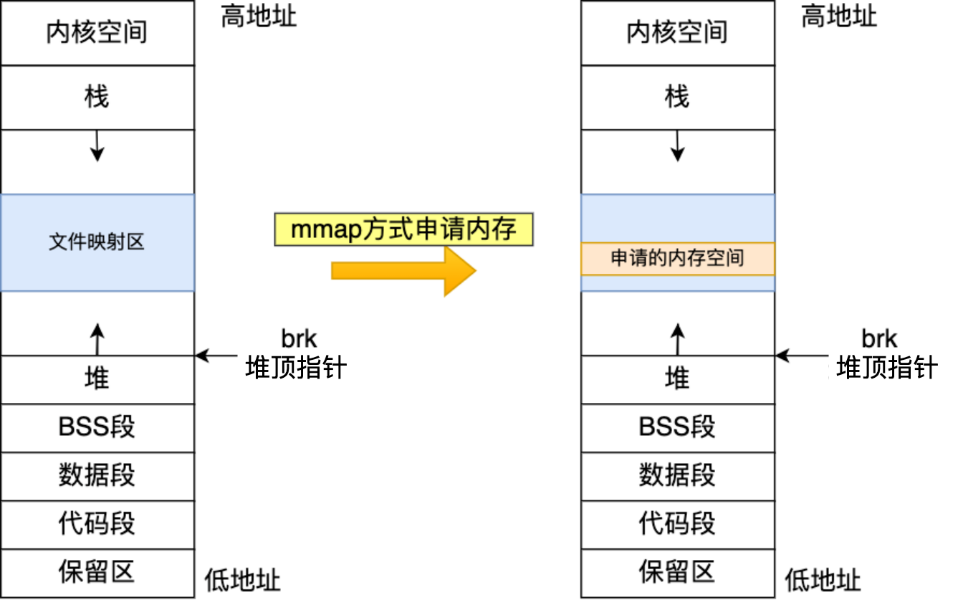
malloc() 并不是系统调用，而是 C 库里的函数，用于动态分配内存。

malloc 申请内存的时候，会有两种方式向操作系统申请堆内存。

* 方式一：通过 brk() 系统调用从堆分配内存



* 方式二：通过 mmap() 系统调用在文件映射区域分配内存；



什么场景下 malloc() 会通过 brk() 分配内存？又是什么场景下通过 mmap() 分配内存？

* 如果用户分配的内存**小于 128 KB**，则通过 brk() 申请内存；
* 如果用户分配的内存**大于 128 KB**，则通过 mmap() 申请内存；

## malloc() 分配的是哪种内存吗？

**虚拟内存**

如果分配后的虚拟内存**没有被访问的话，虚拟内存是不会映射到物理内存的**，这样就不会占用物理内存了。

## malloc(1) 会分配多大的虚拟内存？

malloc() 在分配内存的时候，并不是老老实实按用户预期申请的字节数来分配内存空间大小，而是**会预分配更大的空间作为内存池**。

以 malloc 默认的内存管理器（Ptmalloc2）为例，会分配**132K 字节的内存**

## free 释放内存，会归还给操作系统吗？

* malloc 通过 **brk() 方式**申请的内存，free 释放内存的时候，**并不会把内存归还给操作系统，而是缓存在 malloc 的内存池中，待下次使用**；
* malloc 通过**mmap() 方式**申请的内存，free 释放内存的时候，**会把内存归还给操作系统，内存得到真正的释放**。

## 为什么不全部使用 mmap 来分配内存？

① 都用 mmap 来分配内存，等于**每次都要执行系统调用**。执行系统调用是要进入内核态的，然后在回到用户态，运行态的切换会**耗费不少时间**。

② mmap 分配的内存每次释放的时候，都会归还给操作系统，于是**每次 mmap 分配的虚拟地址都是缺页状态的**，然后在第一次访问该虚拟地址的时候，就会**触发缺页中断**。

通过 brk() 系统调用在堆空间申请内存的时候，由于堆空间是连续的，所以直接预分配更大的内存来作为内存池，当内存释放的时候，就缓存在内存池中。

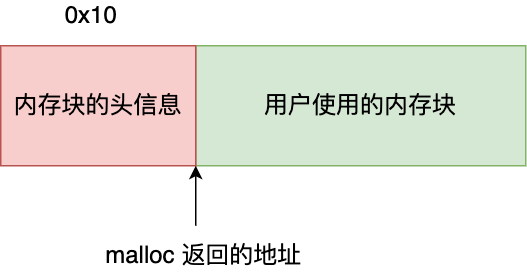
**等下次在申请内存的时候，就直接从内存池取出对应的内存块就行了，而且可能这个内存块的虚拟地址与物理地址的映射关系还存在，这样不仅减少了系统调用的次数，也减少了缺页中断的次数，这将大大降低 CPU 的消耗**。

## 为什么不全部使用 brk 来分配？

对于小块内存，堆内将产生越来越多不可用的**外部碎片**，导致“内存泄露”（**一种形式上的、不可被valgrind检测的内存泄漏**）。

## free() 函数只传入一个内存地址，为什么能知道要释放多大的内存？

 malloc 返回给用户态的内存起始地址比进程的堆空间起始地址多了 16 字节。这个多出来的 16 字节就是**保存了该内存块的描述信息**，比如有该**内存块的大小**。



# 4.3 内存满了，会发生什么？

## 内存分配的过程是怎样的？

应用程序通过 malloc 函数申请内存的时候，**实际上申请的是虚拟内存**，此时并不会分配物理内存。当应用程序**读写了这块虚拟内存**，CPU 就会去访问这个虚拟内存， 这时会发现这个虚拟内存没有映射到物理内存， CPU 就会产生**缺页中断**，进程会从用户态切换到内核态，并将缺页中断交给内核的 Page Fault Handler （缺页中断函数）处理。

缺页中断处理函数会看是否有空闲的物理内存，如果有，就直接分配物理内存，并建立虚拟内存与物理内存之间的映射关系。

如果没有空闲的物理内存，那么内核就会开始进行**回收内存**的工作，回收的方式主要是两种：**直接内存回收**和**后台内存回收**。

* **后台内存回收**（kswapd）：在物理内存紧张的时候，会唤醒 kswapd 内核线程来回收内存，这个回收内存的过程**异步**的，不会阻塞进程的执行。
* **直接内存回收**（direct reclaim）：如果后台异步回收跟不上进程内存申请的速度，就会开始直接回收，这个回收内存的过程是**同步**的，会阻塞进程的执行。

如果直接内存回收后，空闲的物理内存仍然无法满足此次物理内存的申请，那么内核就会**触发 OOM （Out of Memory）机制**。

OOM Killer 机制会根据算法选择一个占用物理内存较高的进程，然后将其杀死，以便释放内存资源，如果物理内存依然不足，OOM Killer 会继续杀死占用物理内存较高的进程，直到释放足够的内存位置。

## 哪些内存可以被回收？

* **文件页**（File-backed Page）：内核缓存的磁盘数据（**Buffer**）和内核缓存的文件数据（**Cache**）都叫作文件页。大部分文件页，都可以直接释放内存，以后有需要时，再从磁盘重新读取就可以了。而那些被应用程序修改过，并且暂时还没写入磁盘的数据（也就是脏页），就得先写入磁盘，然后才能进行内存释放。所以，**回收干净页的方式是直接释放内存，回收脏页的方式是先写回磁盘后再释放内存**。
* **匿名页**（Anonymous Page）：这部分内存没有实际载体，不像文件缓存有硬盘文件这样一个载体，比如堆、栈数据等。这部分内存很可能还要再次被访问，所以不能直接释放内存，它们**回收的方式是通过 Linux 的 Swap 机制**，Swap 会把不常访问的内存先写到磁盘中，然后释放这些内存，给其他更需要的进程使用。再次访问这些内存时，重新从磁盘读入内存就可以了。

文件页和匿名页的回收都是基于 **LRU 算法**

* 文件页的回收：对于干净页是直接释放内存，这个操作不会影响性能，而对于脏页会先写回到磁盘再释放内存，这个操作会发生磁盘 I/O 的，这个操作是会影响系统性能的。
* 匿名页的回收：如果开启了 Swap 机制，那么 Swap 机制会将不常访问的匿名页换出到磁盘中，下次访问时，再从磁盘换入到内存中，这个操作是会影响系统性能的。

针对**回收内存导致的性能影响**，常见的解决方式：

### 调整文件页和匿名页的回收倾向

Linux 提供了一个 /proc/sys/vm/swappiness 选项，用来调整文件页和匿名页的回收倾向。

swappiness 的范围是 0-100，数值越大，越积极使用 Swap，也就是更倾向于回收匿名页；数值越小，越消极使用 Swap，也就是更倾向于回收文件页。

建议 swappiness 设置为 0（默认值是 60），这样在回收内存的时候，会更倾向于文件页的回收，但是**并不代表不会回收匿名页**。

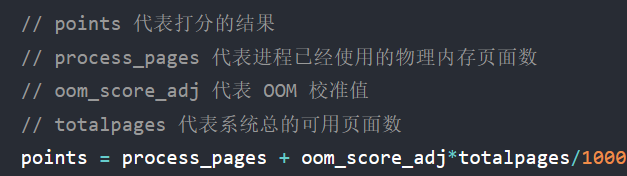
### 尽早触发 kswapd 内核线程异步回收内存

## 如何保护一个进程不被 OOM 杀掉呢？

Linux 内核里有一个 **oom\_badness() 函数**，它会把系统中可以被杀掉的进程扫描一遍，并对每个进程打分，得分最高的进程就会被首先杀掉。

进程得分的结果受下面这两个方面影响：

* 第一，进程已经使用的物理内存页面数。
* 第二，每个进程的 OOM 校准值 oom\_score\_adj。它是可以通过 /proc/[pid]/oom\_score\_adj 来配置的。我们可以在设置 -1000 到 1000 之间的任意一个数值，调整进程被 OOM Kill 的几率。



* 如果你不想某个进程被首先杀掉，那你可以调整该进程的 oom\_score\_adj，从而改变这个进程的得分结果，降低该进程被 OOM 杀死的概率。
* 如果你想某个进程**无论如何都不能被杀掉，那你可以将 oom\_score\_adj 配置为 -1000。**

我们最好将一些很重要的系统服务的 oom\_score\_adj 配置为 -1000，比如 sshd，因为这些系统服务一旦被杀掉，我们就很难再登陆进系统了。

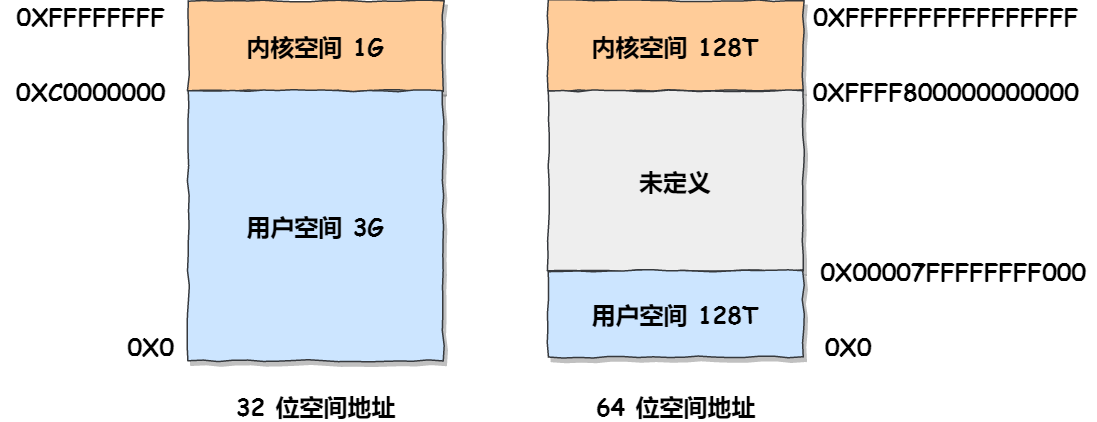
但是，不建议将我们自己的业务程序的 oom\_score\_adj 设置为 -1000，因为业务程序一旦发生了内存泄漏，而它又不能被杀掉，这就会导致随着它的内存开销变大，OOM killer 不停地被唤醒，从而把其他进程一个个给杀掉。

# 4.4 在 4GB 物理内存的机器上，申请 8G 内存会怎么样？

这个问题要考虑三个前置条件：

* **操作系统是 32 位的，还是 64 位的？**
* **申请完 8G 内存后会不会被使用？**
* **操作系统有没有使用 Swap 机制？**

## 操作系统虚拟内存大小



 32 位操作系统，进程最多只能申请 3 GB 大小的虚拟内存空间，所以进程申请 8GB 内存的话，在申请虚拟内存阶段就会失败（报错cannot allocate memory，也就是无法申请内存失败）。

64 位操作系统，进程可以使用 128 TB 大小的虚拟内存空间，所以进程申请 8GB 内存是没问题的，因为进程申请内存是申请虚拟内存，只要不读写（能否读写要看是否开启Swap）这个虚拟内存，操作系统就不会分配物理内存。

## Swap 机制的作用

如果申请物理内存大小超过了空闲物理内存大小，并进行访问，就要看操作系统有没有开启 Swap 机制：

* 如果没有开启 Swap 机制，程序就会直接 OOM；
* 如果有开启 Swap 机制，程序可以正常运行。

Swap 就是把一块磁盘空间或者本地文件，当成内存来使用，它包含换出和换入两个过程：

* **换出（Swap Out）** ，是把进程暂时不用的内存数据存储到磁盘中，并释放这些数据占用的内存；
* **换入（Swap In）**，是在进程再次访问这些内存的时候，把它们从磁盘读到内存中来；

频繁地读写硬盘，会显著降低操作系统的运行速率，这也是 Swap 的弊端。

Linux 中的 Swap 机制会在内存不足和内存闲置的场景下触发：

* **内存不足**：当系统需要的内存超过了可用的物理内存时，内核会将内存中不常使用的内存页交换到磁盘上为当前进程让出内存，保证正在执行的进程的可用性，这个内存回收的过程是强制的直接内存回收（Direct Page Reclaim）。直接内存回收是同步的过程，会阻塞当前申请内存的进程。
* **内存闲置**：应用程序在启动阶段使用的大量内存在启动后往往都不会使用，通过后台运行的守护进程（kSwapd），我们可以将这部分只使用一次的内存交换到磁盘上为其他内存的申请预留空间。kSwapd 是 Linux 负责页面置换（Page replacement）的守护进程，它也是负责交换闲置内存的主要进程，它会在[空闲内存低于一定水位 (opens new window)](https://xiaolincoding.com/os/3_memory/mem_reclaim.html#%E5%B0%BD%E6%97%A9%E8%A7%A6%E5%8F%91-kSwapd-%E5%86%85%E6%A0%B8%E7%BA%BF%E7%A8%8B%E5%BC%82%E6%AD%A5%E5%9B%9E%E6%94%B6%E5%86%85%E5%AD%98)时，回收内存页中的空闲内存保证系统中的其他进程可以尽快获得申请的内存。kSwapd 是后台进程，所以回收内存的过程是异步的，不会阻塞当前申请内存的进程。

# 4.5 如何避免预读失效和缓存污染的问题？

## 传统 LRU 是如何管理内存数据的？

LRU 算法一般是用「链表」作为数据结构来实现的，链表头部的数据是最近使用的，而链表末尾的数据是最久没被使用的。当空间不够了，就淘汰最久没被使用的节点，也就是链表末尾的数据，从而腾出内存空间。

**节点的基本数据单位都是页**

传统的 LRU 算法的实现思路：

* 当访问的页在内存里，就直接把该页对应的 LRU 链表节点移动到链表的头部。
* 当访问的页不在内存里，除了要把该页放入到 LRU 链表的头部，还要淘汰 LRU 链表末尾的页。

传统的 LRU 算法无法避免下面这两个问题：

* **预读失效**导致缓存命中率下降；
* **缓存污染**导致缓存命中率下降；

### 什么是预读机制？

* 应用程序**只想读取**磁盘上文件 A 的 offset 为 **0-3KB 范围内的数据**，由于磁盘的基本读写单位（页）为 block（4KB），于是操作系统至少会读 0-4KB 的内容，这恰好可以在一个 page 中装下。
* 但是操作系统出于空间局部性原理（靠近当前被访问数据的数据，在未来很大概率会被访问到），会选择将磁盘块 offset [4KB,8KB)、[8KB,12KB) 以及 [12KB,16KB) 都加载到内存，于是额外在内存中申请了 3 个 page；

预读机制带来的好处就是**减少了磁盘 I/O 次数，提高系统磁盘 I/O 吞吐量**。

### 预读失效会带来什么问题？

如果**这些被提前加载进来的页，并没有被访问**，相当于这个预读工作是白做了，这个就是**预读失效**。如果使用**传统的 LRU 算法**，就会把「预读页」放到 LRU 链表头部，而当内存空间不够的时候，还需要把末尾的页淘汰掉。如果这些「预读页」如果一直不会被访问到，就会出现一个很奇怪的问题，**不会被访问的预读页却占用了 LRU 链表前排的位置，而末尾淘汰的页，可能是热点数据，这样就大大降低了缓存命中率** 。

### 如何避免预读失效造成的影响？

**让预读页停留在内存里的时间要尽可能的短，让真正被访问的页才移动到 LRU 链表的头部，从而保证真正被读取的热数据留在内存里的时间尽可能长**。

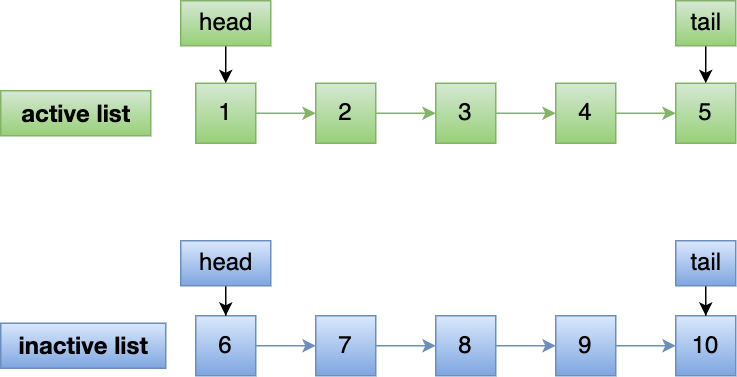
设计思想：**将数据分为了冷数据和热数据，然后分别进行 LRU 算法**。

Linux 是如何避免预读失效带来的影响？

Linux 操作系统实现了**两个LRU链表**：**活跃 LRU 链表（active\_list）和非活跃 LRU 链表（inactive\_list）**；

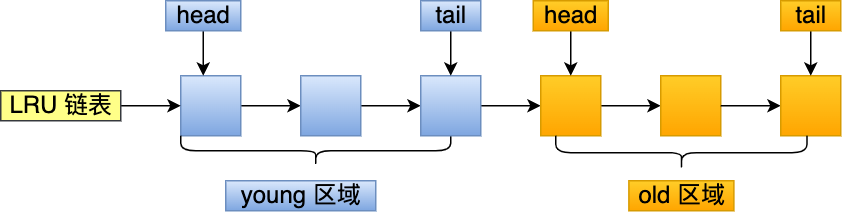
* **active list** 活跃内存页链表，这里存放的是**最近被访问过（活跃）的内存页**；
* **inactive list** 不活跃内存页链表，这里存放的是**很少被访问（非活跃）的内存页**；

有了这两个 LRU 链表后，**预读页就只需要加入到 inactive list 区域的头部，当页被真正访问的时候，才将页插入 active list 的头部**。如果预读的页一直没有被访问，就会从 inactive list 移除，这样就不会影响 active list 中的热点数据。 预读页被访问时，active list 末尾的页，会被**降级**到 inactive list 作为 inactive list 的头部，这个过程并不会有数据被淘汰。



MySQL 是如何避免预读失效带来的影响？

MySQL 的 Innodb 存储引擎是在**一个 LRU 链表上划分2个区域**：**young 区域 和 old 区域**。过程和Linux两个链表类似。



## 缓存污染，怎么办？

### 什么是缓存污染？

使用「只要数据被访问一次，就将数据加入到活跃 LRU 链表头部（或者 young 区域）」这种方式，**存在缓存污染的问题**。

当我们在批量读取数据的时候，由于数据被访问了一次，这些大量数据都会被加入到「活跃 LRU 链表」里，然后之前缓存在活跃 LRU 链表（或者 young 区域）里的热点数据全部都被淘汰了，**如果这些大量的数据在很长一段时间都不会被访问的话，那么整个活跃 LRU 链表（或者 young 区域）就被污染了**。

### 怎么避免缓存污染造成的影响？

**缓存污染的原因**：LRU 算法只要数据被访问一次，就将数据加入活跃 LRU 链表（或者 young 区域），**这种 LRU 算法进入活跃 LRU 链表的门槛太低了**。

解决方法：**提高进入到活跃 LRU 链表（或者 young 区域）的门槛**

Linux 操作系统和 MySQL Innodb 存储引擎分别是这样提高门槛的：

* **Linux 操作系统**：在内存页被**访问第二次**的时候，才将页从 inactive list 升级到 active list 里。
* **MySQL Innodb**：在内存页被**访问第二次**的时候，还要进行**停留在 old 区域的时间判断**：
  + 如果第二次的访问时间与第一次访问的**时间间隔在 1 秒内**（默认值），那么该页就**不会**被从 old 区域升级到 young 区域；
  + 如果第二次的访问时间与第一次访问的**时间间隔超过 1 秒**，那么该页就**会**从 old 区域升级到 young 区域；

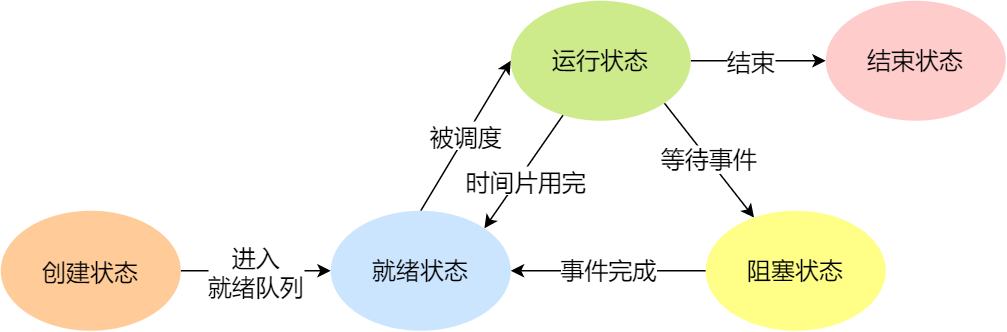
# 五、进程管理

# 5.1 进程、线程基础知识

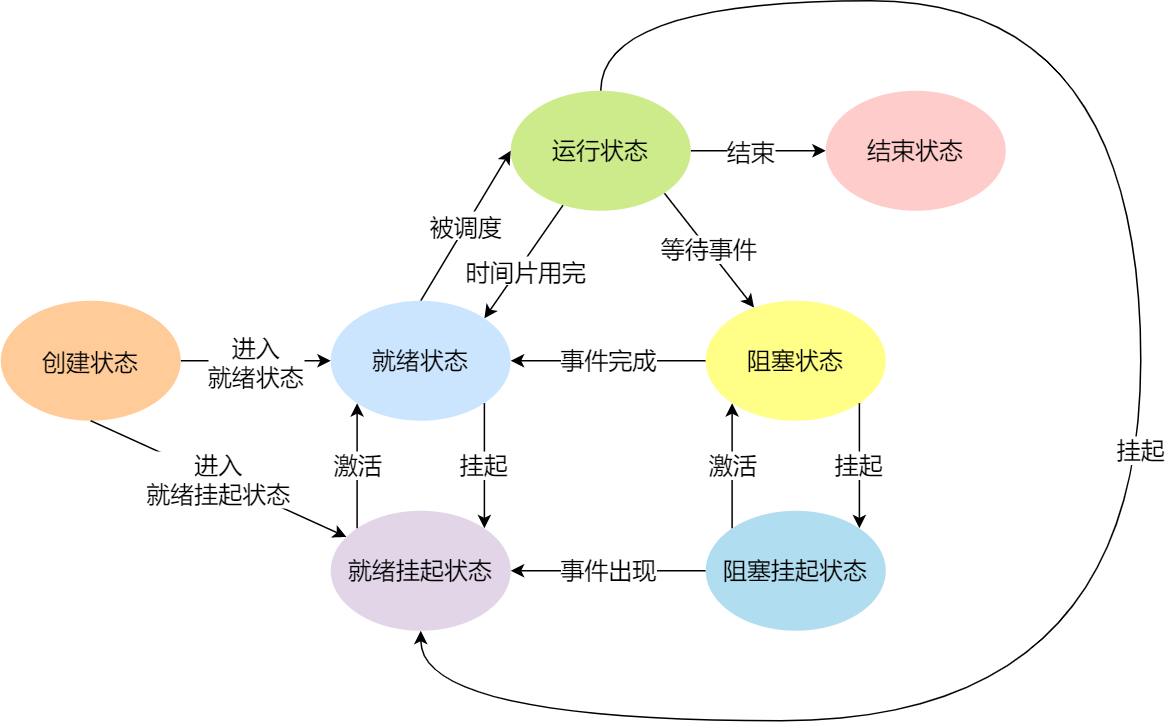
## 进程

**运行中的程序，就被称为「进程」（Process）**

### 进程的状态



如果有大量处于阻塞状态的进程，进程可能会占用着物理内存空间，是一种浪费物理内存的行为。在虚拟内存管理的操作系统中，通常会把阻塞状态的进程的物理内存空间换出到硬盘，等需要再次运行的时候，再从硬盘换入到物理内存。那么就需要一个新的状态来**描述进程没有占用实际的物理内存空间的情况，这个状态就是挂起状态**。



导致进程挂起的原因：

* 操作系统中把阻塞状态的进程的物理内存空间换出到硬盘
* 通过 sleep 让进程间歇性挂起，其工作原理是设置一个定时器，到期后唤醒进程。
* 用户希望挂起一个程序的执行，比如在 Linux 中用 Ctrl+Z 挂起进程；

### 进程的控制结构

操作系统中，是用**进程控制块**（process control block，**PCB**）数据结构来描述进程的

**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（页表等），**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。

每个 PCB 是如何组织的呢？

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**；
* 另外，对于运行队列在单核 CPU 系统中则只有一个**运行指针**了，因为单核 CPU 在某个时间，只能运行一个程序。

### 进程的控制

**01 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。

创建进程的过程如下：

* 申请一个空白的 PCB，并向 PCB 中填写一些控制和管理进程的信息，比如进程的唯一标识等；
* 为该进程分配运行时所必需的资源，比如内存资源；
* 将 PCB 插入到就绪队列，等待被调度运行；

**02 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）。

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。而当父进程被终止时，该父进程的子进程就变为孤儿进程，会被 1 号进程收养，并由 1 号进程对它们完成状态收集工作。

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，则应将该进程的子进程交给 1 号进程接管；
* 将该进程所拥有的全部资源都归还给操作系统；
* 将其从 PCB 所在队列中删除；

**03 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 插入到阻塞队列中去；

**04 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。

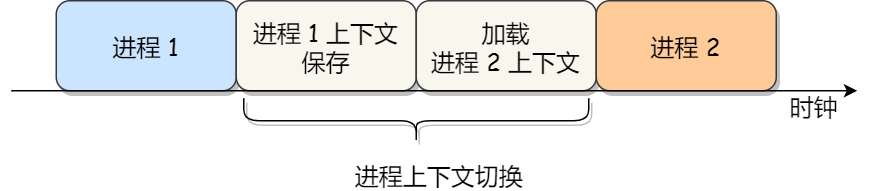
### 进程的上下文切换

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。

**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。**

CPU上下文包括**CPU 寄存器**和**程序计数器**

通常会把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



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，进程就从运行状态变为就绪状态，系统从就绪队列选择另外一个进程运行；
* 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；

## 线程

### 为什么使用线程？

多进程存在的问题：

* 进程之间如何通信、共享数据复杂
* 维护进程的系统开销较大，如创建进程时，分配资源、建立 PCB；终止进程时，回收资源、撤销 PCB；进程切换时，保存当前进程的状态信息；

### 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崩溃时，会导致其所属进程的所有线程崩溃（这里是针对 C/C++，**Java中的线程奔溃不会造成进程崩溃**）

### 线程与进程的比较

|  | **进程** | **线程** | **协程** |
| --- | --- | --- | --- |
| 定义 | 资源分配和拥有的基本单位 | 程序执行的基本单位 | 用户态的轻量级线程，线程内部调度的基本单位 |
| 切换情况 | 进程CPU环境(栈、寄存器、页表和文件句柄等)的保存以及新调度的进程CPU环境的设置 | 保存和设置程序计数器、少量寄存器和栈的内容 | 先将寄存器上下文和栈保存，等切换回来的时候再进行恢复 |
| 切换者 | 操作系统 | 操作系统 | 用户 |
| 切换过程 | 用户态->内核态->用户态 | 用户态->内核态->用户态 | 用户态(没有陷入内核) |
| 调用栈 | 内核栈 | 内核栈 | 用户栈 |
| 拥有资源 | CPU资源、内存资源、文件资源和句柄等 | 程序计数器、寄存器、栈和状态字 | 拥有自己的寄存器上下文和栈 |
| 并发性 | 不同进程之间切换实现并发，各自占有CPU实现并行 | 一个进程内部的多个线程并发执行 | 同一时间只能执行一个协程，而其他协程处于休眠状态，适合对任务进行分时处理 |
| 系统开销 | 切换虚拟地址空间，切换内核栈和硬件上下文，CPU高速缓存失效、页表切换，开销很大 | 切换时只需保存和设置少量寄存器内容，因此开销很小 | 直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快 |
| 通信方面 | 进程间通信需要借助操作系统 | 线程间可以直接读写进程数据段(如全局变量)来进行通信 | 共享内存、消息队列 |

线程与进程的比较如下：

* 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；
* 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；
* 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；
* 线程能减少并发执行的时间和空间开销；

对于，线程相比进程能减少开销，体现在：

* 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，不管是时间效率，还是空间效率线程比进程都要高

### 线程的上下文切换

操作系统的任务调度，**实际上的调度对象是线程**，而进程只是给线程提供了虚拟内存、全局变量等资源。

线程上下文切换的是什么？

* 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**；

### 线程的实现

主要有三种线程的实现方式：

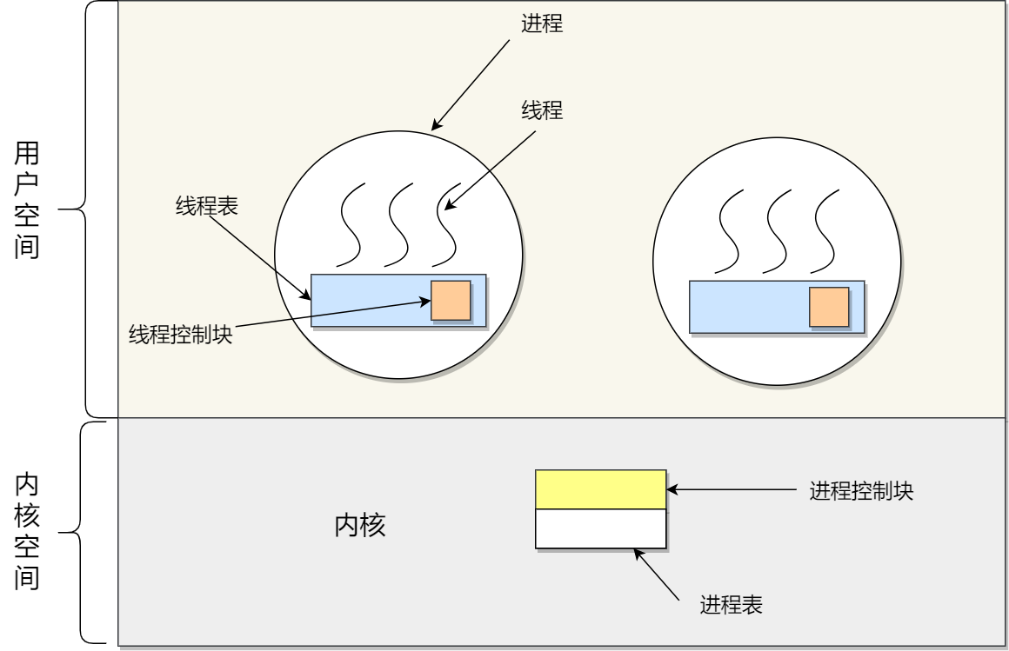
* **用户线程（*User Thread*）**：在**用户空间**实现的线程，**不是由内核管理**的线程，是由**用户态的线程库**来完成线程的管理；
* **内核线程（*Kernel Thread*）**：在**内核**中实现的线程，是由**内核管理**的线程；
* **轻量级进程（*LightWeight Process*）**：在内核中来支持用户线程；

用户线程和内核线程的对应关系：**多对一、一对一、多对多**

用户线程

**用户线程的整个线程管理和调度，操作系统是不直接参与的，而是由用户级线程库函数来完成线程的管理，包括线程的创建、终止、同步和调度等。**

用户级线程的模型是**多对一**的关系，即多个用户线程对应同一个内核线程



**用户线程的优点**：

* 每个进程都需要有它私有的线程控制块（TCB）列表，用来跟踪记录它各个线程状态信息（PC、栈指针、寄存器），TCB 由用户级线程库函数来维护，可用于不支持线程技术的操作系统；
* 用户线程的切换也是由线程库函数来完成的，无需用户态与内核态的切换，所以速度特别快；

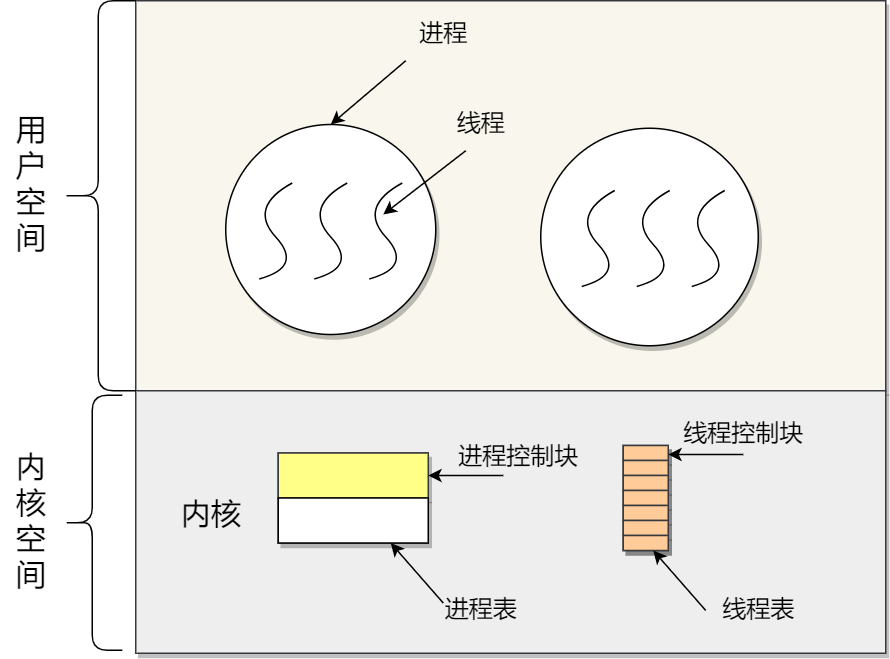
**用户线程的缺点**：

* 由于操作系统不参与线程的调度，如果一个线程发起了系统调用而阻塞，那进程所包含的用户线程都不能执行了。
* 当一个线程开始运行后，除非它主动地交出 CPU 的使用权，否则它所在的进程当中的其他线程无法运行，因为用户态的线程没法打断当前运行中的线程，它没有这个特权，只有操作系统才有，但是用户线程不是由操作系统管理的。
* 由于时间片分配给进程，故与其他进程比，在多线程执行时，每个线程得到的时间片较少，执行会比较慢；

内核线程

**内核线程是由操作系统管理的，线程对应的 TCB 自然是放在操作系统里的，这样线程的创建、终止和管理都是由操作系统负责。**

内核线程的模型，也就类似前面提到的**一对一**的关系，即**一个用户线程对应一个内核线程**



**内核线程的优点**：

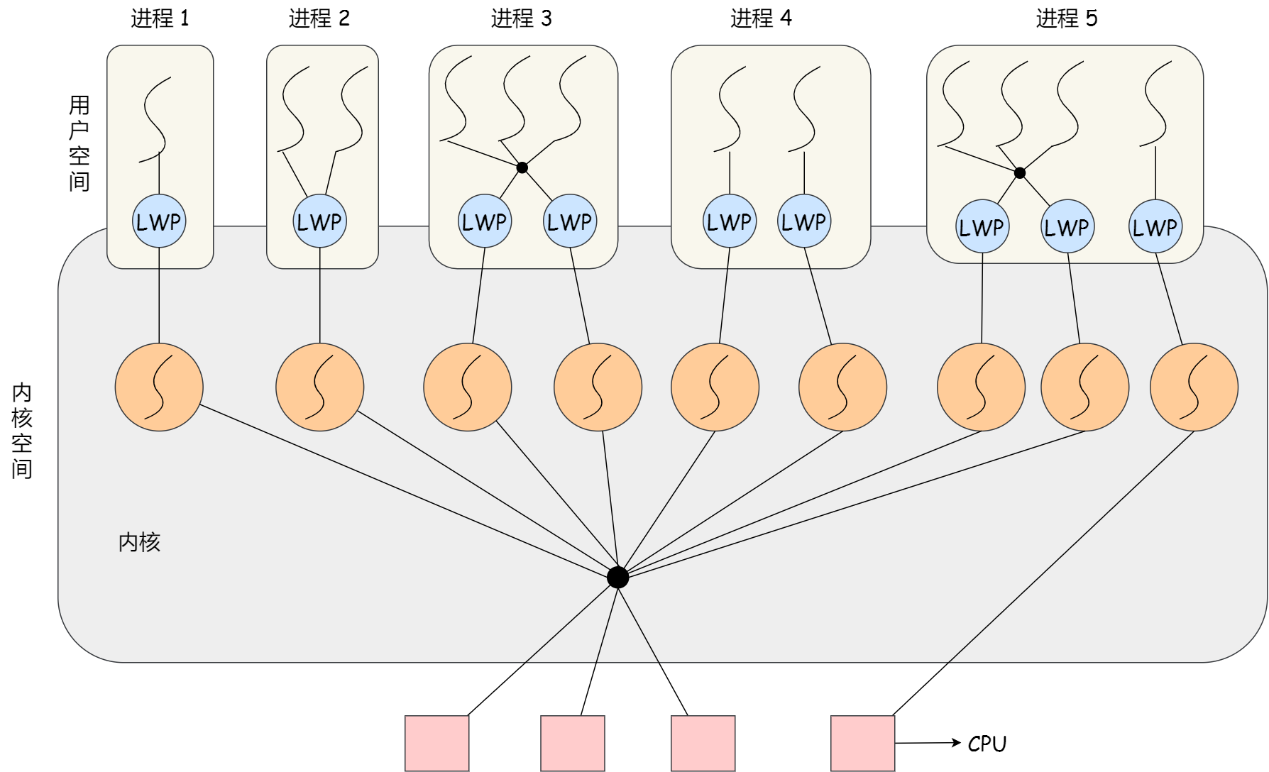
* 在一个进程当中，如果某个内核线程发起系统调用而被阻塞，并不会影响其他内核线程的运行；
* 分配给线程，多线程的进程获得更多的 CPU 运行时间；

**内核线程的缺点**：

* 在支持内核线程的操作系统中，由内核来维护进程和线程的上下文信息，如 PCB 和 TCB；
* 线程的创建、终止和切换都是通过系统调用的方式来进行，因此对于系统来说，系统开销比较大；

轻量级进程

**内核支持的用户线程，一个进程可有一个或多个 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，而且 LWP 是由内核管理并像普通进程一样被调度**。LWP 之上也是可以使用用户线程的，如下图：



## 调度

**选择一个进程运行**这一功能是在操作系统中完成的，通常称为**调度程序**（scheduler）

### 调度时机

当进程**从一个运行状态到另外一状态变化**的时候，会触发一次调度：

* *从*就绪态 -> 运行态：当进程被创建时，会进入到就绪队列，操作系统会从就绪队列选择一个进程运行；
* *从*运行态 -> 阻塞态：当进程发生 I/O 事件而阻塞时，操作系统必须选择另外一个进程运行；
* *从*运行态 -> 结束态：当进程退出结束后，操作系统得从就绪队列选择另外一个进程运行；

根据如何**处理时钟中断** ，把调度算法分为两类：

* **非抢占式调度算法**挑选一个进程，然后让该进程运行直到被阻塞，或者直到该进程退出，才会调用另外一个进程，也就是说不理会时钟中断这个事情。
* **抢占式调度算法**挑选一个进程，然后让该进程只运行某段时间，如果在该时段结束时，该进程仍然在运行时，则会把它挂起，接着调度程序从就绪队列挑选另外一个进程。这种抢占式调度处理，需要在时间间隔的末端发生**时钟中断**，以便把 CPU 控制返回给调度程序进行调度，也就是常说的**时间片机制**。

### 调度原则

原则一：如果运行的程序，发生了 I/O 事件的请求，那 CPU 使用率必然会很低，因为此时进程在阻塞等待硬盘的数据返回。这样的过程，势必会造成 CPU 突然的空闲。所以，**为了提高 CPU 利用率，在这种发送 I/O 事件致使 CPU 空闲的情况下，调度程序需要从就绪队列中选择一个进程来运行。**

原则二：有的程序执行某个任务花费的时间会比较长，如果这个程序一直占用着 CPU，会造成系统吞吐量（CPU 在单位时间内完成的进程数量）的降低。所以，**要提高系统的吞吐率，调度程序要权衡长任务和短任务进程的运行完成数量。**

原则三：从进程开始到结束的过程中，实际上是包含两个时间，分别是进程运行时间和进程等待时间，这两个时间总和就称为周转时间。进程的**周转时间越小越好**，**如果进程的等待时间很长而运行时间很短，那周转时间就很长，这不是我们所期望的，调度程序应该避免这种情况发生。**

原则四：处于就绪队列的进程，也不能等太久，当然希望这个等待的时间越短越好，这样可以使得进程更快的在 CPU 中执行。所以，**就绪队列中进程的等待时间也是调度程序所需要考虑的原则。**

原则五：对于鼠标、键盘这种交互式比较强的应用，我们当然希望它的响应时间越快越好，否则就会影响用户体验了。所以，**对于交互式比较强的应用，响应时间也是调度程序需要考虑的原则。**

总结：

* **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+阻塞时间+等待时间的总和，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。

### 调度算法

**① 单核 CPU 系统**

**01 先来先服务调度算法（First Come First Serve, FCFS）**

**每次从就绪队列选择最先进入队列的进程，然后一直运行直到进程退出或被阻塞，才会继续从队列中选择第一个进程接着运行。**

缺点：长作业先运行了，那么后面的短作业等待的时间就会很长

特点：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。

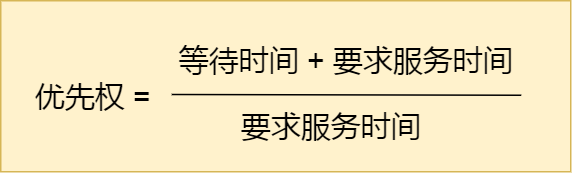
**02 最短作业优先调度算法（Shortest Job First, SJF）**

**优先选择运行时间最短的进程来运行**

缺点：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。

**03 高响应比优先调度算法（Highest Response Ratio Next, HRRN）**

**每次进行进程调度时，先计算「响应比优先级」，然后把「响应比优先级」最高的进程投入运行**



特点：

* 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；

缺点：服务时间是不可预估的。所以，高响应比优先调度算法是「理想型」的调度算法，现实中是实现不了的。

**04 时间片轮转调度算法（Round Robin, RR）**

最古老、最简单、最公平且使用最广的调度算法

**每个进程被分配一个时间段，称为时间片（*Quantum*），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 分配给另外一个进程；
* 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；

**时间片的长度**的选择：

* 如果时间片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。

20ms~50ms 通常比较合理

**05 最高优先级调度算法（Highest Priority First，HPF）**

**多用户计算机系统**希望调度程序能**从就绪队列中选择最高优先级的进程进行运行**。

进程的优先级可以分为：

* **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：

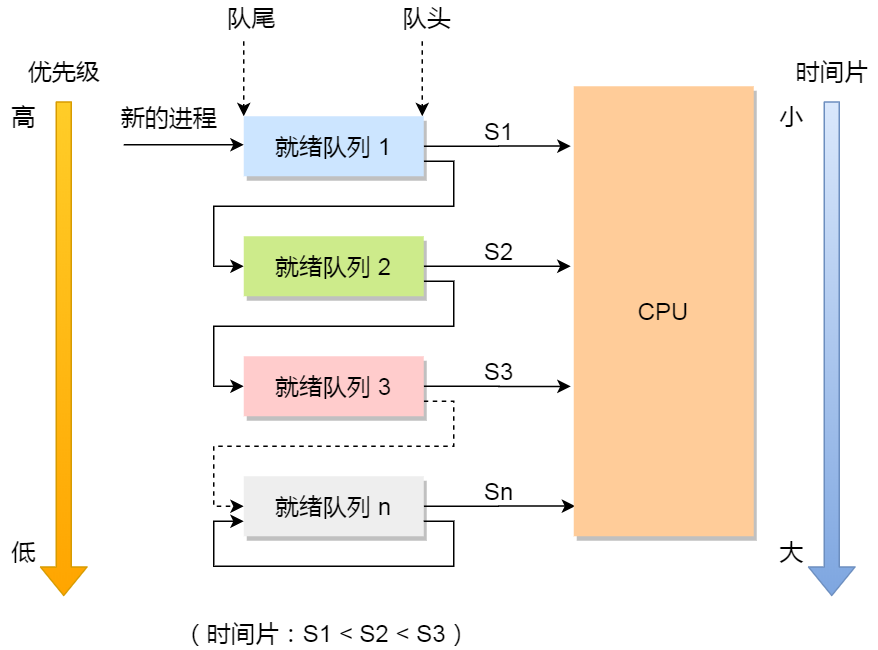
* **非抢占式**：当就绪队列中出现优先级高的进程，运行完当前进程，再选择优先级高的进程。
* **抢占式**：当就绪队列中出现优先级高的进程，当前进程挂起，调度优先级高的进程运行。

缺点：导致**低优先级的进程永远不会运行**

**06 多级反馈队列调度算法（Multilevel Feedback Queue）**

是「时间片轮转算法」和「最高优先级算法」的综合和发展。

* 「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



工作流程：

* 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，但是运行时间也变更长了，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。**

# 5.2 进程间有哪些通信方式？

## 管道

* 管道其实是一个在**内核内存中维护的缓冲器**，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同
* 管道拥有文件的特质：读操作、写操作
  + **匿名管道**没有文件实体
  + **有名管道**有文件实体，但不存储数据。可以按照操作文件的方式对管道进行操作
* **一个管道是一个字节流**，使用管道时不存在消息或者消息边界的概念，从管道读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少
* 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺序是完全一样的
* 在管道中的**数据的传递方向是单向的，一端用于写入，一端用于读取，管道是半双工的**
* 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据，**在管道中无法使用 lseek() 来随机访问数据**
* 匿名管道只能在**具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用**

缺点：**管道这种通信方式效率低，不适合进程间频繁地交换数据**



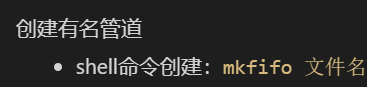
匿名管道（**pipe**）。使用 fork 创建子进程，**创建的子进程会复制父进程的文件描述符**，这样就做到了两个进程各有两个「 fd[0] 与 fd[1]」，两个进程就可以通过各自的 fd 写入和读取同一个管道文件实现跨进程通信。管道只能一端写入，另一端读出，所以上面这种模式容易造成混乱，因为父进程和子进程都可以同时写入，也都可以读出。那么，为了避免这种情况，通常的做法是：

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

如果需要双向通信，则应该创建两个管道。

Linux**管道符 | 的实现**：

执行 A | B命令的时候，**A 进程和 B 进程都是 shell 创建出来的子进程**，A 和 B 之间不存在父子关系，它俩的父进程都是 shell。





有名管道（**FIFO**），提前创建了一个类型为管道的设备文件，在进程里只要使用这个设备文件，就可以相互通信。

## 消息队列

**消息队列是保存在内核中的消息链表**，在发送数据时，会分成一个一个独立的数据单元，也就是消息体（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，所以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。

消息队列**生命周期随内核**，如果没有释放消息队列或者没有关闭操作系统，消息队列会一直存在

缺点：**一是通信不及时，二是附件也有大小限制**

**消息队列不适合比较大数据的传输**，因为在内核中每个消息体都有一个最大长度的限制，同时所有队列所包含的全部消息体的总长度也是有上限。在 Linux 内核中，会有两个宏定义 MSGMAX 和 MSGMNB，它们以字节为单位，分别定义了一条消息的最大长度和一个队列的最大长度。

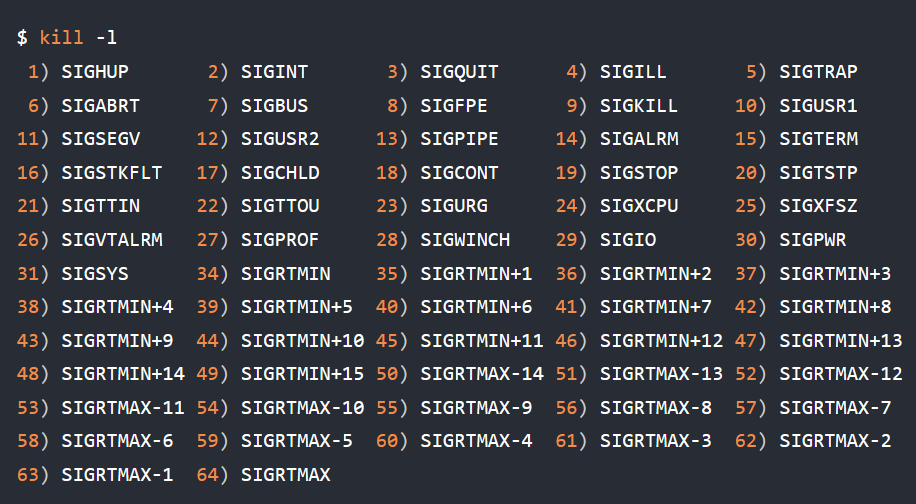
**消息队列通信过程中，存在用户态与内核态之间的数据拷贝开销**，因为进程写入数据到内核中的消息队列时，会发生从用户态拷贝数据到内核态的过程，同理另一进程读取内核中的消息数据时，会发生从内核态拷贝数据到用户态的过程。

## 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**

* **共享内存允许两个或者多个进程共享物理内存的同一块区域（通常被称为段）**。由于一个**共享内存段会成为一个进程用户空间的一部分**，因此这种 IPC（Inter-Process Communication） 机制**无需内核介入**。所以需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用
* 与管道等要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 IPC 技术的速度更快

## 信号



Linux中对于前台进程：

* Ctrl+C 产生 SIGINT 信号，表示终止该进程；
* Ctrl+Z 产生 SIGTSTP 信号，表示停止该进程，但还未结束；

对于后台进程：

* kill -9 1050 ，表示给 PID 为 1050 的进程发送 SIGKILL 信号，立即结束该进程；

信号事件的来源主要有**硬件来源**（如键盘 Cltr+C ）和**软件来源**（如 kill 命令）。

信号是进程间通信机制中**唯一的异步通信机制**，因为可以在任何时候发送信号给某一进程，一旦有信号产生，我们就有下面这几种，用户进程对信号的处理方式。

**1.执行默认操作**。Linux 对每种信号都规定了默认操作，例如，上面列表中的 SIGTERM 信号的默认动作是终止进程。

**2.捕捉信号**。我们可以为信号**自定义一个信号处理函数**。当信号发生时，我们就执行相应的信号处理函数。

**3.忽略信号**。当我们不希望处理某些信号的时候，就可以忽略该信号，不做任何处理。有两个信号是应用进程无法捕捉和忽略的，即 SIGKILL 和 SEGSTOP，它们用于在任何时候中断或结束某一进程

信号的 5 种默认处理动作：

* + Term：终止进程
  + Ign：当前进程忽略掉这个信号
  + Core：终止进程，并生成一个Core文件
  + Stop：暂停当前进程
  + Cont：继续执行当前被暂停的进程

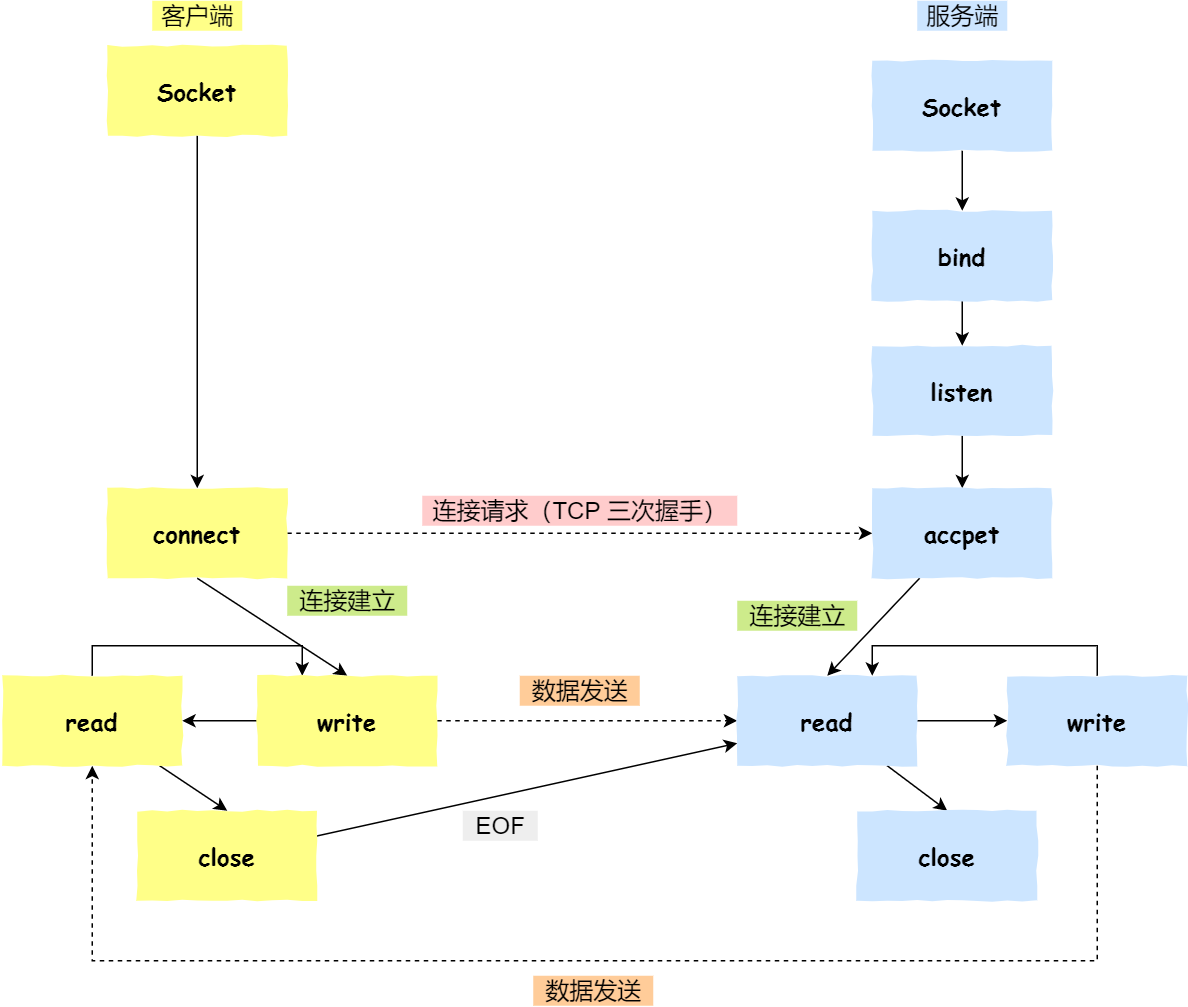
## Socket

**跨网络与不同主机上的进程之间通信，就需要 Socket 通信了（Socket也可以同一主机）。**

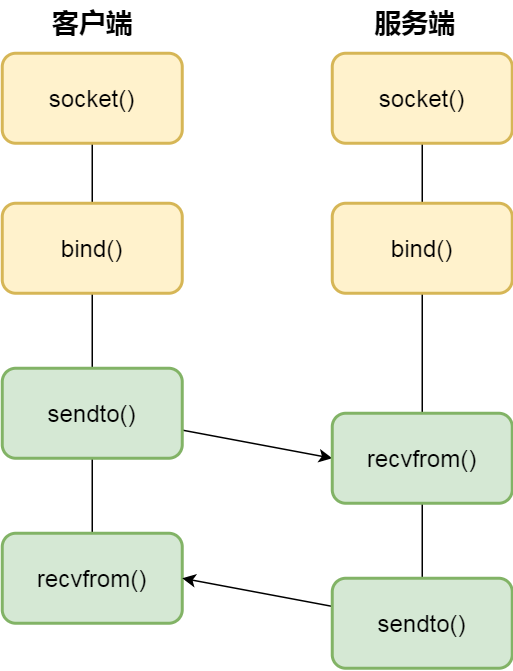


* domain 参数用来指定协议族，比如 AF\_INET 用于 IPV4、AF\_INET6 用于 IPV6、AF\_LOCAL/AF\_UNIX 用于本机；
* type 参数用来指定通信特性，比如 SOCK\_STREAM 表示的是字节流，对应 TCP；SOCK\_DGRAM 表示的是数据报，对应 UDP；SOCK\_RAW 表示的是原始套接字；
* protocal 参数原本是用来指定通信协议的，但现在基本废弃。因为协议已经通过前面两个参数指定完成，protocol 目前一般写成 0 即可；

针对 TCP 协议通信的 socket 编程模型



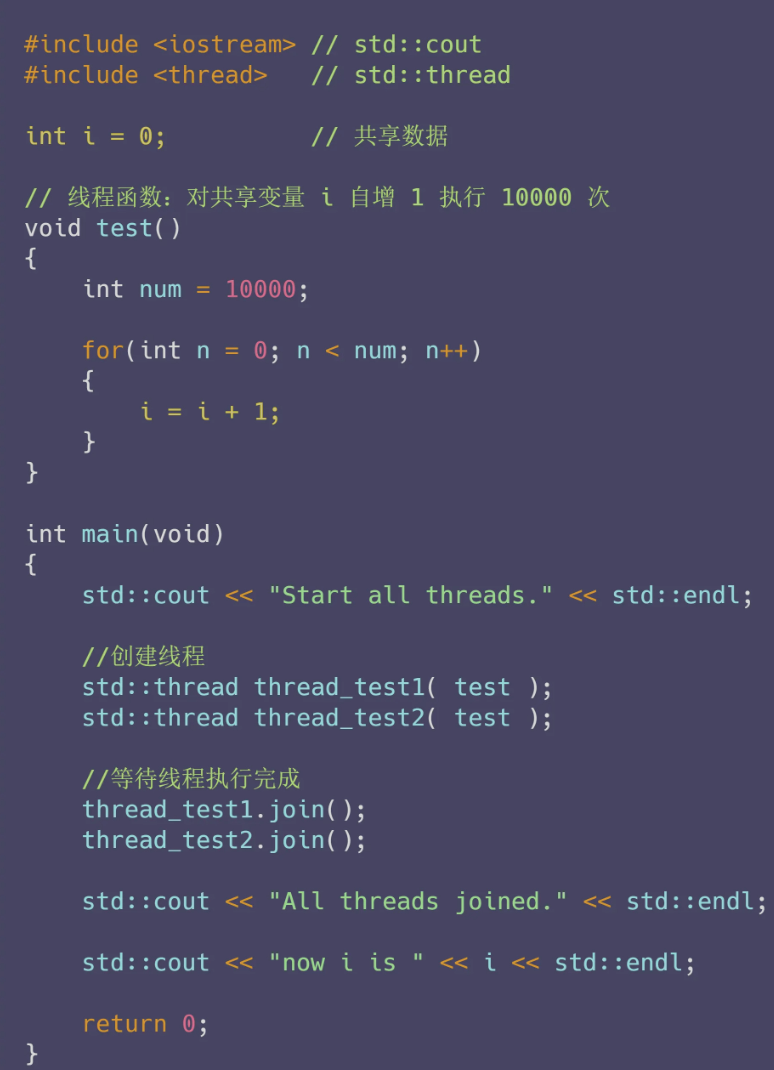
针对 UDP 协议通信的 socket 编程模型

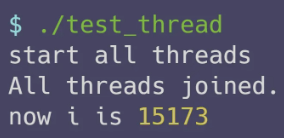


# 5.3 多线程冲突了怎么办？

## 竞争

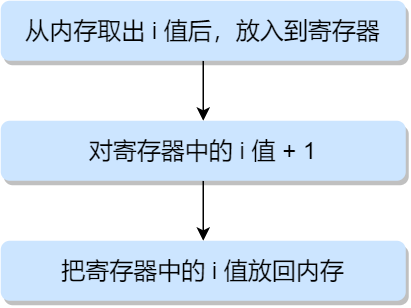
线程之间是可以共享进程的资源，比如代码段、堆空间、数据段、打开的文件等资源，但每个线程都有自己独立的寄存器和栈空间。



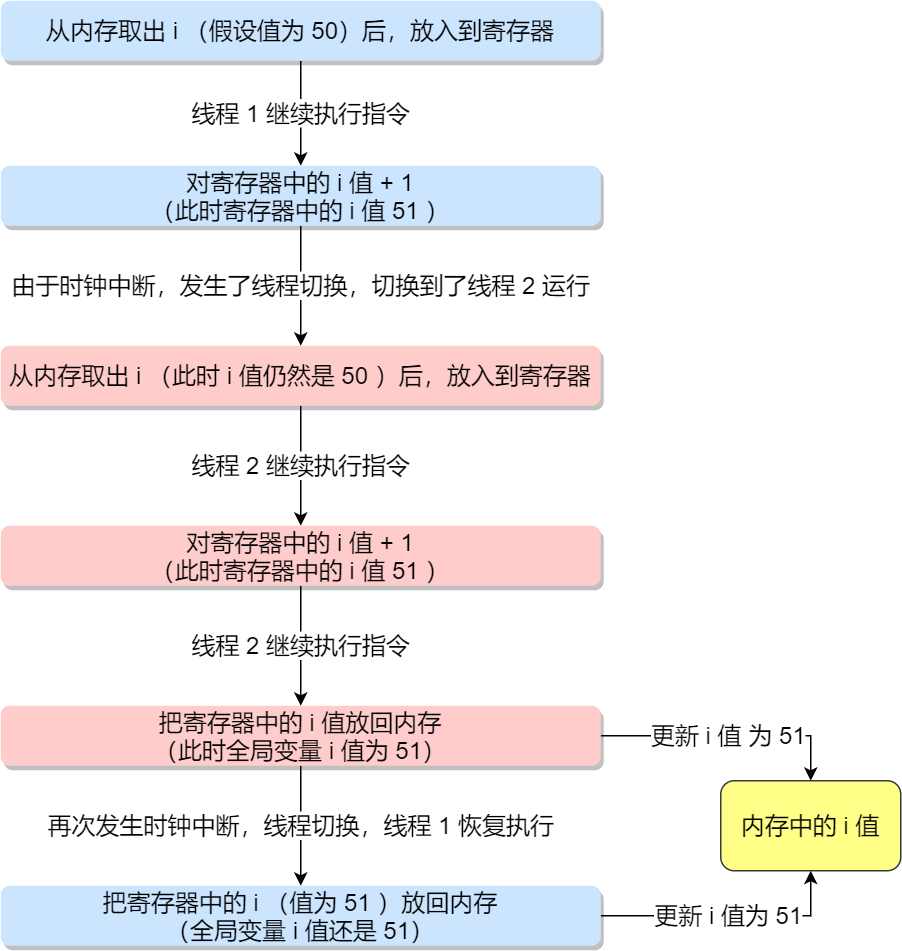


为什么会发生这种情况？

i = i + 1的过程：



发生多线程冲突的过程：



### 互斥的概念

多线程相互竞争操作共享变量时，当在执行过程中发生了上下文切换，会得到错误的结果。

多线程执行操作共享变量的这段代码可能会导致竞争状态，因此我们将此段代码称为**临界区（critical section），它是访问共享资源的代码片段，一定不能给多线程同时执行。**

### 同步的概念

**同步，就是并发进程/线程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通信息称为进程/线程同步**。

**同步与互斥的区别**：

* **同步**：「操作 A 应在操作 B 之前执行」，「操作 C 必须在操作 A 和操作 B 都完成之后才能执行」等；
* **互斥**：「操作 A 和操作 B 不能在同一时刻执行」；

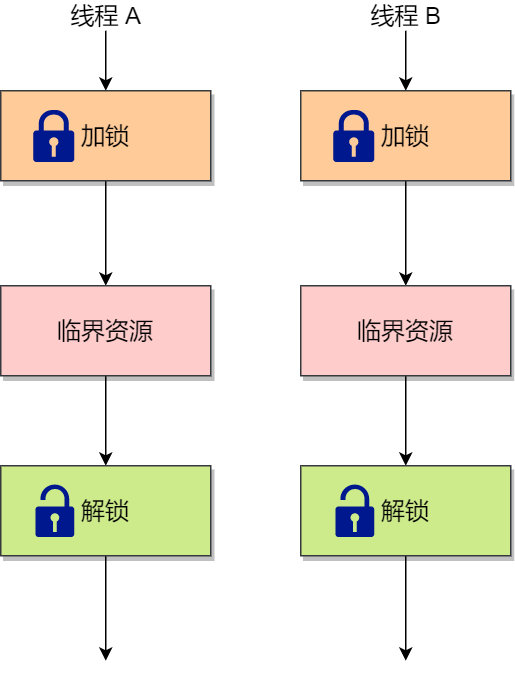
## 互斥与同步的实现和使用

主要的方法有两种：

* **锁**：加锁、解锁操作；
* **信号量**：P、V 操作；

### 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」

当获取不到锁时，线程就会一直阻塞等待，不做任何事情，所以就被称为「忙等待锁」，也被称为**自旋锁（spin lock）**。

「无忙等待锁」

无忙等待锁就是获取不到锁的时候，不用自旋。既然不想自旋，那当没获取到锁的时候，就**把当前线程放入到锁的等待队列，然后执行调度程序，把 CPU 让给其他线程执行**。

### 信号量

**信号量表示资源的数量**，对应的变量是一个整型（sem）变量。

**有两个原子操作的系统调用函数来控制信号量**：

* **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；

**个人理解：**

**① sem≥0时表示剩余资源量，<0时表示当前阻塞等待的进程/线程数量**

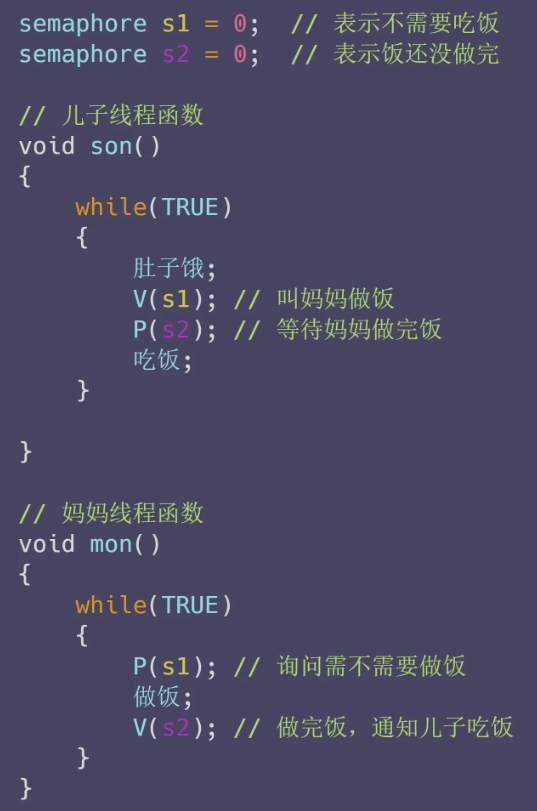
**② 为什么sem>0不需要唤醒等待进程/线程？因为>0时不可能有在等待的进程/线程。**

PV 操作如何使用

① 使用**信号量实现临界区的互斥访问**

为每类共享资源设置一个信号量 s，其初值为 1，表示该临界资源未被占用。只要把进入临界区的操作置于 P(s) 和 V(s) 之间，即可实现进程/线程互斥。

② 使用**信号量实现事件同步**



### 生产者-消费者问题



生产者-消费者问题描述：

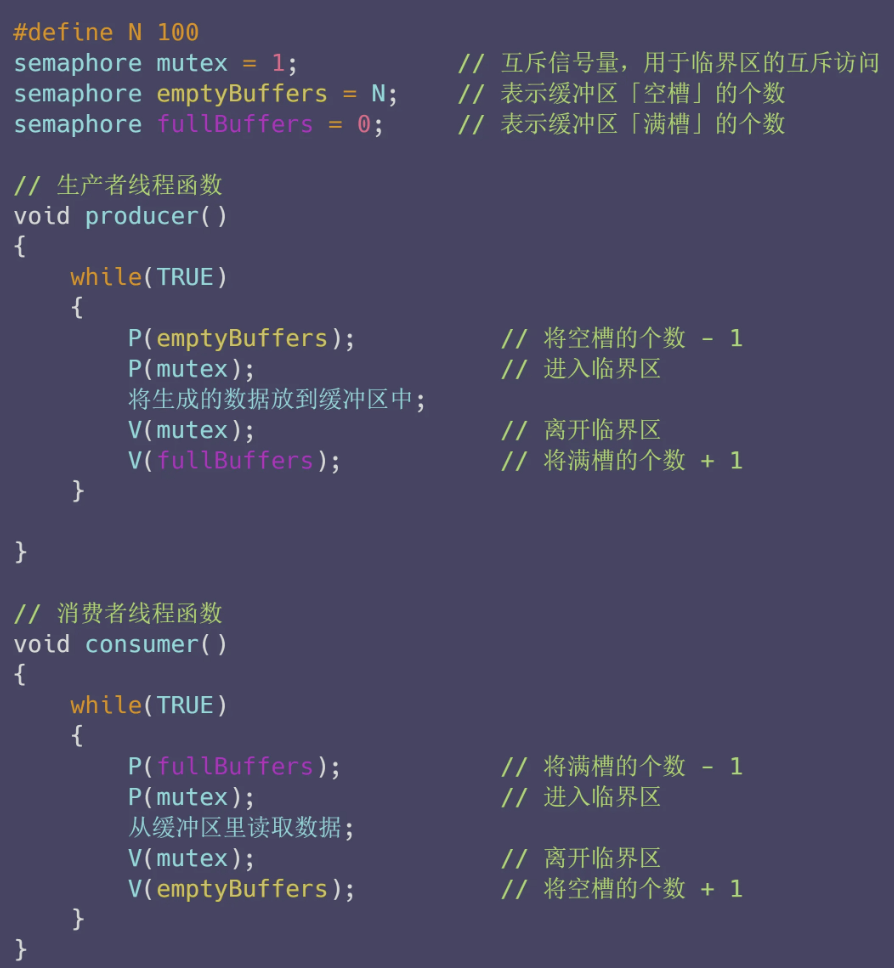
* **生产者**在生成数据后，放在一个缓冲区中；
* **消费者**从缓冲区取出数据处理；
* 任何时刻，**只能有一个**生产者或消费者可以访问缓冲区；

对问题分析可以得出：

* 任何时刻只能有一个线程操作缓冲区，说明操作缓冲区是临界代码，**需要互斥**；
* 缓冲区空时，消费者必须等待生产者生成数据；缓冲区满时，生产者必须等待消费者取出数据。说明生产者和消费者**需要同步**。

需要三个信号量，分别是：

* **互斥信号量** **mutex**：用于互斥访问缓冲区，初始化值为 1；
* **资源信号量** **fullBuffers**：用于消费者询问缓冲区是否有数据，有数据则读取数据，初始化值为 0（表明缓冲区一开始为空）；
* **资源信号量 emptyBuffers**：用于生产者询问缓冲区是否有空位，有空位则生成数据，初始化值为 n （缓冲区大小）；



## 经典同步问题

### 哲学家就餐问题

问题描述：

* 5 个老大哥哲学家，闲着没事做，围绕着一张圆桌吃面；
* 巧就巧在，这个桌子只有 5 支叉子，每两个哲学家之间放一支叉子；
* 哲学家围在一起先思考，思考中途饿了就会想进餐；
* **这些哲学家要两支叉子才愿意吃面，也就是需要拿到左右两边的叉子才进餐**；
* **吃完后，会把两支叉子放回原处，继续思考**；

**假设五位哲学家同时拿起左边的叉子，这样就没有人能够拿到他们右边的叉子，发生了死锁的现象**。

解决方法：

① 设置互斥量，每次只能有一个人进餐。**缺点：同时只能一个人。**

② **偶数编号的哲学家「先拿左边的叉子后拿右边的叉子」，奇数编号的哲学家「先拿右边的叉子后拿左边的叉子」**

③ **用一个数组 state 来记录每一位哲学家的三个状态，分别是在进餐状态、思考状态、饥饿状态（正在试图拿叉子）。一个哲学家只有在两个邻居都没有进餐时，才可以进入进餐状态。**

### 读者-写者问题

读者只会读取数据，不会修改数据，而写者即可以读也可以修改数据。

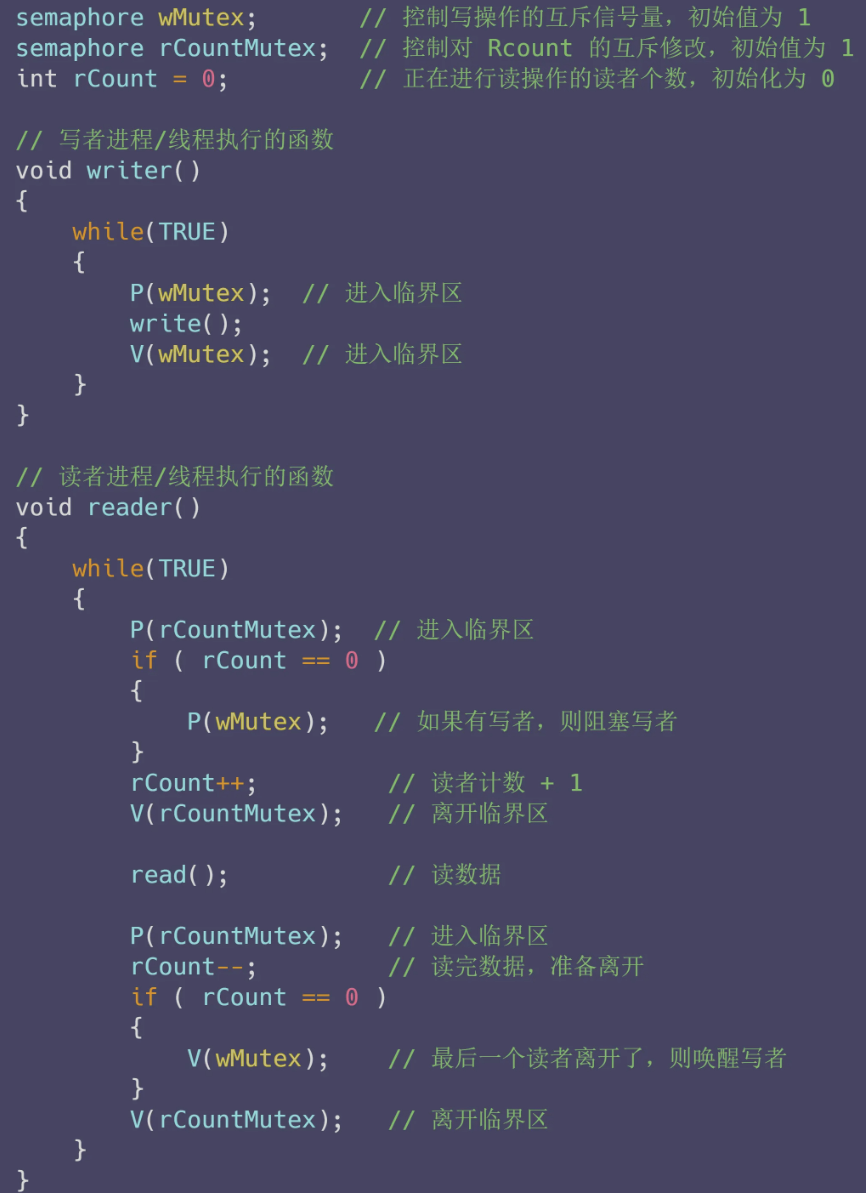
读者-写者的问题描述：

* 「读-读」允许：同一时刻，允许多个读者同时读
* 「读-写」互斥：没有写者时读者才能读，没有读者时写者才能写
* 「写-写」互斥：没有其他写者时，写者才能写

方案一

使用信号量的方式来尝试解决：

* 信号量 wMutex：控制写操作的互斥信号量，初始值为 1 ；
* 读者计数 rCount：正在进行读操作的读者个数，初始化为 0；
* 信号量 rCountMutex：控制对 rCount 读者计数器的互斥修改，初始值为 1；



实际上是**读者优先策略**

缺点：如果读者持续不断进入，则写者会处于饥饿状态。

方案二

**写者优先策略**：

* 只要有写者准备要写入，写者应尽快执行写操作，后来的读者就必须阻塞；
* 如果有写者持续不断写入，则读者就处于饥饿；

方案三

**公平策略**：

* 优先级相同；
* 写者、读者互斥访问；
* 只能一个写者访问临界区；
* 可以有多个读者同时访问临界资源；

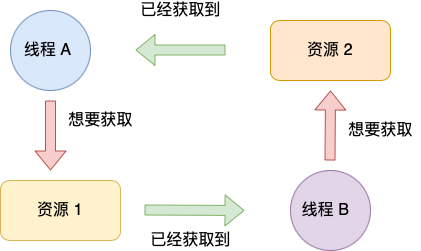
# 5.4 怎么避免死锁？

## 死锁的概念

两个线程为了**保护两个不同的共享资源而使用了两个互斥锁**，那么这两个互斥锁应用不当的时候，可能会造成**两个线程都在等待对方释放锁**，在没有外力的作用下，这些线程会**一直相互等待**，就没办法继续运行，这种情况就是发生了**死锁**。

死锁只有**同时**满足以下四个条件才会发生：

* 互斥条件：**多个线程不能同时使用同一个资源**
* 持有并等待条件：**线程在等待资源的同时并不会释放自己已经持有的资源**
* 不可剥夺条件：线程已经持有的资源**在自己使用完之前不能被其他线程获取**
* 环路等待条件：**两个线程获取资源的顺序构成了环形链**



## 利用工具排查死锁问题

Java：jstack

C: linux下pstack + gdb

## 避免死锁问题的发生

核心思想：**破环死锁四个条件其中一个**

最常见的并且可行的就是**使用资源有序分配法，来破环环路等待条件**。

即**保证各个线程获取资源的顺序要一样**

# 5.5 什么是悲观锁、乐观锁？

## 互斥锁与自旋锁

* **互斥锁**加锁失败后，线程会**释放 CPU** ，给其他线程；
* **自旋锁**加锁失败后，线程会**忙等待**，直到它拿到锁；

**互斥锁加锁失败而阻塞的现象，是由操作系统内核实现的**。当加锁失败时，内核会将线程置为**「睡眠」**状态，等到锁被释放后，内核会在合适的时机唤醒线程，当这个线程成功获取到锁后，于是就可以继续执行。

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。**上下切换的耗时大概在几十纳秒到几微秒之间，**如果能确定被锁住的代码执行时间很短，就不应该用互斥锁，而应该选用自旋锁，否则使用互斥锁。**

**自旋锁**是通过 CPU 提供的 CAS 函数（Compare And Swap），在**「用户态」**完成加锁和解锁操作，不会主动产生线程上下文切换，所以相比互斥锁来说，会快一些，开销也小一些。

注意：**在单核 CPU 上，需要抢占式的调度器（即不断通过时钟中断一个线程，运行其他线程）。否则，自旋锁在单 CPU 上无法使用，因为一个自旋的线程永远不会放弃 CPU。**

## 读写锁

**读写锁适用于能明确区分读操作和写操作的场景**

## 乐观锁与悲观锁

**悲观锁**认为**多线程同时修改共享资源的概率比较高，于是很容易出现冲突，所以访问共享资源前，先要上锁**。

**乐观锁**假定**冲突的概率很低**，它的工作方式是：**先修改完共享资源，再验证这段时间内有没有发生冲突，如果没有其他线程在修改资源，那么操作完成，如果发现有其他线程已经修改过这个资源，就放弃本次操作**。

**乐观锁全程并没有加锁，所以它也叫无锁编程**。

# 5.6 一个进程最多可以创建多少个线程？

Linux系统中与2个因素有关：

* **进程的虚拟内存空间上限（主要针对32位系统，用户态3G虚拟内存）**，因为创建一个线程，操作系统需要为其分配一个栈空间，如果线程数量越多，所需的栈空间就要越大，那么虚拟内存就会占用的越多。
* **系统参数限制（主要针对64位系统，用户态128T虚拟内存）**，虽然 Linux 并没有内核参数来控制单个进程创建的最大线程个数，但是有系统级别的参数来控制整个系统的最大线程个数。