# 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；

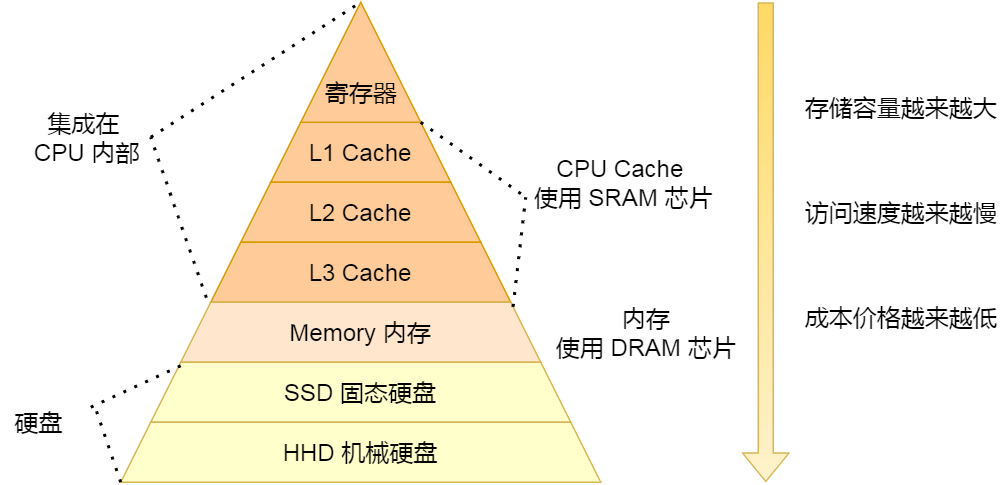
CPU 要读写内存数据的时候，一般需要通过下面这三个总线：

* 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。

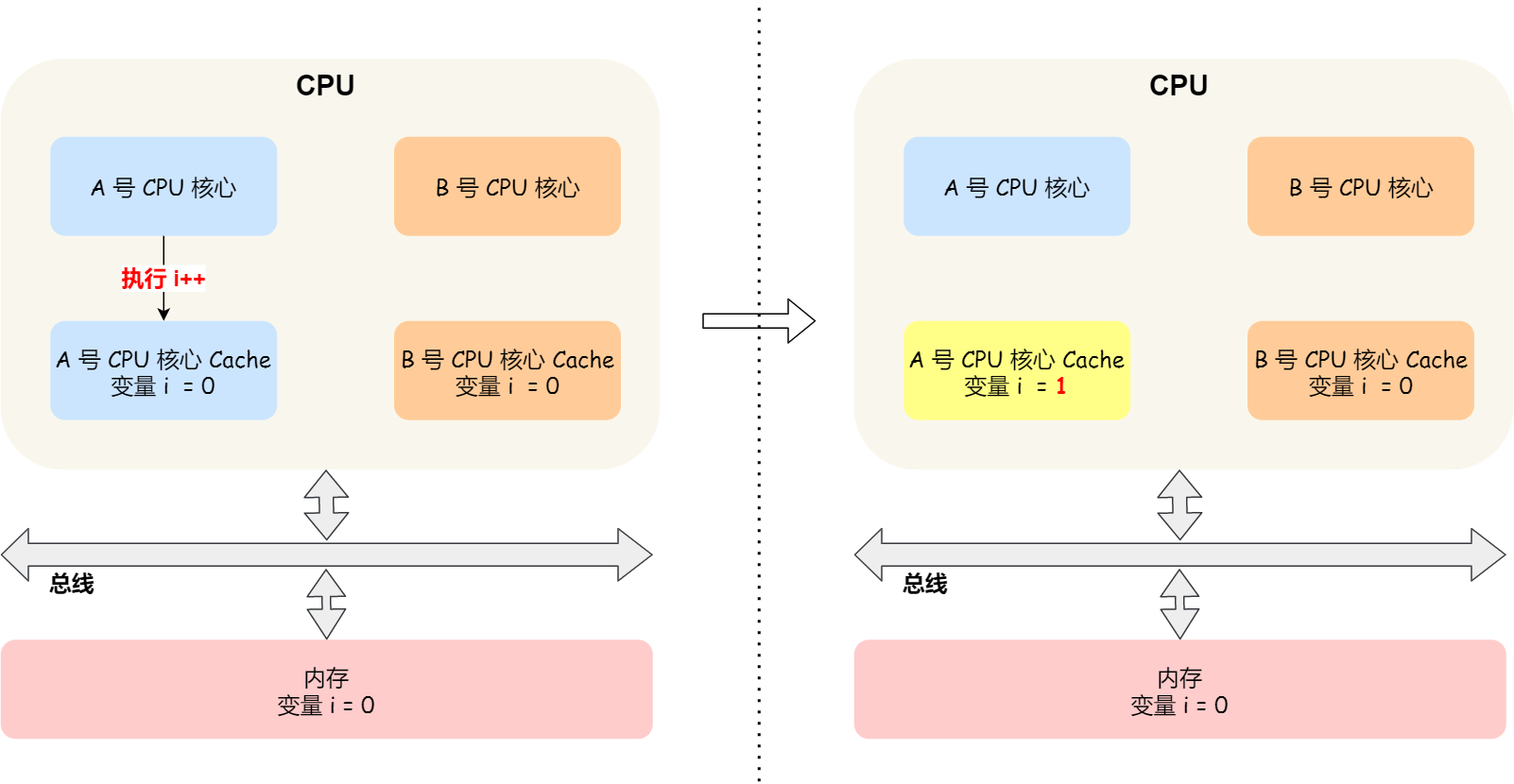
### 写回

**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



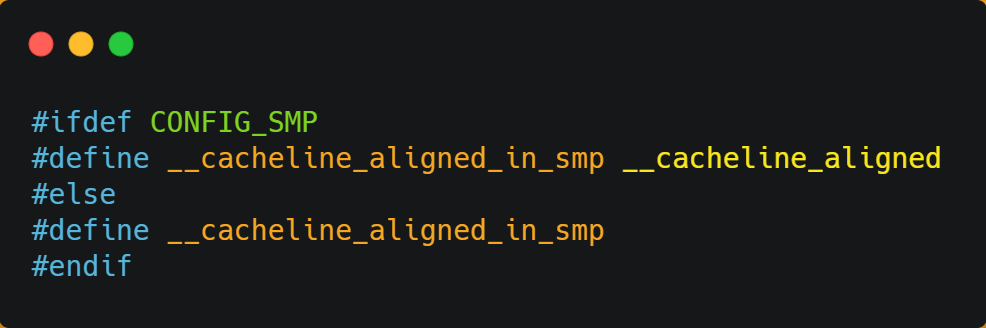
要保证做到下面这 2 点：

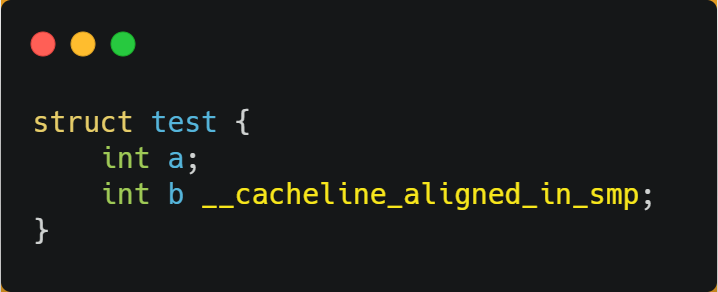
* 第一点，某个 CPU 核心里的 Cache 数据更新时，必须要传播到其他核心的 Cache，这个称为写传播（*Write Propagation*）；
* 第二点，某个 CPU 核心里对数据的操作顺序，必须在其他核心看起来**顺序是一样**的，这个称为事务的串行化（*Transaction Serialization*）。

### 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。





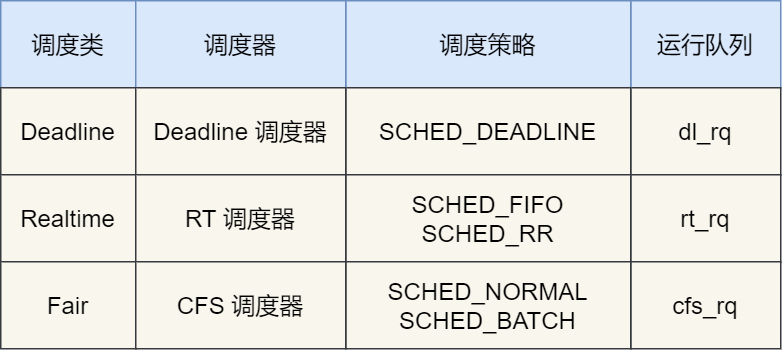
针对在同一个 Cache Line 中的共享的数据，如果在多核之间竞争比较严重，为了防止伪共享现象的发生，可以采用上面的宏定义使得变量在 Cache Line 里是对齐的。

## 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

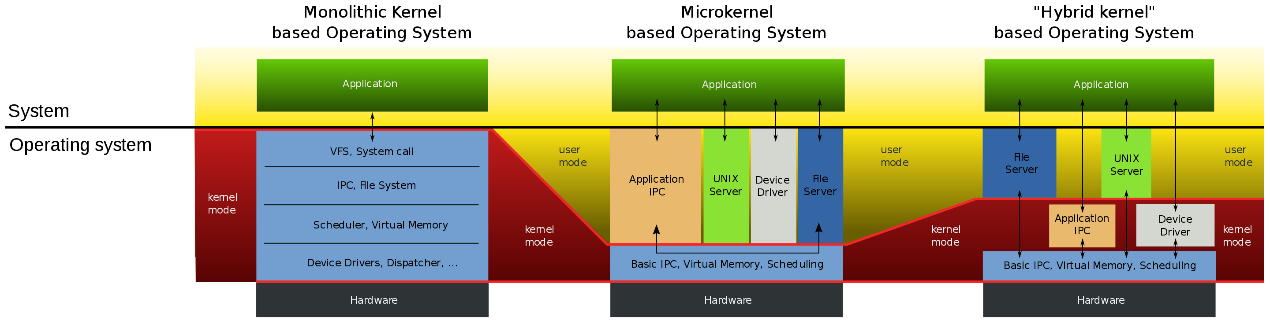
## Linux 的设计

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**

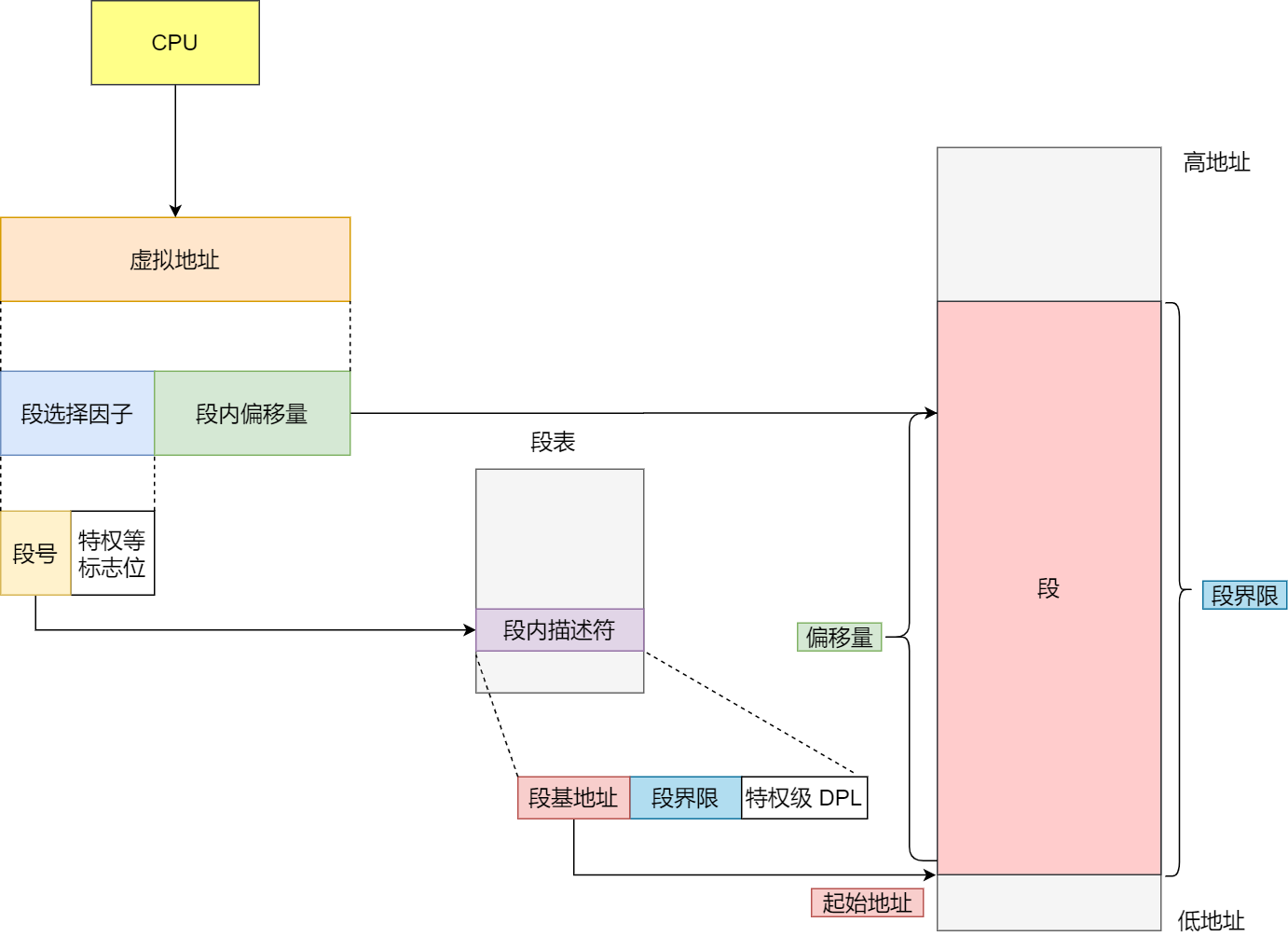
## 虚拟内存有什么作用：

* 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**。



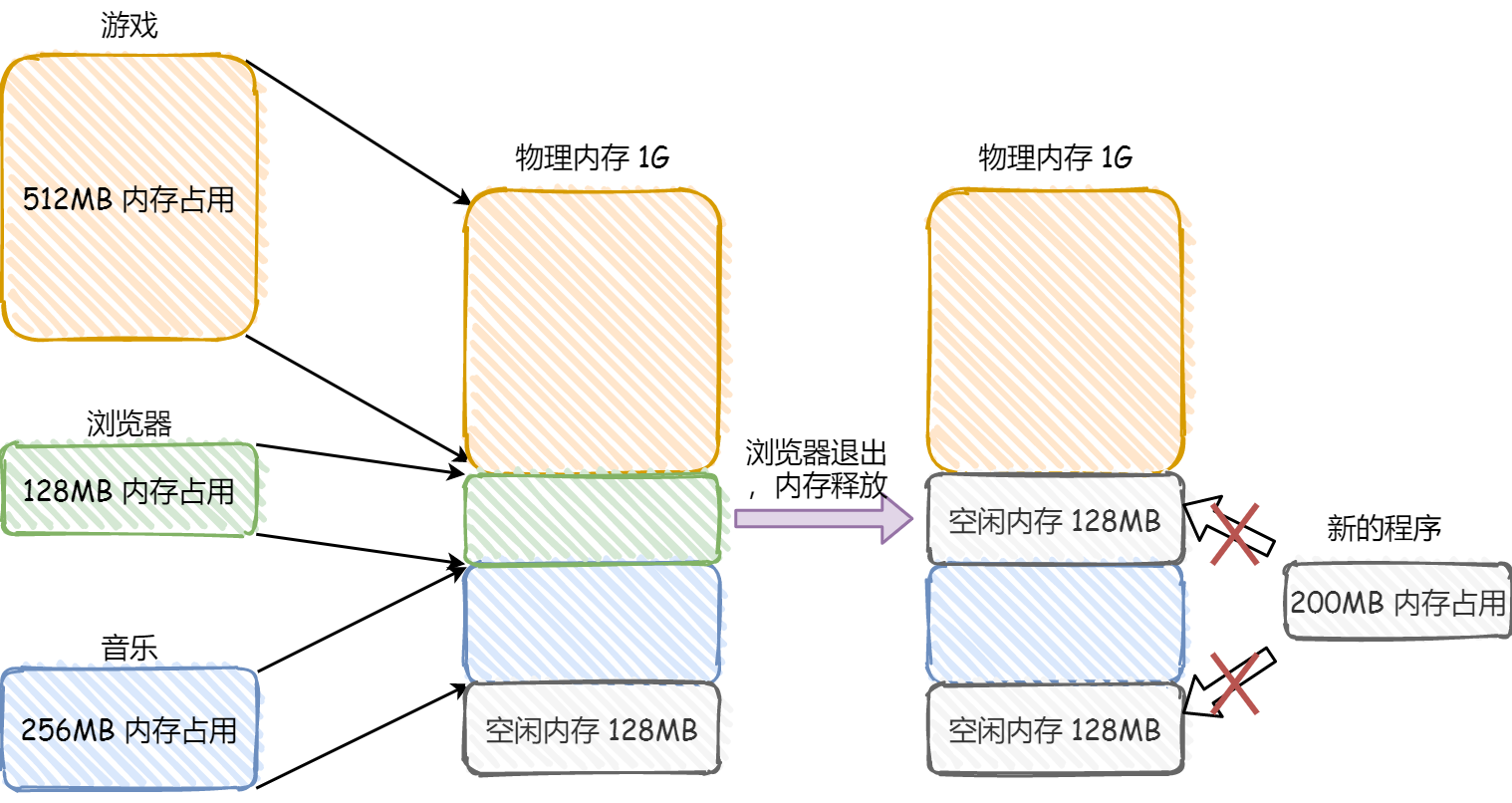
缺点：

* 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。

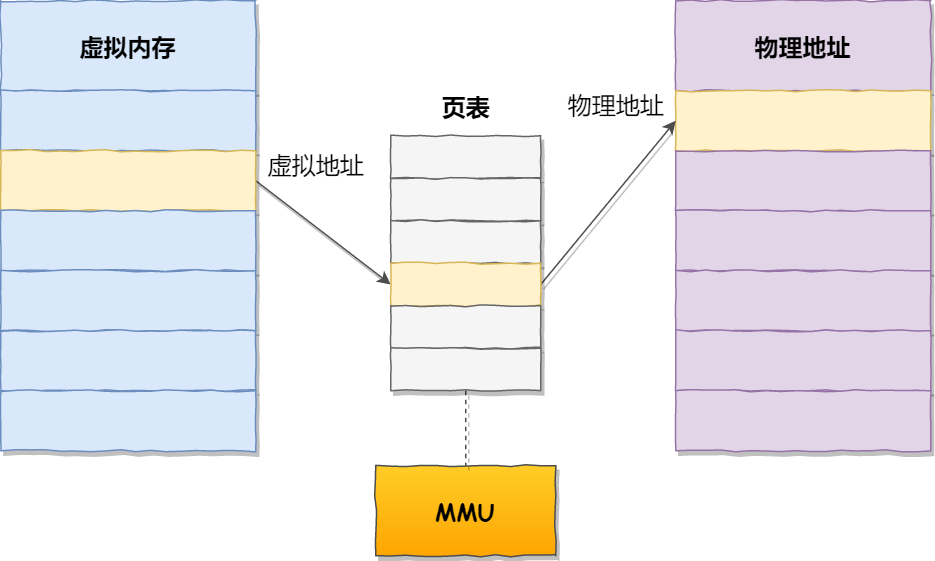
分段为什么会导致内存交换效率低的问题？

**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。

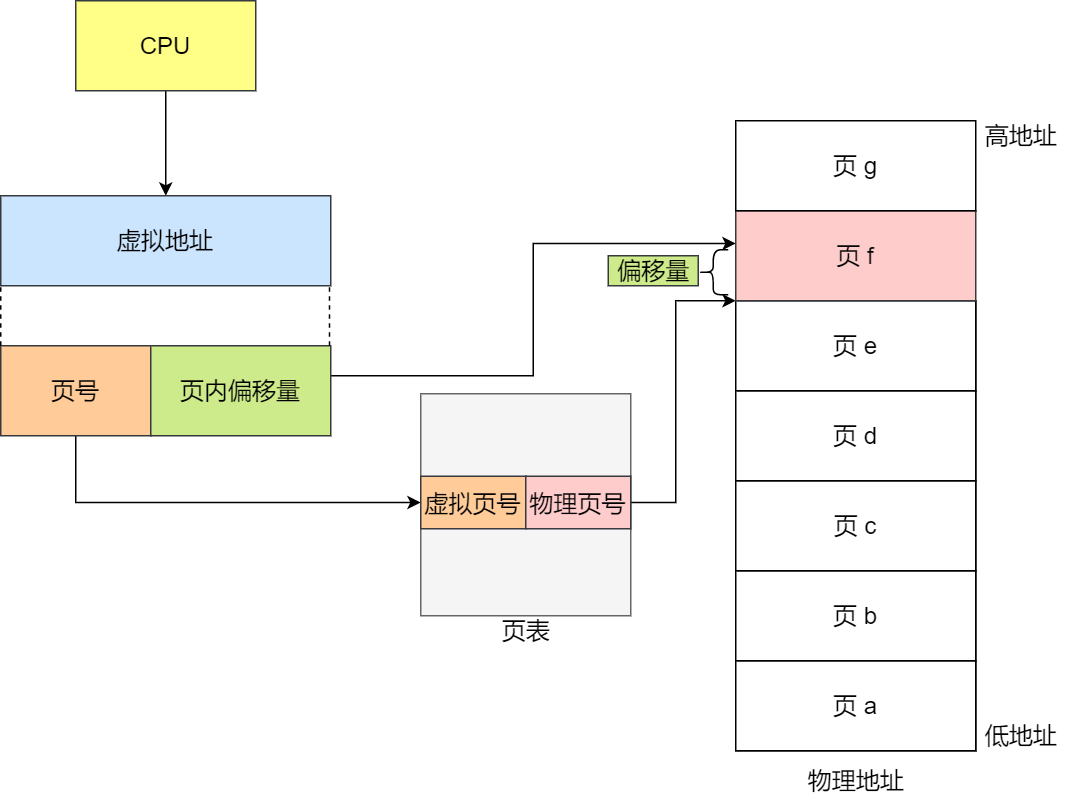
**内存分页时页与页之间是紧密排列的，所以不会有外部碎片。**但是，因为内存分页机制分配内存的**最小单位是一页**，即使程序不足一页大小，我们最少只能分配一个页，所以页内会出现内存浪费，所以针对**内存分页机制会有内部内存碎片**的现象。

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？

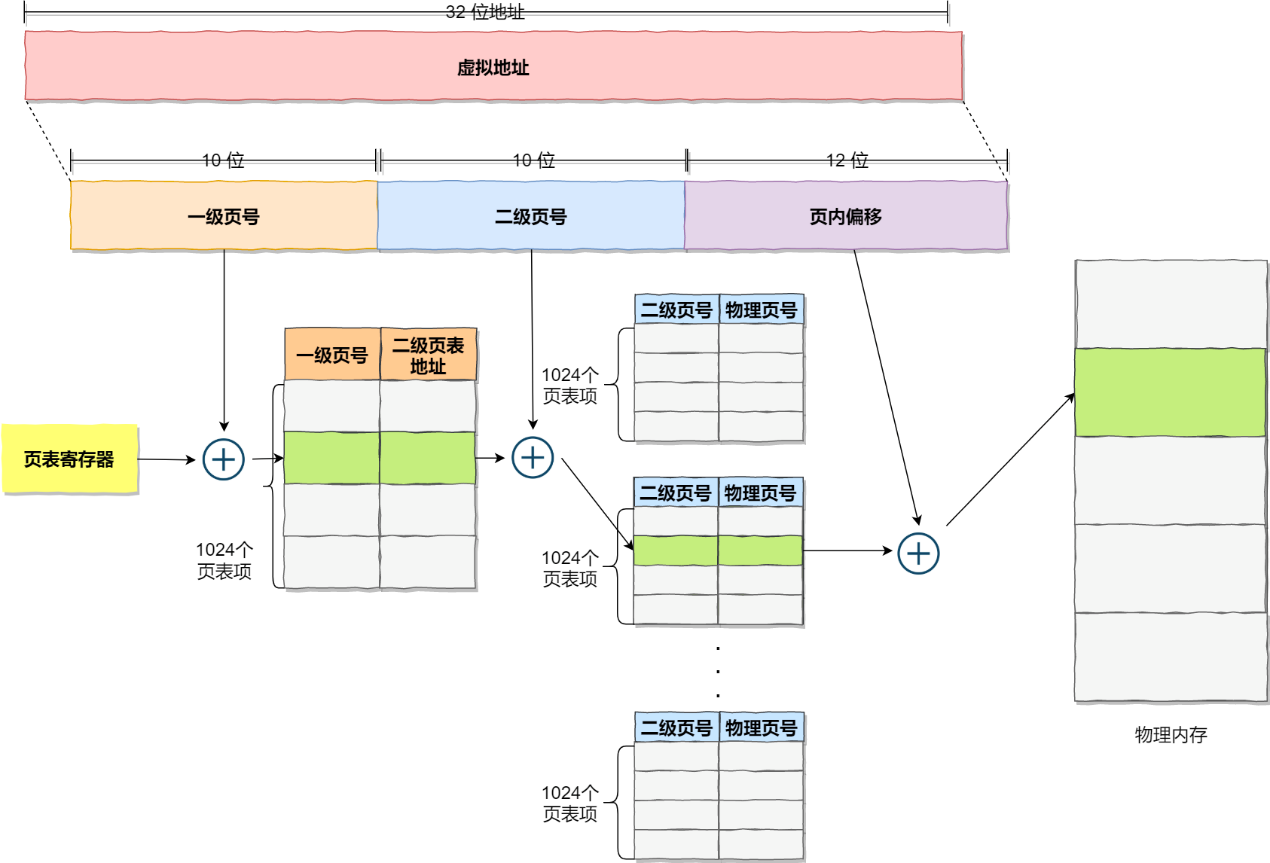
分页机制下，虚拟地址分为两部分，**页号**和**页内偏移**。页号作为页表的索引，**页表**包含物理页每页所在**物理内存的基地址**，这个基地址与页内偏移的组合就形成了物理内存地址，见下图。



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

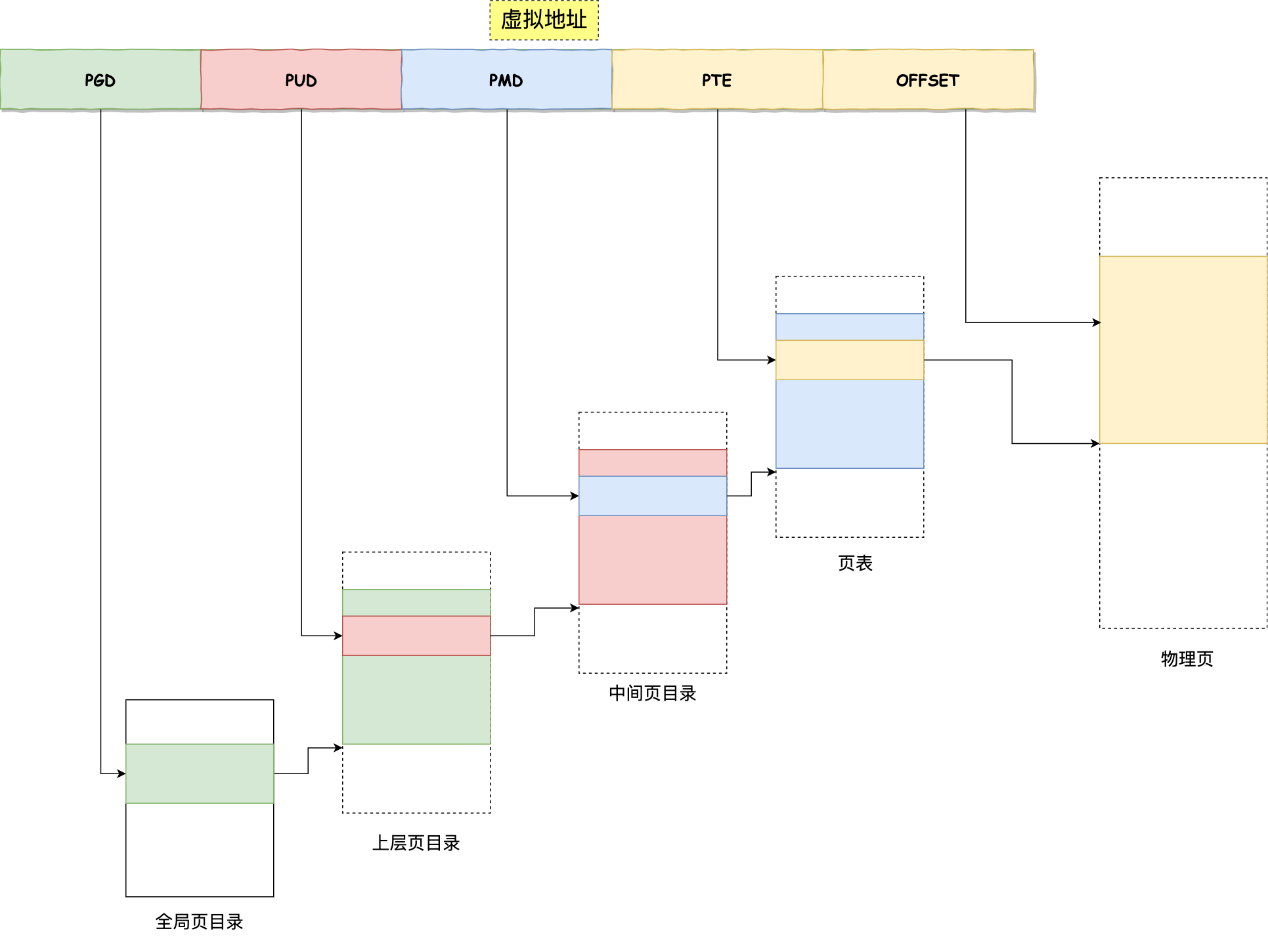


**如果某个一级页表的页表项没有被用到，也就不需要创建这个页表项对应的二级页表了，即可以在需要时才创建二级页表。**假设只有 20% 的一级页表项被用到了，那么页表占用的内存空间就只有 4KB（一级页表） + 20% \* 4MB（二级页表）= 0.804MB

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

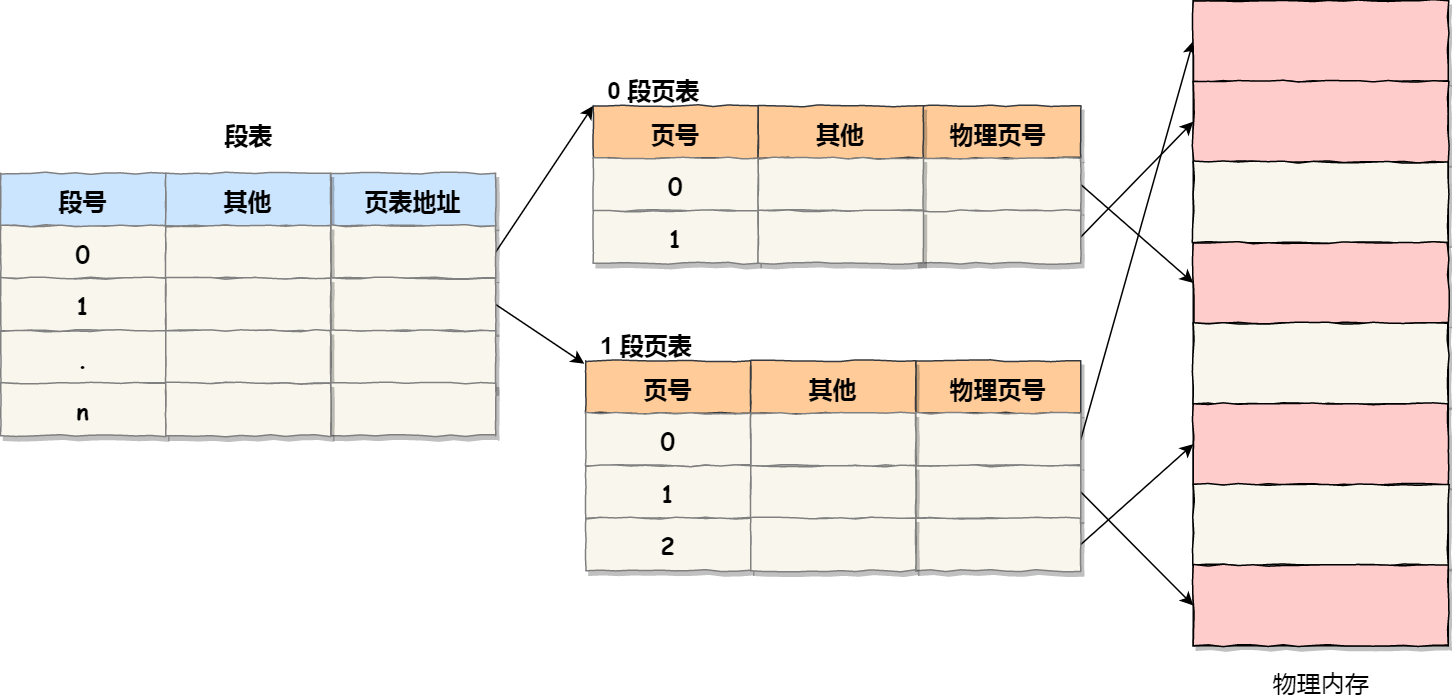
有了 TLB 后，那么 CPU 在寻址时，**会先查 TLB，如果没找到，才会继续查常规的页表**。TLB 的命中率其实是很高的，因为程序最常访问的页就那么几个。

## 段页式内存管理

段页式内存管理实现的方式：

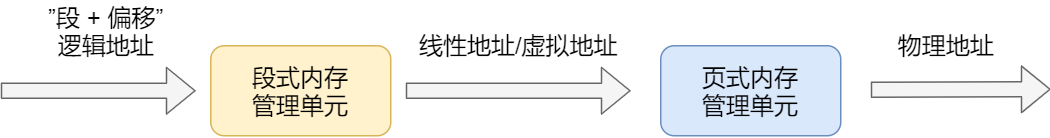
* 先将程序划分为多个有逻辑意义的段，也就是前面提到的分段机制；
* 接着再把每个段划分为多个页，也就是对分段划分出来的连续空间，再划分固定大小的页；

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

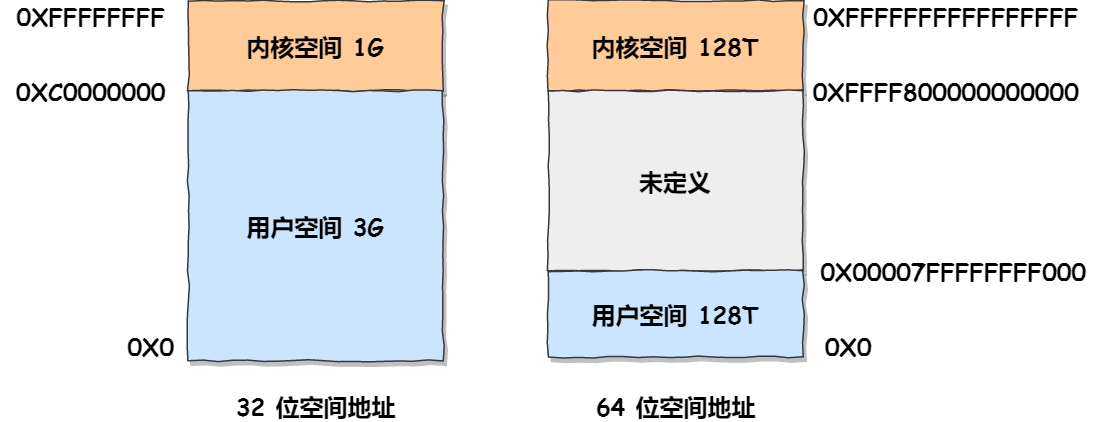
由于早期处理器的原因，**Linux 内存主要采用的是页式内存管理，但同时也不可避免地涉及了段机制**。



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

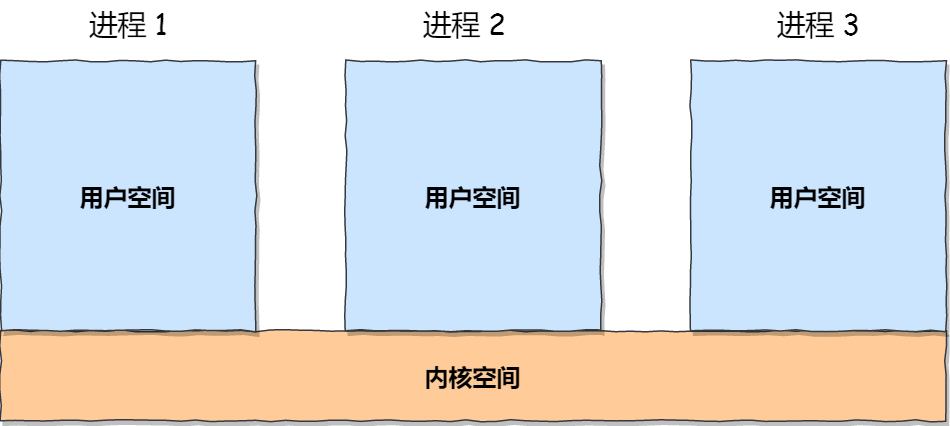
Linux 操作系统中，虚拟地址空间的内部又被分为**内核空间和用户空间**两部分，不同位数的系统，地址空间的范围也不同。比如最常见的 32 位和 64 位系统，如下所示：



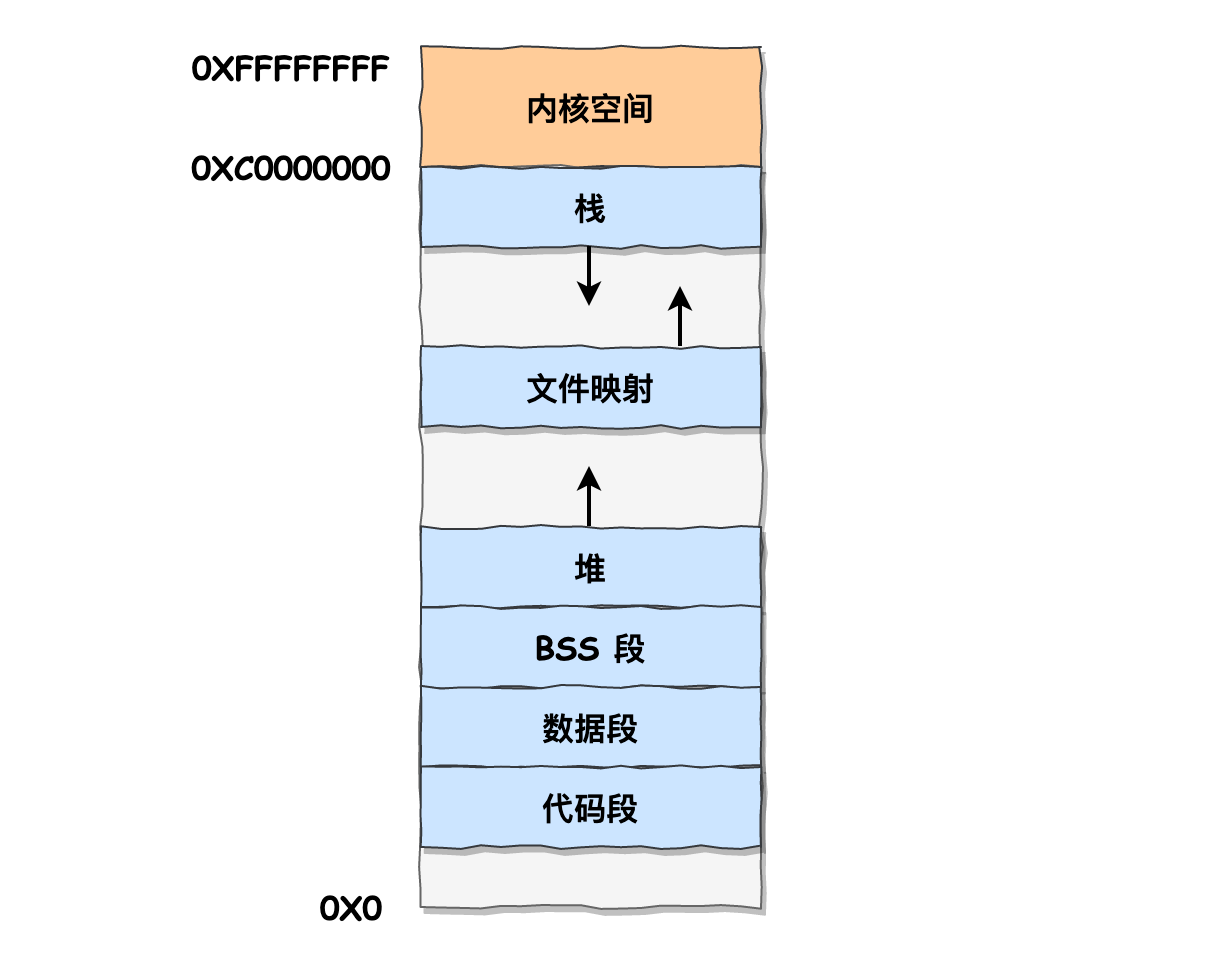
**内核空间与用户空间的区别：**

* 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" \t "_blank)）；
* 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；

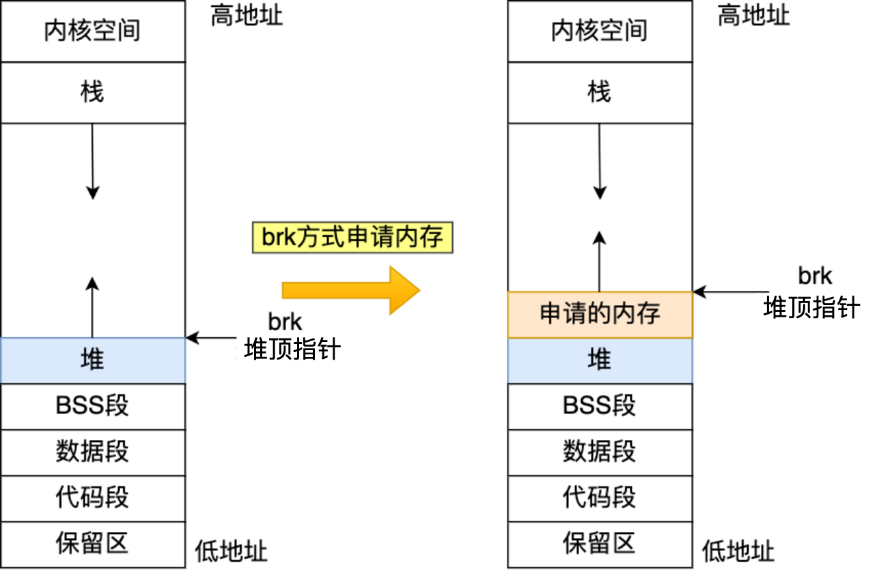
代码段下面还有一段内存空间的（灰色部分），这一块区域是「**保留区**」，之所以要有保留区这是因为在大多数的系统里，我们认为**比较小数值的地址不是一个合法地址**

# 4.2 malloc 是如何分配内存的？

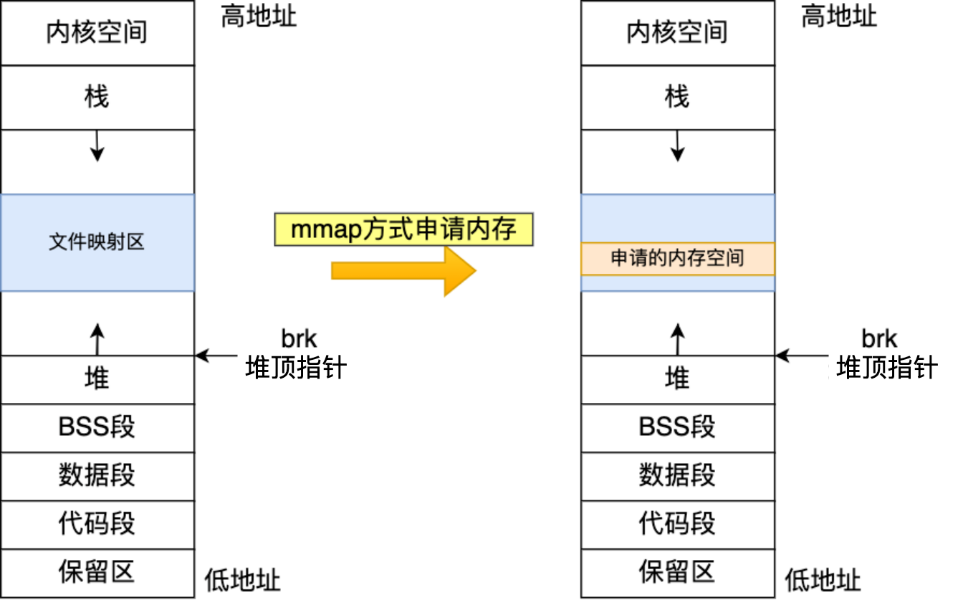
malloc() 并不是系统调用，而是 C 库里的函数，用于动态分配内存。

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。

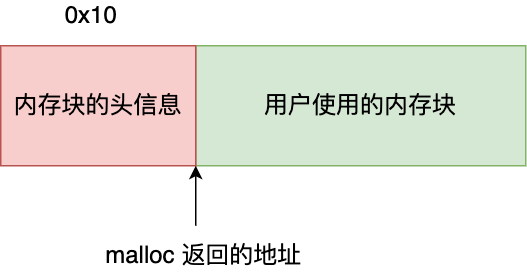
**等下次在申请内存的时候，就直接从内存池取出对应的内存块就行了，而且可能这个内存块的虚拟地址与物理地址的映射关系还存在，这样不仅减少了系统调用的次数，也减少了缺页中断的次数，这将大大降低 CPU 的消耗**。

## 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**。

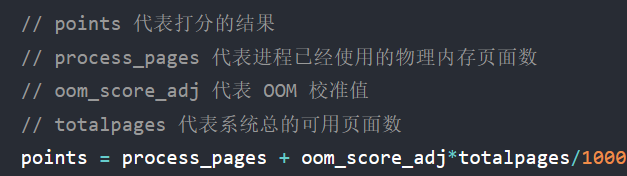
### 尽早触发 kswapd 内核线程异步回收内存

## 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。

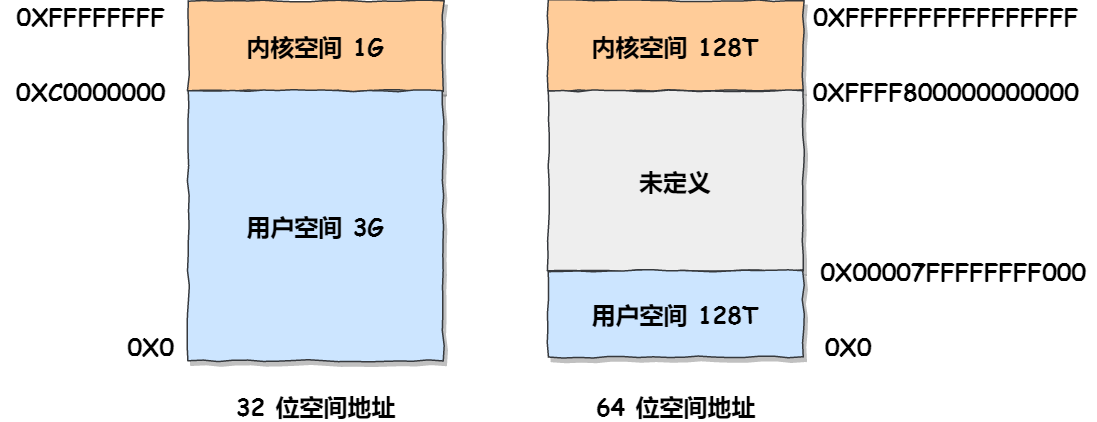
但是，不建议将我们自己的业务程序的 oom\_score\_adj 设置为 -1000，因为业务程序一旦发生了内存泄漏，而它又不能被杀掉，这就会导致随着它的内存开销变大，OOM killer 不停地被唤醒，从而把其他进程一个个给杀掉。

# 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" \l "%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" \t "_blank)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** 。