# Kernel知识点Update

## Resource

Linux内核揭秘 <https://xinqiu.gitbooks.io/linux-insides-cn/content/>

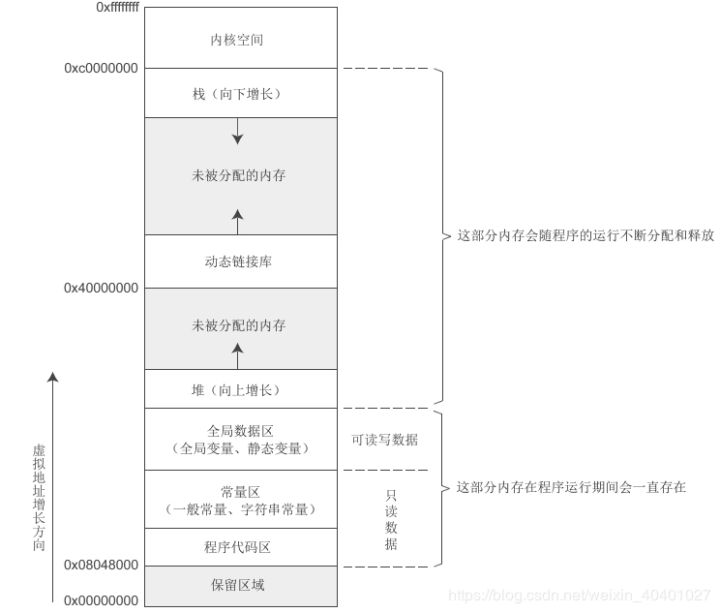
# 内存管理

## 内存地址空间分布

<https://zhuanlan.zhihu.com/p/163744831>

1、为什么32位操作系统得到虚拟内存为4G？

因为cpu是32位的，所以称为32位操作系统，即4字节。地址总线也是32位，即4字节。最大的寻址能力即为2^32B（32bit = 32/8 byte，每个内存单元占8bit）,即4G。32位系统实际能访问的量也是0x00000000到0xFFFFFFFF这之间的内存地址。内核空间会占用1G或者2G，用户使用的空间为3G或2G，Linux 默认情况下会将高地址的 1GB（2G） 空间分配给内核。



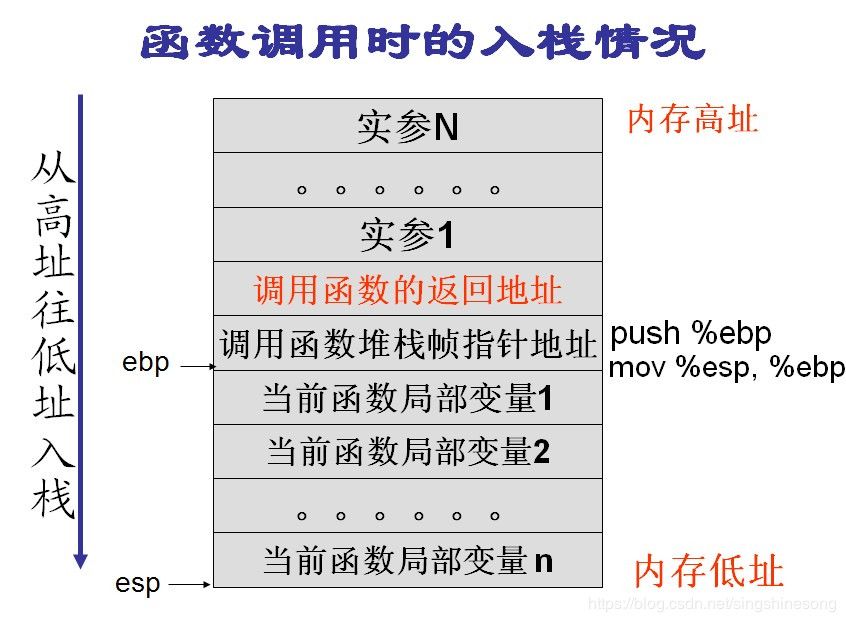
32位操作系统内存分布

C中函数参数是存放在哪的？

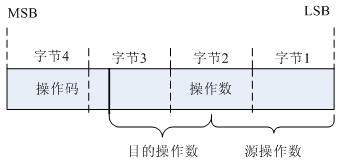
1、形参（值传递）

2、实参（指针传递、C++引用传递）

在栈区一般会存储函数的参数值、局部变量等内容。在函数调用时，堆栈内数据的排列如下图所示：



需要补充的知识点：CPU寻址方式、内存管理

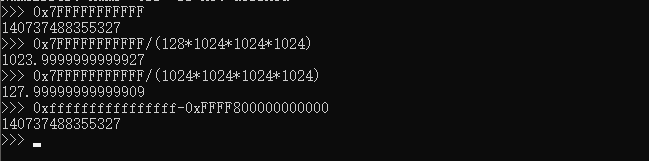
[CPU体系架构-寻址方式 - blogernice - 博客园​www.cnblogs.com](https://link.zhihu.com/?target=https://www.cnblogs.com/blogernice/articles/9875902.html" \t "https://zhuanlan.zhihu.com/p/_blank)

2、那64位操作系统的寻址能力是不是2^64B?

实际地址空间大小是2^48B，还有 16 比特暂时没有用，这样虚拟地址也有256TB。

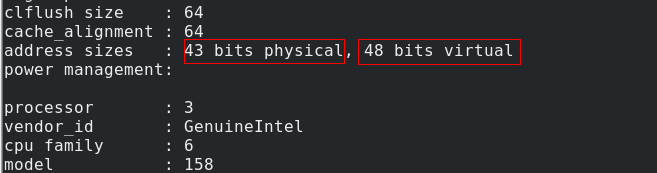
其中，0x0000000000000000~0x00007fffffffffff 表示用户空间， 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF 表示内核空间，共提供 256TB(2^48) 的寻址空间。用户和系统各128TB。

1024\*1024\*1024\*1024 = 1TB，0x00007fffffffffff=128TB



64位linux一般使用48位来表示虚拟地址空间，使用43位来表示物理地址空间，可以通过cat /proc/cpuinfo来查看（参考系统 Centos8）

cat /proc/cpuinfo



3、用户进程能访问多少物理内存？内核代码能访问多少物理内存？

32位系统用户进程最大可以访问3GB，内核代码可以访问所有物理内存。

64位系统用户进程最大可以访问超过512GB，内核代码可以访问所有物理内存。

## 问题

为何64位系统用户进程最大可以访问512GB的内存？

# 进程调度

内核在微观上，把CPU的运行时间分成许多分，然后安排给各个进程轮流运行，造成宏观上所有的进程仿佛同时在执行。双核CPU，实际上最多只能有两个进程在同时运行，大家在top、vmstat命令里看到的正在运行的进程，并不是真的在占有着CPU哈。

所以，一些设计良好的高性能进程，比如nginx，都是实际上有几颗CPU，就配几个工作进程，道理就在这。比如你的服务器有8颗CPU，那么nginx worker应当只有8个，当你多于8个时，内核可能会放超过多个nginx worker进程到1个runqueue里，会发生什么呢？就是在这颗CPU上，会比较均匀的把时间分配给这几个nginx worker，每个worker进程运行完一个时间片后，内核需要做进程切换，把正在运行的进程上下文保存下来。假设内核分配的时间片是100ms，做进程切换的时间是5ms，那么进程性能下降还是很明显的，跟你配置的worker有关，越多下降得越厉害。

当然，这是跟nginx的设计有关的。nginx是事件驱动的全异步进程，本身设计上就几乎不存在阻塞和中断，nginx的设计者就希望每一个nginx worker可以独占CPU的几乎全部时间片，这点就是nginx worker数量配置的依据所在。

当然，实际的运行进程里，大部分并不是nginx这种希望独占CPU全部时间片的进程，许多进程，比如vi，它在很多时间是在等待用户输入，这时vi在等待IO中断，是不占用时间片的，内核面对多样化的进程，就需要技巧性的分配CPU时间片了。

内核分配时间片是有策略和倾向性的。换句话说，内核是偏心的，它喜欢的是IO消耗型进程，因为这类进程如果不能及时响应，用户就会很不爽，所以它总会下意识的多分配CPU运行时间给这类进程。而CPU消耗进程内核就不太关心了。这有道理吗？太有了，CPU消耗型慢一点用户感知不出来，电信号和生物信号运转速度差距巨大。虽然内核尽量多的分配时间片给IO消耗型进程，但IO消耗进程常常在睡觉，给它的时间片根本用不掉。很合理吧？

那么内核具体是怎么实现这种偏心呢？通过动态调整进程的优先级，以及分配不同长短的CPU时间处来实现。先说内核如何决定时间片的长度。

对每一个进程，有一个整型static\_prio表示用户设置的静态优先级，内核里它与nice值是对应的。

[https://cloud.tencent.com/developer/article/1449443https://cloud.tencent.com/developer/article/1449443](https://cloud.tencent.com/developer/article/1449443https:/cloud.tencent.com/developer/article/1449443)

## 中断

1. 中断vs轮询

Linux 内核需要对连接到计算机上的所有硬件设备进行管理，毫无疑问这是它的份内事。如果要管理这些设备，首先得和它们互相通信才行，一般有两种方案可实现这种功能：

中断（interrupt） 让硬件在需要的时候向内核发出信号（变内核主动为硬件主动）。

中断的好处是响应及时，如果数据量较小，则不会占用太多的CPU事件；缺点是数据量大时，会产生过多中断，而每个中断都要消耗不少的CPU时间，从而导致效率反而不如轮询高。

轮询（polling） 让内核定期对设备的状态进行查询，然后做出相应的处理；

轮询方式与中断方式相反，它更适合处理大量数据，因为每次轮询不需要消耗过多的CPU时间；缺点是即使只接收很少数据或不接收数据时，也要占用CPU时间。

轮询中断结合

NAPI是两者的结合，数据量低时采用中断，数据量高时采用轮询。平时是中断方式，当有数据到达时，会触发中断处理函数执行，中断处理函数关闭中断开始处理。如果此时有数据到达，则没必要再触发中断了，因为中断处理函数中会轮询处理数据，直到没有新数据时才打开中断。

从物理学的角度看，中断是一种电信号，由硬件设备产生，并直接送入中断控制器（如 8259A）的输入引脚上，然后再由中断控制器向处理器发送相应的信号。处理器一经检测到该信号，便中断自己当前正在处理的工作，转而去处理中断。此后，处理器会通知 OS 已经产生中断。这样，OS 就可以对这个中断进行适当的处理。不同的设备对应的中断不同，而每个中断都通过一个唯一的数字标识，这些值通常被称为中断请求线。

2. 中断vs异常

中断可分为同步（synchronous）中断和异步（asynchronous）中断：

同步中断是当指令执行时由 CPU 控制单元产生，之所以称为同步，是因为只有在一条指令执行完毕后 CPU 才会发出中断，而不是发生在代码指令执行期间，比如系统调用。

异步中断是指由其他硬件设备依照 CPU 时钟信号随机产生，即意味着中断能够在指令之间发生，例如键盘中断。

根据 Intel 官方资料，同步中断称为异常（exception），异步中断被称为中断（interrupt）。

中断可分为可屏蔽中断（Maskable interrupt）和非屏蔽中断（Nomaskable interrupt）。

异常可分为故障（fault）、陷阱（trap）、终止（abort）三类。

3. 中断上下文 vs 进程上下文

当执行一个中断处理程序的时候，内核处于中断上下文，与进程没啥关系，没有后备进程，所以不可以睡眠。

中断处理程序打断了其它的代码（甚至可能打断另一中断处理程序，以及软中断），所以必须快速，简洁。

4. 上半部 vs下半部

中断处理程序要快！ VS 中断处理程序完成的工作量多!

把中断处理切为两个部分，

上半部是接收到中断就立即执行，但是只做有严格时限的工作

能够允许稍后完成的工作会推迟到下半部，在合适的时机，下半部会开中断执行

5. 中断下半部–软中断 vs tasklet vs workqueue

软中断

产生后并不是马上可以执行，必须要等待内核的调度才能执行。软中断不能被自己打断(即单个cpu上软中断不能嵌套执行)，只能被硬件中断打断（上半部）。

可以并发运行在多个CPU上（即使同一类型的也可以）。所以软中断必须设计为可重入的函数（允许多个CPU同时操作），因此也需要使用自旋锁来保其数据结构。

static struct softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp;

enum

{

HI\_SOFTIRQ=0, /\*用于高优先级的tasklet\*/

TIMER\_SOFTIRQ, /\*用于定时器的下半部\*/

NET\_TX\_SOFTIRQ, /\*用于网络层发包\*/

NET\_RX\_SOFTIRQ, /\*用于网络层收报\*/

BLOCK\_SOFTIRQ,

BLOCK\_IOPOLL\_SOFTIRQ,

TASKLET\_SOFTIRQ, /\*用于低优先级的tasklet\*/

SCHED\_SOFTIRQ,

HRTIMER\_SOFTIRQ,

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

NR\_SOFTIRQS

};

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*)) // 注册软中断

void raise\_softirq(unsigned int nr) // 触发软中断

do\_softirq-->\_\_do\_softirq // 执行软中断

## tasklet

由于软中断必须使用可重入函数，这就导致设计上的复杂度变高，作为设备驱动程序的开发者来说，增加了负担。而如果某种应用并不需要在多个CPU上并行执行，那么软中断其实是没有必要的。因此诞生了弥补以上两个要求的tasklet。

它具有以下特性：

a）一种特定类型的tasklet只能运行在一个CPU上，不能并行，只能串行执行。

b）多个不同类型的tasklet可以并行在多个CPU上。

c）软中断是静态分配的，在内核编译好之后，就不能改变。但tasklet就灵活许多，可以在运行时改变（比如添加模块时）。

tasklet是在两种软中断类型的基础上实现的，因此如果不需要软中断的并行特性，tasklet就是最好的选择。也就是说tasklet是软中断的一种特殊用法，即延迟情况下的串行执行。

工作队列

软中断和tasklet都是运行在中断上下文中，它们与任一进程无关，没有支持的进程完成重新调度。所以软中断和tasklet不能睡眠、不能阻塞，它们的代码中不能含有导致睡眠的动作，如减少信号量、从用户空间拷贝数据或手工分配内存等。

工作队列是Linux 2.6 内核中新增加的一种下半部机制。它与其它几种下半部分机制最大的区别就是它可以把工作推后，交由一个内核线程去执行。内核线程只在内核空间运行，没有自己的用户空间，它和普通进程一样可以被调度，也可以被抢占。该工作队列总是会在进程上下文执行。这样，通过工作队列执行的代码能占尽进程上下文的所有优势，最重要的就是工作队列允许重新调度甚至是睡眠。

如果推后执行的任务需要睡眠，那么就选择工作队列；

如果推后执行的任务不需要睡眠，那么就选择tasklet。

实际上，工作队列的本质就是将工作交给内核线程处理，因此其可以用内核线程替换。但是内核线程的创建和销毁对编程者的要求较高，而工作队列实现了内核线程的封装，不易出错，所以我们也推荐使用工作队列。

————————————————

版权声明：本文为CSDN博主「网络安全研发随想」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/gengzhikui1992/article/details/85282005>

## 可重入函数

一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误；而不可重入的函数由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题，这类函数是不能运行在多任务环境下的。

————————————————

版权声明：本文为CSDN博主「Mike江」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/tennysonsky/article/details/45127125>

## 用户抢占和内核抢占

<https://www.cnblogs.com/linhaostudy/p/9888910.html>

# 时钟管理

ntp时钟调整策略

<https://blog.csdn.net/varyall/article/details/80220418>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Offset*（与服务器的时间差） | *0-128ms* | *128ms-600s* | *600s-1000s* | *1000s*以上 |
| 使用*-X*参数 | 微调 | 微调（速度大约是*0.5ms/s*，调整*600*秒要*14*天左右） | 跳跃 | 退出（加*-g*参数可忽略） |
| 不使用*-X*参数 | 微调 | 跳跃 | 跳跃 | 退出（加*-g*参数可忽略） |

深度解析CephX原理—调节NTP时钟的困境

<https://hackerain.me/2019/12/15/ceph/cephx.html>