# Leo_wlCnBlogs


**导航**

博客园
首页
新随笔
联系
订阅XML
管理



‹ 2018年8月 ›

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 29 | 30 | 31 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |



**公告**

**我的标签**

**2014 2013下 2013上 2012下 LVS中文 开源 反向代理 CUDA**




# GC算法精解（五分钟让你彻底明白标记/清除算法）

**阅读目录**



**回到目录**

## GC算法精解（五分钟让你彻底明白标记/清除算法）

相信不少猿友看到标题就认为LZ是标题党了，不过既然您已经被LZ忽悠进来了，那就好好的享受一顿算法大餐吧。不过LZ丑话说前面哦，这篇文章应该能让各位彻底理解标记/清除算法，不过倘若各位猿友不能在五分钟内看完，那就不是LZ的错啦。

好了，前面只是小小开个玩笑，让各位猿友放松下心情。下面即将与各位分享的，是GC算法中最基础的算法------**标记/清除算法**。如果搞清楚这个算法，那么后面两个就完全是小菜一碟了。

首先，我们回想一下上一章提到的根搜索算法，它可以解决我们应该回收哪些对象的问题，但是它显然还不能承担垃圾搜集的重任，**因为我们在程序（程序也就是指我们运行在JVM上的JAVA程序）运行期间如果想进行垃圾回收，就必须让GC线程与程序当中的线程互相配合，才能在不影响程序运行的前提下，顺利的将垃圾进行回收。**

为了达到这个目的，标记/清除算法就应运而生了。**它的做法是当堆中的有效内存空间（available memory）被耗尽的时候，就会停止整个程序（也被成为stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。**

下面LZ具体解释一下标记和清除分别都会做些什么。


**统计**

随笔 - 16726
文章 - 28
评论 - 1641
引用 - 0

**我的好友**

Artech
asp.net
Banq
DNN
Domain Driven Design
hacker2012
infoworld
Jianqiang Bao
jv9
Manavi
Martin Fowler
Muhammad Mosa
MVP
programmer
ScottGu博客[英文]
ScottGu博客[中文]
sql mag
wcf
阿捷
冯大辉
侯伯薇
吉日嘎拉



昵称：**HackerVirus**
园龄：**8年8个月**
粉丝：**2892**
关注：**247**
**+加关注**

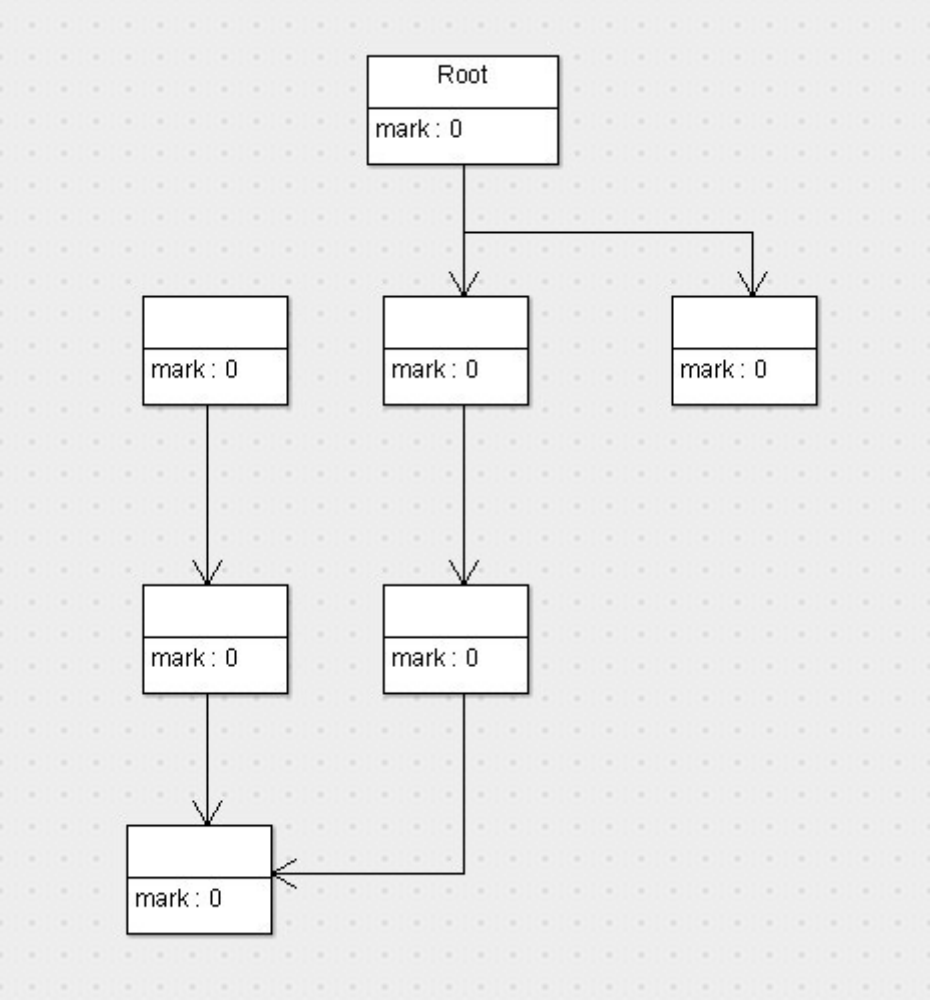
**标记：标记的过程其实就是，遍历所有的GC Roots，然后将所有GC Roots可达的对象标记为存活的对象。**

**清除：清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉。**

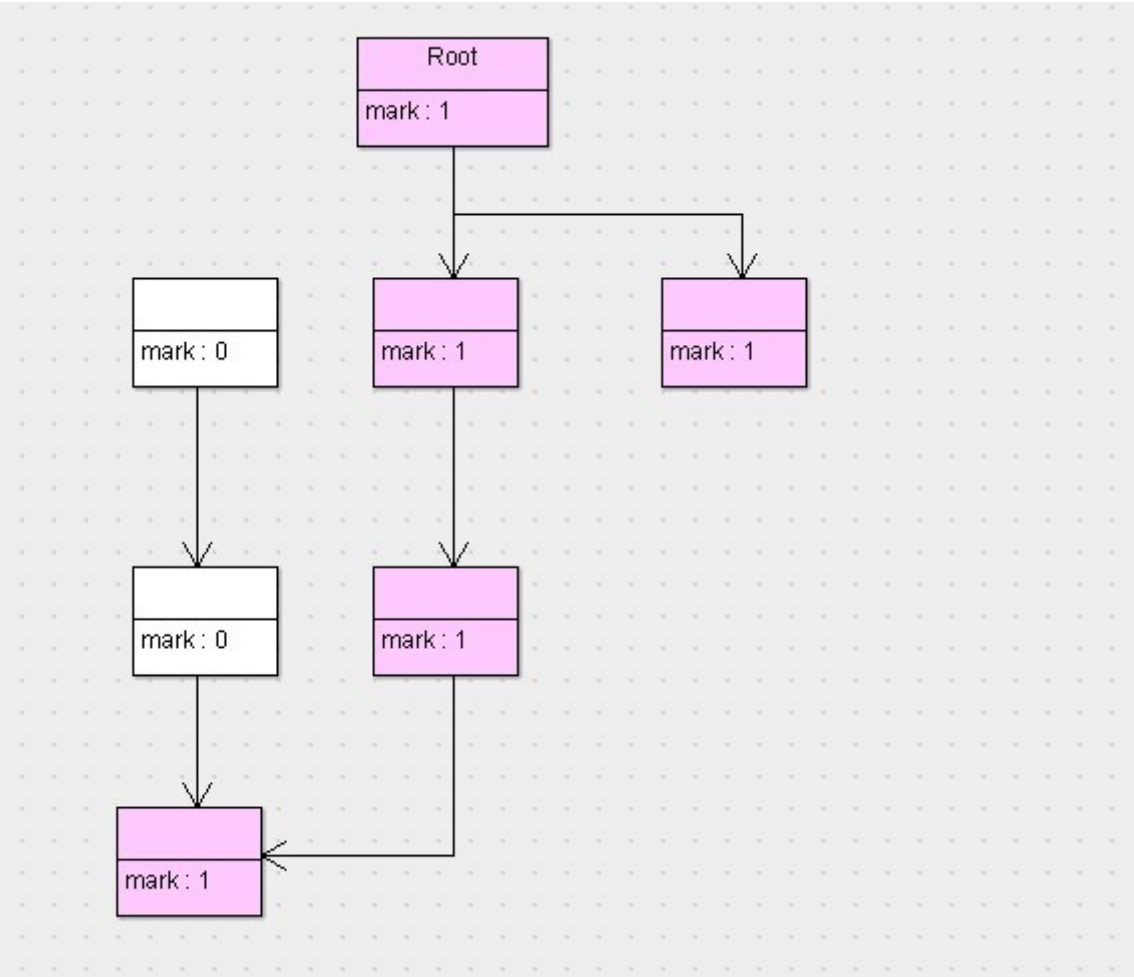
其实这两个步骤并不是特别复杂，也很容易理解。LZ用通俗的话解释一下标记/清除算法，**就是当程序运行期间，若可以使用的内存被耗尽的时候，GC线程就会被触发并将程序暂停，随后将依旧存活的对象标记一遍，最终再将堆中所有没被标记的对象全部清除掉，接下来便让程序恢复运行。**

下面LZ给各位制作了一组描述上面过程的图片，结合着图片，我们来直观的看下这一过程，首先是第一张图。

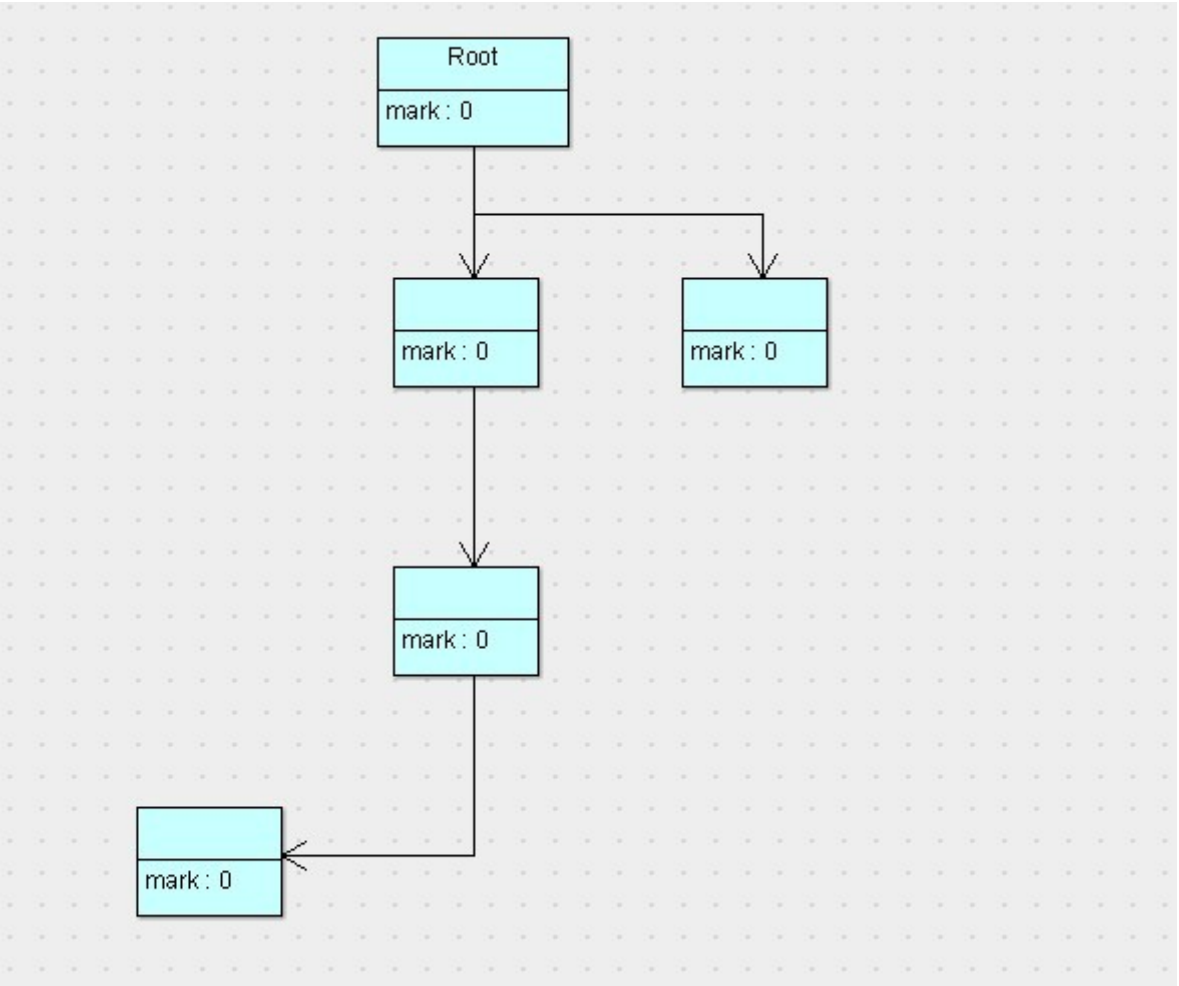
李天平
李永京
灵动生活
刘铁猛
秋色园
圣殿骑士
图灵书籍
微软论坛
伍迷
小洋（燕洋天）
徐磊
徐明璐
张逸
赵劼 Jeffrey Zhao
周 金根
周雪峰
周银辉




这张图代表的是程序运行期间所有对象的状态，**它们的标志位全部是0**（也就是未标记，以下默认0就是未标记，1为已标记），假设这会儿**有效内存空间耗尽**了，JVM将会停止应用程序的运行并开启GC线程，然后开始进行标记工作，按照根搜索算法，标记完以后，对象的状态如下图。




可以看到，按照根搜索算法，所有从root对象可达的对象就被标记为了存活的对象，此时已经完成了第一阶段标记。接下来，就要执行第二阶段清除了，那么清除完以后，剩下的对象以及对象的状态如下图所示。




可以看到，没有被标记的对象将会回收清除掉，而被标记的对象将会留下，并且会将标记位重新归0。接下来就不用说了，唤醒停止的程序线程，让程序继续运行即可。

其实这一过程并不复杂，甚至可以说非常简单，各位说对吗。不过其中有一点值得LZ一提，就是为什么非要停止程序的运行呢?

这个其实也不难理解，LZ举个最简单的例子，假设我们的程序与GC线程是一起运行的，各位试想这样一种场景。

假设我们刚标记完图中最右边的那个对象，暂且记为A，结果此时在程序当中又new了一个新对象B，且A对象可以到达B对象。但是由于此时A对象已经标记结束，B对象此时的标记位依然是0，因为它错过了标记阶段。因此当接下来轮到清除阶段的时候，新对象B将会被苦逼的清除掉。如此一来，不难想象结果，GC线程将会导致程序无法正常工作。

上面的结果当然令人无法接受，我们刚new了一个对象，结果经过一次GC，忽然变成null了，这还怎么玩？

**回到目录**

## 用户空间与内核空间，进程上下文与中断上下文[总结]

到此为止，标记/清除算法LZ已经介绍完了，下面我们来看下它的缺点，其实了解完它的算法原理，它的缺点就很好理解了。

1、首先，**它的缺点就是效率比较低（递归与全堆对象遍历），而且在进行GC的时候，需要停止应用程序，这会导致用户体验非常差劲**，尤其对于交互式的应用程序来说简直是无法接受。试想一下，如果你玩一个网站，这个网站一个小时就挂五分钟，你还玩吗？

2、第二点主要的缺点，则是**这种方式清理出来的空闲内存是不连续的**，这点不难理解，我们的死亡对象都是随即的出现在内存的各个角落的，现在把它们清除之后，内存的布局自然会乱七八糟。而为了应付这一点，JVM就不得不维持一个内存的空闲列表，这又是一种开销。而且在分配数组对象的时候，寻找连续的内存空间会不太好找。

看完它的缺点估计有的猿友要忍不住吐糟了，“这么说这个算法根本没法用嘛，那LZ还介绍这么个玩意干什么。”

猿友们莫要着急，一个算法有缺点，高人们自然会想尽办法去完善它的。而接下来我们要介绍的两种算法，皆是在标记/清除算法的基础上优化而产生的。具体的内容，下一次LZ再和各位分享。

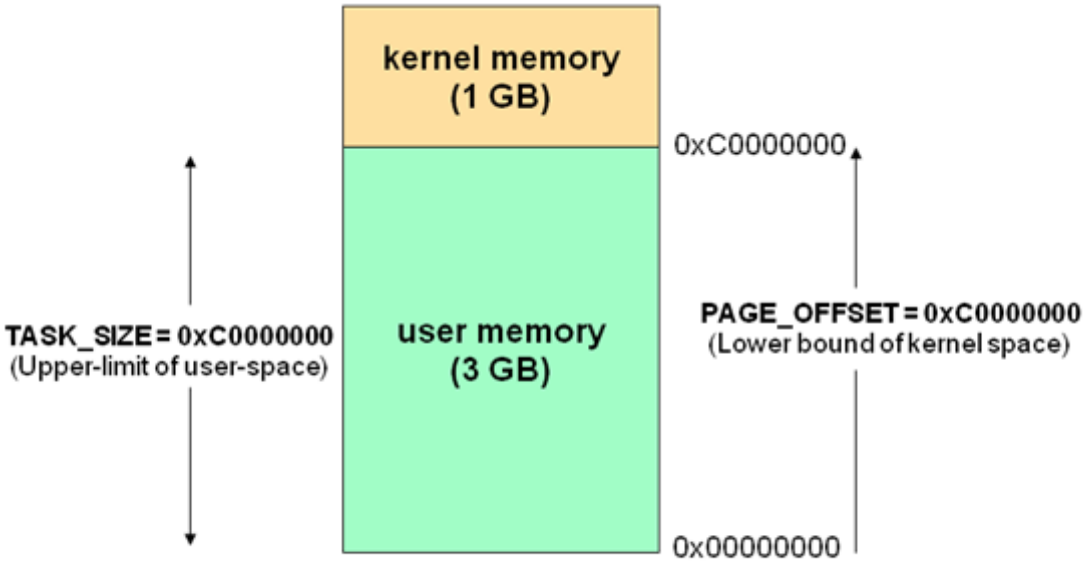
本次的分享就到此结束了，希望各位看完都能有所收获，0.0。

### 1、前言

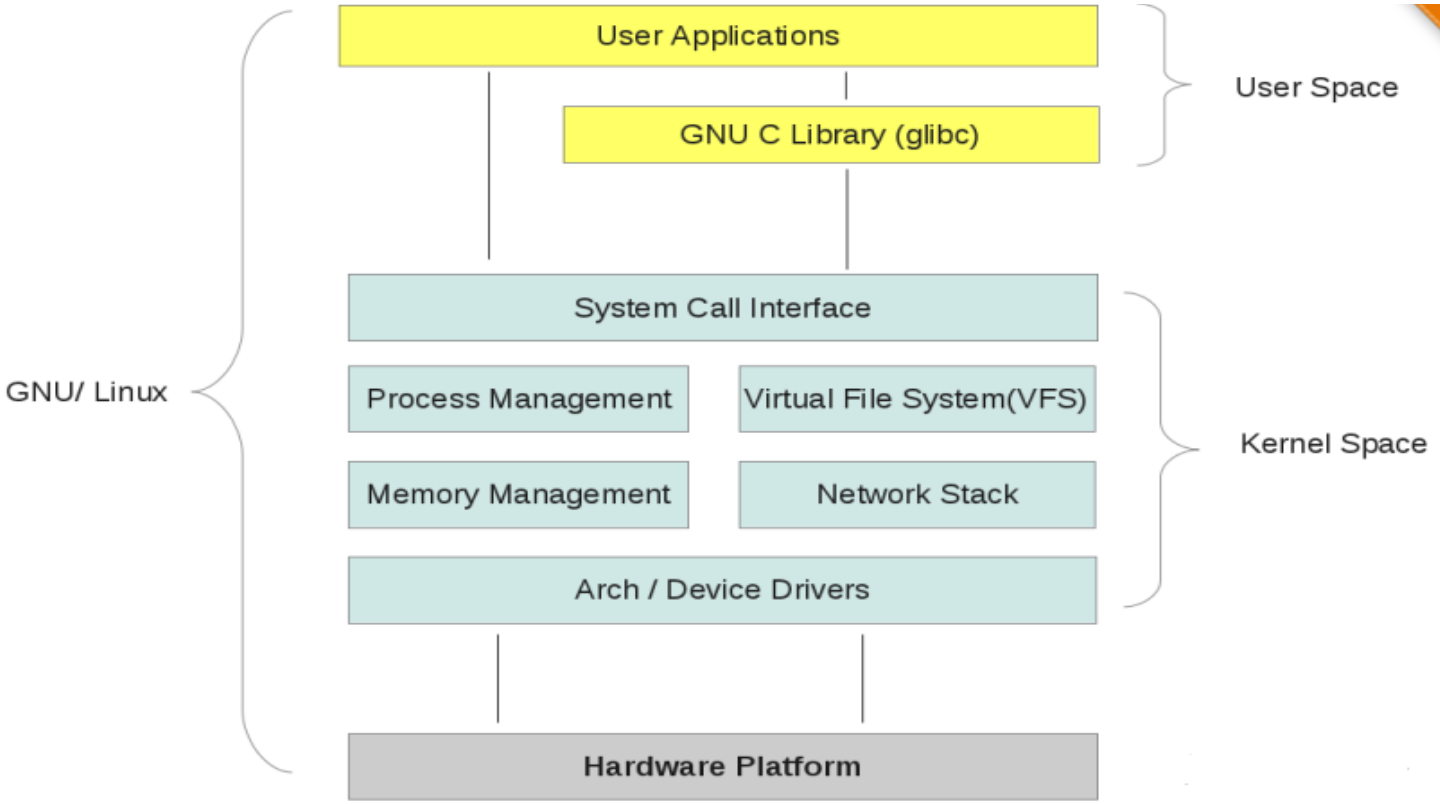
最近在学习linux内核方面的知识，经常会看到用户空间与内核空间及进程上下文与中断上下文。看着很熟悉，半天又说不出到底是怎么回事，有什么区别。看书过程经常被感觉欺骗，似懂非懂的感觉，很是不爽，今天好好结合书和网上的资料总结一下，加深理解。

## 2、用户空间与内核空间

我们知道现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操心系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核，保证内核的安全，操心系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间。每个进程可以通过系统调用进入内核，因此，Linux内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有4G字节的虚拟空间。空间分配如下图所示：




**有了用户空间和内核空间，整个linux内部结构可以分为三部分，从最底层到最上层依次是：硬件-->内核空间-->用户空间。**如下图所示：




**需要注意的细节问题：**

（1） 内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。不管是内核空间还是用户空间，它们都处于虚拟空间中。

（2） Linux使用两级保护机制：0级供内核使用，3级供用户程序使用。

**内核态与用户态：**

（1）**当一个任务（进程）执行系统调用而陷入内核代码中执行时，称进程处于内核运行态（内核态）。**此时处理器处于特权级最高的（0级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。

（2）**当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。**此时处理器在特权级最低的（3级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此

时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。

**参考资料：**

**http://blog.csdn.net/f22jay/article/details/7925531**

**http://blog.csdn.net/zhangskd/article/details/6956638**

**http://blog.chinaunix.net/uid-26838492-id-3162146.html**

**3、进程上下文与中断上下文**

我在看《linux内核设计与实现》这本书的第三章进程管理时候，看到进程上下文。书中说当一个程序执行了系统调用或者触发某个异常（软中断），此时就会陷入内核空间，内核此时代表进程执行，并处于进程上下文中。看后还是没有弄清楚，什么是进程上下文，如何上google上面狂搜一把，总结如下：

程序在执行过程中通常有用户态和内核态两种状态，CPU对处于内核态根据上下文环境进一步细分，因此有了下面三种状态：

（1）内核态，运行于进程上下文，内核代表进程运行于内核空间。
（2）内核态，运行于中断上下文，内核代表硬件运行于内核空间。
（3）用户态，运行于用户空间。

**上下文context： 上下文简单说来就是一个环境。**

用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存 器值、变量等。**所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。**

相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存信息等。一个进程的上下文可以分为三个部分:用户级上下文、寄存器上下文以及系统级上下文。

（1）用户级上下文: 正文、数据、用户堆栈以及共享存储区；
（2）寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

（3）系统级上下文: 进程控制块task_struct、内存管理信息(mm_struct、vm_area_struct、pgd、pte)、内核栈。

**当发生进程调度时，进行进程切换就是上下文切换(context switch).操作系统必须对上面提到的全部信息进行切换，新调度的进程才能运行。而系统调用进行的模式切换(mode switch)。模式切换与进程切换比较起来，容易很多，而且节省时间，因为模式切换最主要的任务只是切换进程寄存器上下文的切换。**

硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。**所谓的“ 中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。**中断时，内核不代表任何进程运行，它一般只访问系统空间，而不会访问进程空间，内核在中断上下文中执行时一般不会阻塞。

**摘录Linux注释的内容如下:**

**Process Context**

**----------**

**One of the most important parts of a process is the executing program code. This code is read in from an executable file and executed within the program's address space. Normal program execution occurs in user-space. When a program executes a system call or triggers an exception, it enters kernel-space. At this point, the kernel is said to be "executing on behalf of the process" and is in process context. When in process context, the current macro is valid[7]. Upon exiting the kernel, the process resumes execution in user-space, unless a higher-priority process has become runnable in the interim(过渡期), in which case the scheduler is invoked to select the higher priority process.**

**Other than process context there is interrupt context, In interrupt context, the system is not running on behalf of a process, but is executing an interrupt handler. There is no process tied to interrupt handlers and consequently no process context.**

**System calls and exception handlers are well-defined interfaces into the**

**kernel. A process can begin executing in kernel-space only through one of these interfaces -- all access to the kernel is through these interfaces.**

**----------**

**Interrupt Context**

**----------**

**When executing an interrupt handler or bottom half, the kernel is in interrupt context. Recall that process context is the mode of operation the kernel is in while it is executing on behalf of a process -- for example, executing a system call or running a kernel thread. In process context, the current macro points to the associated task. Furthermore, because a process is coupled to the kernel in process context(因为进程是以进程上文的形式连接到内核中的), process context can sleep or otherwise invoke the scheduler.**

**Interrupt context, on the other hand, is not associated with a process. The current macro is not relevant (although it points to the interrupted process). Without a backing process(由于没有进程的背景),interrupt context cannot sleep -- how would it ever reschedule?(否则怎么再对它重新调度?) Therefore, you cannot call certain functions from interrupt context. If a function sleeps, you cannot use it from your interrupt handler -- this limits the functions that one can call from an interrupt handler.(这是对什么样的函数可以在中断处理程序中使用的限制)**

**Interrupt context is time critical because the interrupt handler interrupts other code. Code should be quick and simple. Busy looping is discouraged. This is a very important point; always keep in mind that your interrupt handler has interrupted other code (possibly even another interrupt handler on a different line!). Because of this asynchronous nature, it is imperative(必须) that all interrupt handlers be as quick and as simple as possible. As much as possible, work should be pushed out from the interrupt handler and performed in a bottom half, which runs at a more convenient time.**

**The setup of an interrupt handler's stacks is a configuration option.**

**Historically, interrupt handlers did not receive(拥有) their own stacks. Instead, they would share the stack of the process that they interrupted[1]. The kernel stack is two pages in size; typically, that is 8KB on 32-bit architectures and 16KB on 64-bit architectures. Because in this setup interrupt handlers share the stack, they must be exceptionally frugal(必须非常节省) with what data they allocate there. Of course, the kernel stack is limited to begin with, so all kernel code should be cautious.**

**A process is always running. When nothing else is schedulable, the idle task runs.**

**----------**

**LINUX完全注释中的一段话：**

**当一个进程在执行时,CPU的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够必得到切换时的状态执行下去。在LINUX中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时,内核就在被中断进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到的资源，以便中继服务结束时能恢复被中断进程的执行。**

**参考资料**:

**http://www.cnblogs.com/hustcat/articles/1505618.html**

**http://mprc.pku.edu.cn/~zhengyansong/blog/?p=199**

**http://blog.chinaunix.net/uid-26980210-id-3235544.html**

分类: **[25]JAVA**

« 上一篇：**wp7之换肤原理简单分析**
» 下一篇：**性能和优化相关**
posted on 2013-08-20 10:16 **HackerVirus** 阅读(6250) 评论(1) **编辑** **收藏**

**评论**

**#1楼**

**刺猬心声** Posted @ 2017-10-25 22:04

特地登录来吐槽，排版真烂。

支持(1) 反对(0)

刷新评论 刷新页面 返回顶部

**注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。**



**最新IT新闻**:

· **大脑皮层中发现新型脑细胞**
· **惠普企业公布第三财季财报：净利同比增58.6%**
· **程维的三句话 决定了滴滴的命运**
· **“第一性原理“不应该是玄学**
· **土巴兔递交赴港上市招股书：去年经调整盈利6350万**

» **更多新闻...**



**最新知识库文章**: