**Linux的任督二脉：进程调度和内存管理**

原创 2017年08月23日 14:45:09

* 标签：
* [Linux 进程调度 内存管理](http://so.csdn.net/so/search/s.do?q=Linux%20%E8%BF%9B%E7%A8%8B%E8%B0%83%E5%BA%A6%20%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86&t=blog)
* 16177

《穆赫兰道》与《内陆帝国》

我在多年的工程生涯中发现很多工程师碰到一个共性的问题：Linux工程师很多，甚至有很多有多年工作经验，但是对一些关键概念的理解非常模糊，比如不理解CPU、内存资源等的真正分布，具体的工作机制，这使得他们对很多问题的分析都摸不到方向。比如进程的调度延时是多少？Linux能否硬实时？多核下多线程如何执行？系统的内存究竟耗到哪里去了？我写的应用程序究竟耗了多少内存？什么是内存泄漏，如何判定内存是否真的泄漏？CPU速度、内存大小和系统性能的关联究竟是什么？内存和I/O存在着怎样的千丝万缕的联系？

若不能回答上述问题，势必造成Linux开发过程中的抓瞎，出现关键bug和性能问题后丈二摸不着。从某种意义上来说，进程调度和内存管理之于Linux，类似任督两脉之于人体。任督两脉属于奇经八脉，任脉主血，为阴脉之海；督脉主气，为阳脉之海。任督两脉分别对十二正经脉中的手足六阴经与六阳经脉起着主导作用，任督通则百脉皆通。对进程调度和内存管理的理解，可以极大地打通我们对Linux系统架构，性能瓶颈，进程资源消耗等一系列问题的理解。

但是，对这两个知识点的理解，本身有一定的难度，尤其是内存管理，看资料都很难看懂。若调度器是悬疑惊悚片鬼才大卫·林奇的《穆赫兰道》,内存管理则极似他的《内陆帝国》，为Linux最晦涩的部分。坦白讲，《穆赫兰道》给我的感觉是晦涩而惊艳，而《内陆帝国》让我感觉到自己在吃屎，实在是只有阴暗、晦涩、看不到希望。

我在学习Linux内存管理的时候，同样有看《内陆帝国》的强烈不愉悦感，整部电影构造的弗洛伊德《梦的解析》的世界有太多苍白的细节，沉闷的对白，阴暗的画面，而没有一个最初层叠的整体概念。逃离这个噩梦，唯一的方法，我们势必应该以一种最简单可靠地方式来理解进程调度和内存管理的精髓，这个时候，细节已经显得不那么重要，而concept则需要吃透再吃透。很多人读Linux的书陷入了纷繁芜杂的细节，而没有理解concept，这个时候，细节会显得那么苍白无力和流离失所。所以，我们更有必要明确每一个工作机制，以及这些工作机制背后的原因，此后，细节只是一个具体的实现。细节是会变的，唯概念不破。

带着问题上路

一切的学习都是为了解决问题，而不是为了学习而学习。为了学习而学习，这种行为实在是太傻了，因为最终也学不好。所以我们要弄清楚进程调度和内存管理究竟能解决什么样的问题。

Linux进程调度以及配套的进程管理回答如下问题：

1.    Linux进程和线程如何创建、退出？进程退出的时候，自己没有释放的资源（如内存没有free）会怎样？

2.    什么是写时拷贝？

3.    Linux的线程如何实现，与进程的本质区别是什么？

4.    Linux能否满足硬实时的需求？

5.    进程如何睡眠等资源，此后又如何被唤醒？

6.    进程的调度延时是多少？

7.    调度器追求的吞吐率和响应延迟之间是什么关系？CPU消耗型和I/O消耗型进程的诉求？

8.    Linux怎么区分进程优先级？实时的调度策略和普通调度策略有什么区别？

9.    nice值的作用是什么？nice值低有什么优势？

10.  Linux可以被改造成硬实时吗？有什么方案？

11.  多核、多线程的情况下，Linux如何实现进程的负载均衡？

12.  这么多线程，究竟哪个线程在哪个CPU核上跑？有没有办法把某个线程固定到某个CPU跑？

13.  多核下如何实现中断、软中断的负载均衡？

14.  如何利用cgroup对进行进程分组，并调控各个group的CPU资源？

15.  CPU利用率和CPU负载之间的关系？CPU负载高一定用户体验差吗？

Linux内存管理回答如下问题：

1.    Linux系统的内存用掉了多少，还剩余多少？下面这个free命令每一个数字是什么意思？

2.    为什么要有DMA、NORMAL、HIGHMEM zone？每个zone的大小是由谁决定的？

3.    系统的内存是如何被内核和应用瓜分掉的？

4.    底层的内存管理算法buddy是怎么工作的？它和内核里面的slab分配器是什么关系？

5.    频繁的内存申请和释放是否会导致内存的碎片化？它的后果是什么？

6.    Linux内存耗尽后，系统会发生怎样的情况？

7.    应用程序的内存是什么时候拿到的？malloc()成功后，是否真的拿到了内存？应用程序的malloc()与free()与内核的关系究竟是什么？

8.    什么是lazy分配机制？应用的内存为什么会延后以最懒惰的方式拿到？

9.    我写的应用究竟耗费了多少内存？进程的vss/rss/pss/uss分别是什么概念？虚拟的，真实的，共享的，独占的，究竟哪个是哪个？

10.  内存为什么要做文件系统的缓存？如何做？缓存何时放弃？

11.  Free命令里面显示的buffers和cached分别是什么？二者有何区别？

12.  交换分区、虚拟内存究竟是什么鬼？它们针对的是什么性质的内存？什么是匿名页？

13.  进程耗费的内存、文件系统的缓存何时回收？回收的算法是不是类似LRU？

14.  怎样追踪和判决发生了内存泄漏？内存泄漏后如何查找泄漏源？

15.  内存大小这样影响系统的性能？CPU、内存、I/O三角如何互动？它们如何综合决定系统的一些关键性能？

以上问题，如果您都能回答，那么恭喜您，您是一个概念清楚的人，Linux出现吞吐低、延迟大、响应慢等问题的时候，你可以找到一个可能的方向。如果您只能回答低于1/3的问题，那么，Linux对您仍然是一片空白，出现问题，您只会陷入瞎猫子乱抓，而捞不到耗子的困境，或者胡乱地意测问题，陷入不断的低水平重试。

试图回答这些问题

本文的目的不是回答这些问题，因为回答这些问题，需要洋洋洒洒数百页的文档，而本文档不会超过10页。所以，本文的目的是试图给出一个回答这些问题的思考问题的出发点，我们倡导面对任何问题的时候，先要弄明白系统的设计目标。

**吞吐vs.响应**

首先我们在思考调度器的时候，我们要理解任何操作系统的调度器设计只追求2个目标：吞吐率大和延迟低。这2个目标有点类似零和游戏，因为吞吐率要大，势必要把更多的时间放在做**真实的有用功**，而不是把时间浪费在频繁的进程**上下文切换**；而延迟要低，势必要求优先级高的进程可以随时抢占进来，打断别人，强行插队。但是，抢占会引起上下文切换，上下文切换的时间本身对吞吐率来讲，是一个消耗，这个消耗可以低到2us或者更低（这看起来没什么？），但是上下文切换更大的消耗不是切换本身，而是切换会引起大量的cache miss。你明明weibo跑的很爽，现在切过去微信，那么CPU的cache是不太容易命中微信的。

不抢肯定响应差，抢了吞吐会下降。Linux不是一个完全照顾吞吐的系统，也不是一个完全照顾响应的系统，它作为一个软实时的操作系统，实际上是想达到某种平衡，同时也提供给用户一定的配置能力，在内核编译的时候，Kernel Features  --->  Preemption Model选项实际上可以让我们编译内核的时候，是倾向于支持吞吐，还是支持响应：

越往上面选，吞吐越好，越好下面选，响应越好。服务器你一个月也难得用一次鼠标，而桌面则显然要求一定的响应，这样可以保证UI行为的表现较好。但是Linux即便选择的是最后一个选项“Preemptible Kernel (Low-Latency Desktop)”，它仍然不是硬实时的。因为，在Linux有三类区间是不可以抢占调度的，这三类区间是：

* 中断
* 软中断
* 持有类似spin\_lock这样的锁而锁住该CPU核调度的情况

如下图，一个绿色的普通进程在T1时刻持有spin\_lock进入一个critical section（该核调度被关），绿色进程T2时刻被中断打断，而后T3时刻IRQ1里面唤醒了红色的RT进程（如果是硬实时RTOS，这个时候RT进程应该能抢入），之后IRQ1后又执行了IRQ2，到T4时刻IRQ1和IRQ2都结束了，红色RT进程仍然不能执行（因为绿色进程还在spin\_lock里面），直到T5时刻，普通进程释放spin\_lock后，红色RT进程才抢入。从T3到T5要多久，**鬼都不知道**，这样就无法满足硬实时系统的“可预期”延迟性，因此Linux不是硬实时操作系统。

Linux的preempt-rt补丁试图把中断、软中断线程化，变成可以被抢占的区间，而把会关本核调度器的spin\_lock替换为可以调度的mutex，它实现了在T3时刻唤醒RT进程的时刻，RT进程可以立即抢占调度进入的目标，避免了T3-T5之间延迟的非确定性。

**CPU消耗型 vs. I/O消耗型**

在Linux运行的进程，分为2类，一类是CPU消耗型（狂算），一类是I/O消耗型（狂睡，等I/O），前者CPU利用率高，后者CPU利用率低。一般而言，I/O消耗型任务对延迟比较敏感，应该被优先调度。比如，你正在疯狂编译安卓，而等鼠标行为的用户界面老不工作（正在狂睡），但是鼠标一点，我们应该优先打断正在编译的进程，而去响应鼠标这个I/O，这样电脑的用户体验才符合人性。

Linux的进程，对于RT进程而言，按照SCHED\_FIFO和SCHED\_RR的策略，优先级高先执行；优先级高的睡眠了后优先级的执行；同等优先级的SCHED\_FIFO先ready的跑到睡，后ready的接着跑；而同等优先级的RR则进行时间片轮转。比如Linux存在如下4个进程，T1~T4（内核里面优先级数字越低，优先级越高）：

那么它们在Linux的跑法就是：

RT的进程调度有一点“恶霸”色彩，我高优先级的没睡，低优先级的你就靠边站。但是Linux的绝大多数进程都不是RT的进程，而是采用SCHED\_NORMAL策略（这符合蜘蛛侠法则）。NORMAL的人比较善良，我们一般用nice来形容它们的优先级，nice越高，优先级越低（你越nice，就越喜欢在地铁让座，当然越坐不到座位）。普通进程的跑法，并不是nice低的一定堵着nice高的（要不然还说什么“善良”），它是按照如下公式进行：

vruntime =  pruntime \* NICE\_0\_LOAD/ weight

*其中NICE\_0\_LOAD是1024，也就是NICE是0的进程的weight*。vruntime是进程的虚拟运行时间，pruntime是物理运行时间，weight是权重，权重完全由nice决定，如下表：

在RT进程都睡过去之后（有一个特例就是RT没睡也会跑普通进程，那就是RT加起来跑地实在太久太久，普通进程必须喝点汤了），Linux开始跑NORMAL的，它倾向于调度vruntime（虚拟运行时间）最小的普通进程，根据我们小学数学知识，vruntime要小，要么分子小（喜欢睡，I/O型进程，pruntime不容易长大），要么分母大（nice值低，优先级高，权重大）。这样一个简单的公式，就同时照顾了普通进程的优先级和CPU/IO消耗情况。  
比如有4个普通进程，如下表，目前显然T1的vruntime最小(这是它喜欢睡的结果)，然后T1被调度到。

|  |  |  |  |
| --- | --- | --- | --- |
|  | pruntime | Weight | vruntime |
| T1 | 8 | 1024（nice=0） | 8\*1024/1024=8 |
| T2 | 10 | 526(nice=3) | 10\*1024/526 =19 |
| T3 | 20 | 1024（nice=0） | 20\*1024/1024=20 |
| T4 | 20 | 820(nice=1) | 20\*1024/820=24 |

然后，我们假设T1被调度再执行12个pruntime，它的vruntime将增大delta\*1024/weight（这里delta是12，weight是1024），于是T1的vruntime成为20，那么这个时候vruntime最小的反而是T2（为19），此后，Linux将倾向于调度T2（尽管T2的nice值大于T1，优先级低于T1，但是它的vruntime现在只有19）。  
所以，普通进程的调度，是一个综合考虑你喜欢干活还是喜欢睡和你的nice值是多少的结果。鉴于此，我们去问一个普通进程的调度延迟究竟有多大，这个问题，本身意义就不是特别大，它完全取决于当前的系统里面还有谁在跑，取决于你唤醒的进程的nice和它前面喜欢不喜欢睡觉。  
明白了这一点，你就不会在Linux里面问一些让回答的人吐血的问题。比如，一个普通进程多久被调度到？明确地说，不知道！装逼的说法，就是“depend on …”，依赖的东西太多。再装逼的说法，就是“一言难尽”，但这也是大实话。

**分配vs. 占据**

**Linux作为一个把应用程序员当傻逼的操作系统，它必须允许应用程序犯错**。所以这类问题就不要问了：进程malloc()了内存，还没有free()就挂了，那么我前面分配的内存没有释放，是不是就泄漏掉了？明确的说，这是不可能的，Linux内核如果这么傻，它是无法应付乱七八糟的各种开源有漏洞软件的，所以进程死的时候，肯定是资源皆被内核释放的，这类傻问题，你明白Linux的出发点，就不会再去问了。

同样的，你在应用程序里面malloc()成功的一刻，也不要以为真的拿到了内存，这个时候你的vss（虚拟地址空间，Virtual Set Size）会增大，但是你的rss(驻留在内存条上的内存，Resident SetSize)内存会随着写到每一页而缓慢增大。所以，分配成功的一刻，顶多只是被忽悠了，和你实际占有还是不占有，暂时没有半毛钱关系。

如下图，最初的堆是8KB,这8KB也写过了，所以堆的vss和rss都是8KB。此后我们调用brk()把堆变大到16KB，但是实际上它占据的内存rss还是8KB,因为第3页还没有写，根本没有真正从内存条上拿到内存。直到写第3页，堆的rss才变为12KB。这就是Linux针对app的lazy分配机制，它的出发点，当然也是防止应用程序傻逼了。

代码段的内存、堆的内存、栈的内存都是这样懒惰地拿到，demanding page。

我们有一台1GB内存的32位Linux系统，我们关闭swap，同时透过修改overcommit\_memory为1来允许申请不超过进程虚拟地址空间的内存：

$ sudo swapoff -a

$ sudo sh -c 'echo 1 >/proc/sys/vm/overcommit\_memory'

此后，我们的应用可以申请一个超级大的内存（比实际内存还大）：

上述程序在1GB的电脑上面运行，申请2GB内存可以申请成功，但是在写到一定程度后，系统出现out-of-memory，上述程序对应的进程作为oom\_score最大（最该死的）的进程被系统杀死。

**隔离vs. 共享**

Linux进程究竟耗费了多少内存，是一个非常复杂的概念，除了上面的vss, rss外，还有pss和uss，这些都是Linux不同于RTOS的显著特点之一。Linux各个进程既要做到隔离，但是隔离中又要实现共享，比如1000个进程都用libc，libc的代码段显然在内存只应该有一份。

下面的一幅图上有3个进程，pid为1044的 bash、pid为1045的 bash和pid为1054的 cat。每个进程透过自己的页表，把虚拟地址空间指向内存条上面的物理地址，每次切换一个进程，即切换一份独特的页表。

仅从此图而言，进程1044的vss和rss分别是：

vss= 1+2+3

rss= 4+5+6

但是是不是“4+5+6”就是1044这个进程耗费的内存呢？这显然也是不准确的，因为4明显被3个进程指向，5明显被2个进程指向，坏事是大家一起干的，不能1044一个人背黑锅。这个时候，就衍生出了一个pss（按比例计算的驻留内存, Proportional Set Size ）的概念，仅从这一幅图而言，进程1044的pss为：

rss= 4/3 +5/2 +6

最后，还有进程1044独占且驻留的内存uss（Unique Set Size ），仅从此图而言，

Uss = 6。

所以，分析Linux，我们不能模棱两可地停留于表面，或者想当然地说：“Linux的进程耗费了多少内存？”因为这个问题，又是一个要靠装逼来回答的问题，“dependon…”。坦白讲，每次当我问到老外问题，老外第一句话就是“depend on…”的时候，我就想上去抽他了，但是我又抑制了这个冲动，因为，很多问题，不是简单的0和1问题，正反问题，黑白问题，它确实是一个“depend on …”的问题。

有时候，小白问大拿一个问题，大拿实在是无法正面回答，于是就支支吾吾一番。这个时候小白会很生气，觉得大拿态度不好，或者在装逼。你实际上，明白很多问题不是简单的0与1问题之后，你就会理解，他真的不是在装逼。**这个时候，我们要反过来检讨自己，是不是我们自己问的问题太LOW逼了？**

思考大于接受

我们前面提出了30个问题，而本文也仅仅只是回答了其中极少的一部分。此文的目的在于建立思维，导入方向，而不是洋洋洒洒地把所有问题回答掉，因为哥确实没有时间写个几百页的文档来一一回答这些问题。很多事情，用口头描述，比直接写冗长地文档要更加容易也轻松。

最后，我仍然想要强调的一个观点是，我们在思维Linux的时候，更多地可以把自己想象成Linus Torvalds，如果你是Linus Torvalds，你要设计Linux，你碰到某个诉求，比如调度器和内存方面的诉求，你应该如何解决。我们不是被动地接受“是什么”，更多地要思考“为什么”，“怎么办”。

如果你是Linus Torvalds，有个傻逼应用程序员要申请1GB内存，你是直接给他，还是假装给他，但是实际没有给他，直到它写的时候再给他？

如果你是Linus Torvalds，有个家伙打开了串口，然后进程就做个1/0运算或者访问空指针挂了，你要不要在这个进程挂的时候给它关闭串口？

如果你是Linus Torvalds，你是要让nice值低（优先级高）的普通进程在睡眠前一直堵着nice值高的进程，还是虽然它优先级高，但是由于跑的时间比较长后，也要让给优先级低(nice值高)的进程？如果你认为nice值低的应该一直跑，那么如何照顾喜欢睡觉的I/O消耗型进程？万一nice值低的进程有bug，进入死循环，那么nice高的进程岂不是丝毫机会都没有？这样的设计，是不是反人类？

…

当你带着这些思考，武装这些concept，再去看Linux的时候，你就从被动的“接受”，变成了主动地“思考”，这正好是任何一个优秀程序员都具备的品质，也是打通进程调度和内存管理任督二脉的关键。

原来便在这顷刻之间，张无忌所练的九阳神功已然大功告成，水火相济，龙虎交会。要知布袋内真气充沛，等于是数十位高手各出真力，同时按摩挤逼他周身数百处穴道，他内内外外的真气激荡，身上数十处玄关一一冲破，只觉全身脉络之中，有如一条条水银在到处流转，舒适无比。

——金庸 《倚天屠龙记》

关于进程的直播

CSDN学院联合笔者的这次关于进程的课程分成4个组成部分，每次课60分钟。每次课后留下3~4个练习题，可以在微信群或者Linuxer公众号留言讨论答案和做题心得。

第一部分深入彻底地搞清楚进程生命周期，进程生命周期创建、退出、停止，以及僵尸是个什么意思；第二部分，深入分析进程创建的写时拷贝技术、以及Linux的线程究竟是怎么回事（为什么称为轻量级进程），此部分也会搞清楚进程0、进程1和托孤，以及睡眠时的等待队列；第三部分，搞清楚Linux进程调度算法，不同的调度策略、实时性，完全公平调度算法；第四部分，搞清楚Linux多核下的CPU、中断、软负载均衡，cgroups调度算法以及Linux为什么不是一个实时操作系统，如何把Linux变成一个硬实时的操作系统。

通过这4部分的学习，彻底理清Linux的进程、线程，弄清楚你写的内核和应用程序在系统里面究竟是如何跑，知其然，知其所以然。

第一部分大纲

1. Linux进程生命周期(就绪、运行、睡眠、停止、僵死)
2. 僵尸是个什么鬼？
3. 停止状态与作业控制，cpulimit
4. 内存泄漏的真实含义
5. task\_struct以及task\_struct之间的关系
6. 初见fork和僵尸

**练习题**

1. fork的例子
2. life-period例子，观察僵尸
3. 用cpulimit控制CPU利用率

第二部分大纲

1. fork、vfork、clone
2. 写时拷贝技术
3. Linux线程的实现本质
4. 进程0和进程1
5. 进程的睡眠和等待队列
6. 孤儿进程的托孤，SUBREAPER

**练习题**

1. fork、vfork、Copy-on-Write例子
2. life-period例子，实验体会托孤
3. pthread\_create例子，strace它
4. 彻底看懂等待队列的案例

第三部分大纲

1. CPU/IO消耗型进程
2. 吞吐率 vs. 响应
3. SCHED\_FIFO、SCHED\_RR
4. SCHED\_NORMAL和CFS
5. nice、renice
6. chrt

**练习题**

1. 运行2个高CPU利用率程序，调整他们的nice
2. 用chrt把一个死循环程序调整为SCHED\_FIFO
3. 阅读ARM的big.LITTLE架构资料，并论述为什么ARM要这么做？

第四部分大纲

1. 多核下负载均衡
2. 中断负载均衡、RPS软中断负载均衡
3. cgroups和CPU资源分群分配
4. Android和NEON对cgroups的采用
5. Linux为什么不是硬实时的
6. preempt-rt对Linux实时性的改造

**练习题**

1. 用time命令跑1个含有2个死循环线程的进程
2. 用taskset调整多线程依附的CPU
3. 创建和分群CPU的cgroup，调整权重和quota
4. cyclictest

报名方法

本课程共有四次直播，直播已经全部结束，录播报名链接：

<http://edu.csdn.net/course/detail/5995>

高清PDF下载：

四次课程的ppt微信版本：

[《Linux进程、线程和调度》系列9.13日第一节课ppt分享](http://mp.weixin.qq.com/s?__biz=MzAwMDUwNDgxOA==&mid=2652662308&idx=1&sn=8075c6248078f7841788d909e535a9c6&chksm=810f2cb9b678a5af8af02be9aecc8ed40650af658e4f2a09d9ad3e0170c4d73d2e2ae2d6db30&scene=21#wechat_redirect)

[《Linux进程、线程和调度》系列9.15日第二节课ppt分享](http://mp.weixin.qq.com/s?__biz=MzAwMDUwNDgxOA==&mid=2652662338&idx=1&sn=180a950d678893b0b316825d45f2c1a3&chksm=810f2cdfb678a5c96a2882a10f354ed5fe792de03343f1be5b9a017ef3a6181cf761e3eefe97&scene=21#wechat_redirect)

[《Linux进程、线程和调度》系列9.19日第三节课ppt分享](http://mp.weixin.qq.com/s?__biz=MzAwMDUwNDgxOA==&mid=2652662375&idx=1&sn=55099114de9135d471dbe18c10250317&chksm=810f2cfab678a5ecb9081b7311f2f6f5f7472e389e3343ae4ad538bfe5c645876f18351e558e&scene=21#wechat_redirect)

[《Linux进程、线程和调度》系列9.22日第四节课ppt分享](http://mp.weixin.qq.com/s?__biz=MzAwMDUwNDgxOA==&mid=2652662397&idx=1&sn=d81c2eac314ac6f6a58a109ab83370e9&chksm=810f2ce0b678a5f6ca254978b50c79e29bef068eb245a460d63fe980e63754c8f449486333f5&scene=21#wechat_redirect)

高清PDF版本下载：

<https://pan.baidu.com/s/1hsMSeMw>