笔记本： Linux
创建时间： 2019/1/8 14:31 更新时间： 2019/1/16 9:44
标签： 性能优化
URL： https://time.geekbang.org/column/article/68728

开篇词 | 别再让Linux性能问题成为你的绊脚石

你好，我是倪朋飞，微软 Azure 的资深工程师，同时也是 Kubernetes 项目维护者，主要负责开源容器编排系统 Kubernetes 在 Azure 的落地实践。

一直以来，我都在云计算领域工作。对于服务器性能的关注，可以追溯到我刚参加工作那会儿。为什么那么早就开始探索性能问题呢？其实是源于一次我永远都忘不了的“事故”。

那会儿我在盛大云工作，忙活了大半夜把产品发布上线后，刚刚躺下打算休息，却突然收到大量的告警。匆忙爬起来登录到服务器之后，我发现有一些系统进程的 CPU 使用率高达 100%。

当时我完全是两眼一抹黑，可以说是只能看到症状，却完全不知道该从哪儿下手去排查和解决它。直到最后，我也没能想到好办法，这次发布也成了我心中之痛。

从那之后，我开始到处查看各种相关书籍，从操作系统原理、到 Linux 内核，再到硬件驱动程序等等。可是，学了那么多知识之后，我还是不能很快解决类似的性能问题。

于是，我又通过网络搜索，或者请教公司的技术大拿，学习了大量性能优化的思路和方法，这期间尝试了大量的 Linux 性能工具。在不断的实践和总结后，我终于知道，怎么**把观察到的性能问题跟系统原理关联起来，特别是把系统从应用程序、库函数、系统调用、再到内核和硬件等不同的层级贯穿起来。**

这段学习可以算得上是我的“黑暗”经历了。我想，不仅是我一个人，很多人应该都有过这样的挫折。比如说：

- 流量高峰期，服务器 CPU 使用率过高报警，你登录 Linux 上去 top 完之后，却不知道怎么进一步定位，到底是系统 CPU 资源太少，还是程序并发部分写的有问题？
- 系统并没有跑什么吃内存的程序，但是敲完 free 命令之后，却发现系统已经没有什么内存了，那到底是哪里占用了内存？为什么？
- 一大早就收到 Zabbix 告警，你发现某台存放监控数据的数据库主机的 iowait 较高，这个时候该怎么办？

这些问题或者场景，你肯定或多或少都遇到过。

实际上，**性能优化一直都是大多数软件工程师头上的“紧箍咒”**，甚至许多工作多年的资深工程师，也无法准确地分析出线上的很多性能问题。

性能问题为什么这么难呢？我觉得主要是因为性能优化是个系统工程，总是牵一发而动全身。它涉及了从程序设计、算法分析、编程语言，再到系统、存储、网络等各种底层基础设施的方方面面。每一个组件都有可能出问题，而且很有可能多个组件同时出问题。

毫无疑问，性能优化是软件系统中最有挑战的工作之一，但是换个角度看，**它也是最考验体现你综合能力的工作之一**。如果说你能把性能优化的各个关键点吃透，那我可以肯定地说，你已经是一个非常优秀的软件工程师了。

那怎样才能掌握这个技能呢？你可以像我前面说的那样，花大量的时间和精力去钻研，从内功到实战一一苦练。当然，那样可行，但也会走很多弯路，而且可能你啃了很多大块头的书，终于拿下了最难的底层体系，却因为缺乏实战经验，在实际开发工作中仍然没有头绪。

其实，对于我们大多数人来说，**最好的学习方式一定是带着问题学习**，而不是先去啃那几本厚厚的原理书籍，这样很容易把自己的信心压垮。

我认为，**学习要会抓重点**。其实只要你了解少数几个系统组件的基本原理和协作方式，掌握基本的性能指标和工具，学会实际工作中性能优化的常用技巧，你就已经可以准确分析和优化大多数的性能问题了。在这个认知的基础上，再反过来去阅读那些经典的操作系统或者其它图书，你才能事半功倍。

所以，在这个专栏里，我会以**案例驱动**的思路，给你讲解 Linux 性能的基本指标、工具，以及相应的观测、分析和调优方法。

具体来看，我会分为 5 个模块。前 4 个模块我会从资源使用的视角出发，带你分析各种 Linux 资源可能会碰到的性能问题，包括 **CPU 性能**、**磁盘 I/O 性能**、**内存性能**以及**网络性能**。每个模块还由浅入深划分为四个不同的篇章。

- **基础篇**，介绍 Linux 必备的基本原理以及对应的性能指标和性能工具。比如怎么理解平均负载，怎么理解上下文切换，Linux 内存的工作原理等等。
- **案例篇**，这里我会通过模拟案例，帮你分析高手在遇到资源瓶颈时，是如何观测、定位、分析并优化这些性能问题的。
- **套路篇**，在理解了基础，亲身体验了模拟案例之后，我会帮你梳理出排查问题的整体思路，也就是检查性能问题的一般步骤，这样，以后你遇到问题，就可以按照这样的路子来。

- **答疑篇**，我相信在学习完每一个模块之后，你都会有很多的问题，在答疑篇里，我会拿出提问频次较高的问题给你系统解答。

第 5 个综合实战模块，我将为你还原真实的工作场景，手把手带你在“**高级战场**”中演练，这样你能把前面学到的所有知识融会贯通，并且看完专栏，马上就能用在工作中。

整个专栏，我会把内容尽量写得通俗易懂，并帮你划出重点、理出知识脉络，再通过案例分析和套路总结，让你学得更透、用得更熟。

明天就要正式开课了，开始之前，我要把何炅说过的那句我特别认同的鸡汤送给你，“**想要得到你就要学会付出，要付出还要坚持；如果你真的觉得很难，那你就放弃，如果你放弃了就不要抱怨。人生就是这样，世界是平衡的，每个人都是通过自己的努力，去决定自己生活的样子。**”

不为别的，就希望你能和我坚持下去，一直到最后一篇文章。这中间，有想不明白的地方，你要先自己多琢磨几次；还是不懂的，你可以在留言区找我问；有需要总结提炼的知识点，你也要自己多下笔。你还可以写下自己的经历，记录你的分析步骤和思路，我都会及时回复你。

最后，你可以在留言区给自己立个 Flag，**哪怕只是在留言区打卡你的学习天数，我相信都是会有效果的**。3 个月后，我们一起再来验收。

总之，让我们一起携手，为你交付“Linux 性能优化”这个大技能！

笔记本： Linux
创建时间： 2019/1/7 16:28 更新时间： 2019/1/16 9:44
标签： 性能优化
URL： https://time.geekbang.org/column/article/69346

01 | 如何学习Linux性能优化?

你是否也曾跟我一样，看了很多书、学了很多 Linux 性能工具，但在面对 Linux 性能问题时，还是束手无策？实际上，性能分析和优化始终是大多数软件工程师的一个痛点。但是，面对难题，我们真的就无解了吗?

固然，性能问题的复杂性增加了学习难度，但这并不能成为我们进阶路上的“拦路虎”。在我看来，大多数人对性能问题“投降”，原因可能只有两个。

一个是你没找到有效的方法学原理，一听到“系统”、“底层”这些词就发怵，觉得东西太难，自己一定学不会，自然也就无法深入学下去，从而不能建立起性能的全局观。

再一个就是，你看到性能问题的根源太复杂，既不懂怎么去分析，也不能抽丝剥茧找到瓶颈。

你可能会想，反正程序出了问题，上网查就是了，用别人的方法，囫囵吞枣地多试几次，有可能就解决了。于是，你懒得深究这些方法为啥有效，更不知道为什么，很多方法在别人的环境有效，到你这儿就不行了。

所以，相同的错误重复在犯，相同的状况也是重复出现。

其实，性能问题并没有你想像得那么难，**只要你理解了应用程序和系统的少数几个基本原理，再进行大量的实战练习，建立起整体性能的全局观**，大多数性能问题的优化就会水到渠成。

我见过很多工程师，在分析应用程序所使用的第三方组件的性能时，并不熟悉这些组件所用的编程语言，却依然可以分析出线上问题的根源，并能通过一些方法进行优化，比如修改应用程序对它们的调用逻辑，或者调整组件的配置选项等。

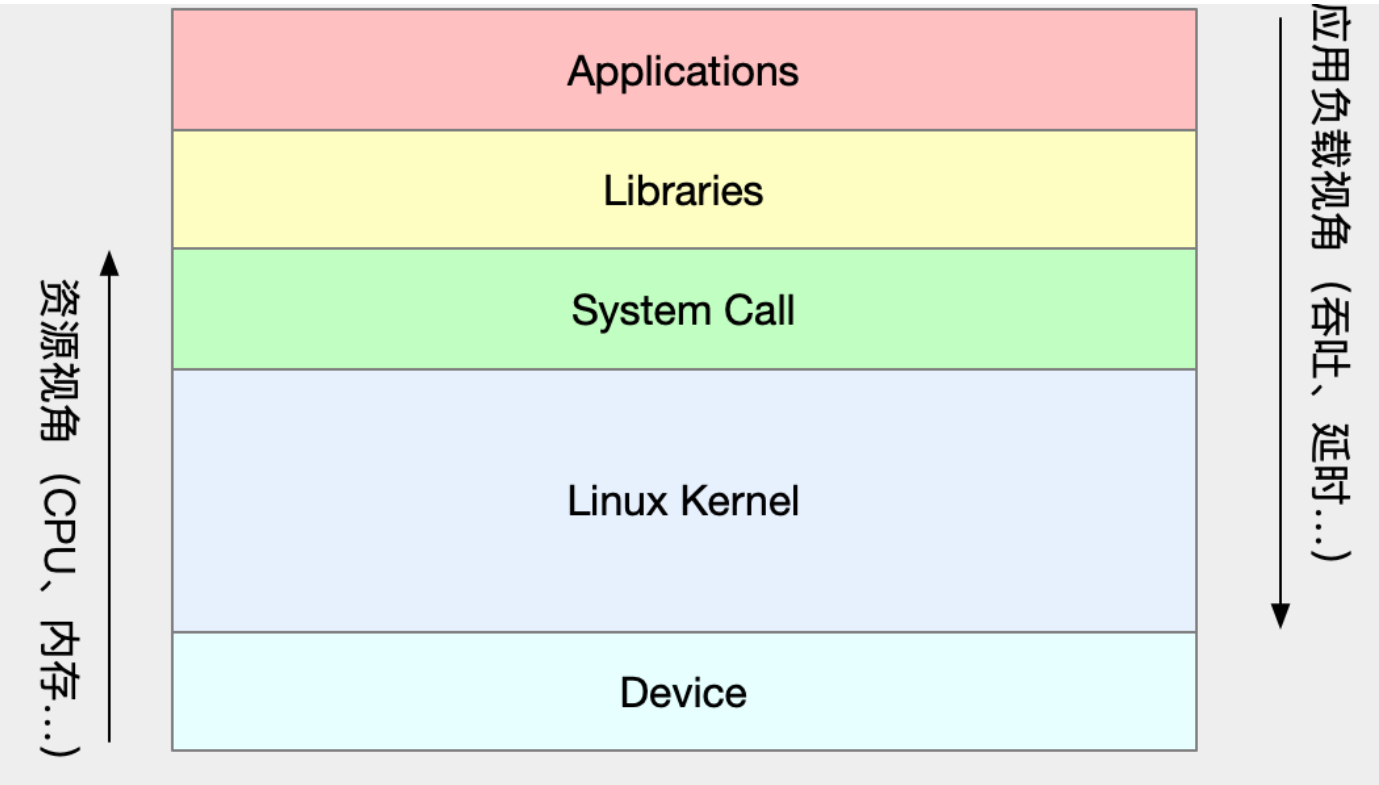
还是那句话，**你不需要了解每个组件的所有实现细节**，只要能理解它们最基本的工作原理和协作方式，你也可以做到。

性能指标是什么?

学习性能优化的第一步，一定是了解“性能指标”这个概念。

当看到性能指标时，你会首先想到什么呢？我相信“**高并发**”和“**响应快**”一定是最先出现在你脑海里的两个词，而它们也正对应着性能优化的两个核心指标——“吞吐”和“延时”。这两个指

标是**从应用负载的视角**来考察性能，直接影响了产品终端的用户体验。跟它们对应的，是**从系统资源的视角**出发的指标，比如资源使用率、饱和度等。



我们知道，随着应用负载的增加，系统资源的使用也会升高，甚至达到极限。而**性能问题的本质**，就是系统资源已经达到瓶颈，但请求的处理却还不够快，无法支撑更多的请求。

性能分析，其实就是**找出应用或系统的瓶颈，并设法去避免或者缓解它们**，从而更高效地利用系统资源处理更多的请求。这包含了一系列的步骤，比如下面这六个步骤。

- 选择指标评估应用程序和系统的性能；
- 为应用程序和系统设置性能目标；
- 进行性能基准测试；
- 性能分析定位瓶颈；
- 优化系统和应用程序；
- 性能监控和告警。

了解了这些性能相关的基本指标和核心步骤后，该怎么学呢？接下来，我来说说要学好 Linux 性能优化的几个重要问题。

学这个专栏需要什么基础

首先你要明白，我们这个专栏的核心是性能的分析和优化，而不是最基本的 Linux 操作系统的使用方法。

因而，我希望你最好用过 Ubuntu 或其他 Linux 操作系统，然后要具备一些**编程基础**，比如：

- 了解 Linux 常用命令的使用方法；
- 知道怎么安装和管理软件包；
- 知道怎么通过编程语言开发应用程序等。

这样，在我讲性能时，你就更容易理解性能背后的原理，特别是在结合专栏里的案例实践后，对性能分析能有更直观的体会。

这个专栏不会像教科书那样，详细教你操作系统、算法原理、网络协议乃至各种编程语言的全部细节，但一些重要的系统原理还是必不可少的。我还会用实际案例一步步教你，贯穿从应用程序到操作系统的各个组件。

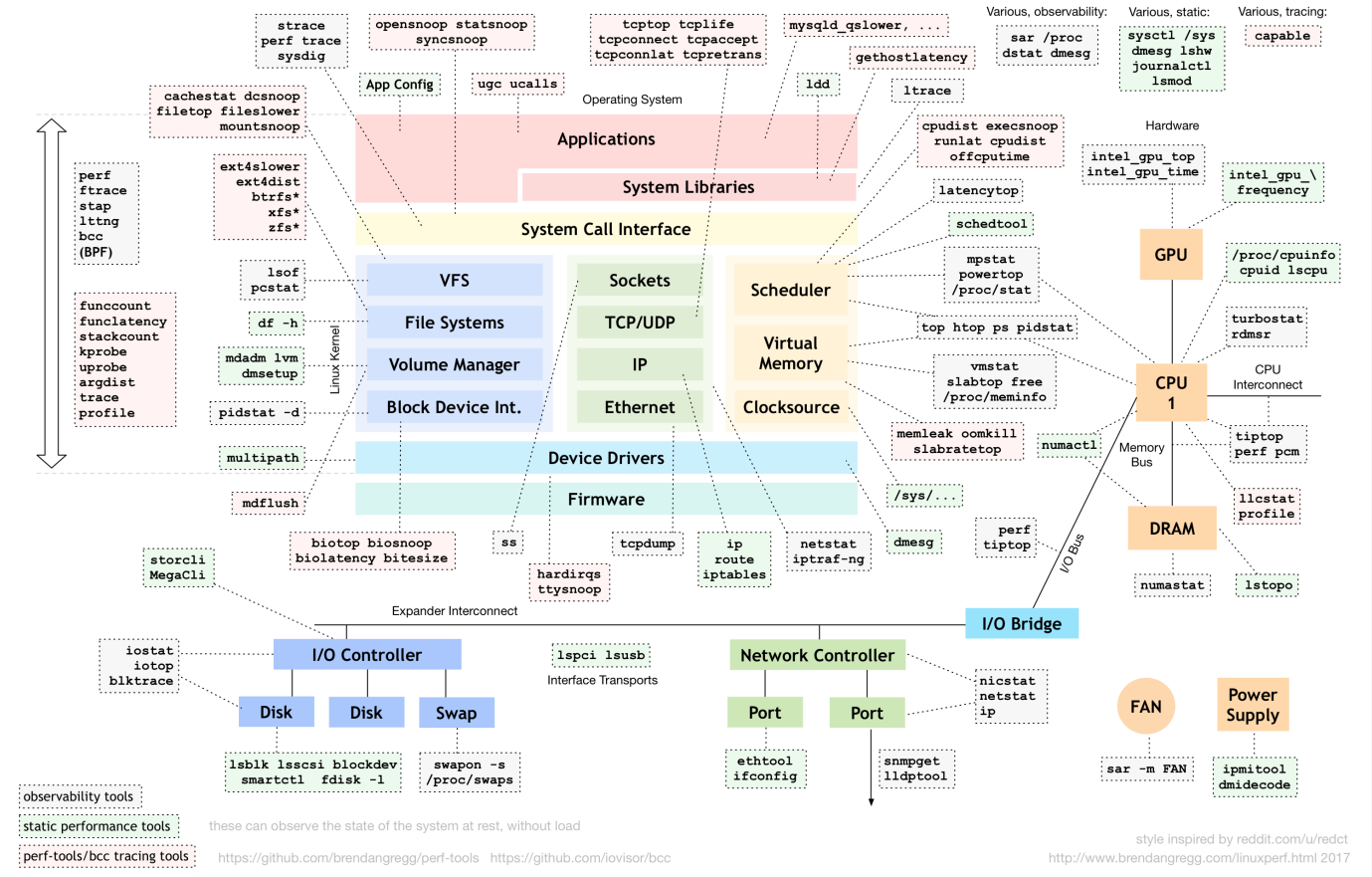
学习的重点是什么？

想要学习好性能分析和优化，**建立整体系统性能的全局观**是最核心的话题。因而，

- 理解最基本的几个系统知识原理；
- 掌握必要的性能工具；
- 通过实际的场景演练，贯穿不同的组件。

这三点，就是我们学习的重中之重。我会在专栏的每篇文章中，针对不同场景，把这三个方面给你讲清楚，你也一定要花时间和心思来消化它们。

其实说到性能工具，就不得不提性能领域的大师布伦丹·格雷格（Brendan Gregg）。他不仅是动态追踪工具 DTrace 的作者，还开发了许许多多的性能工具。我相信你一定见过他所描绘的 Linux 性能工具图谱：



（图片来自brendangregg.com）

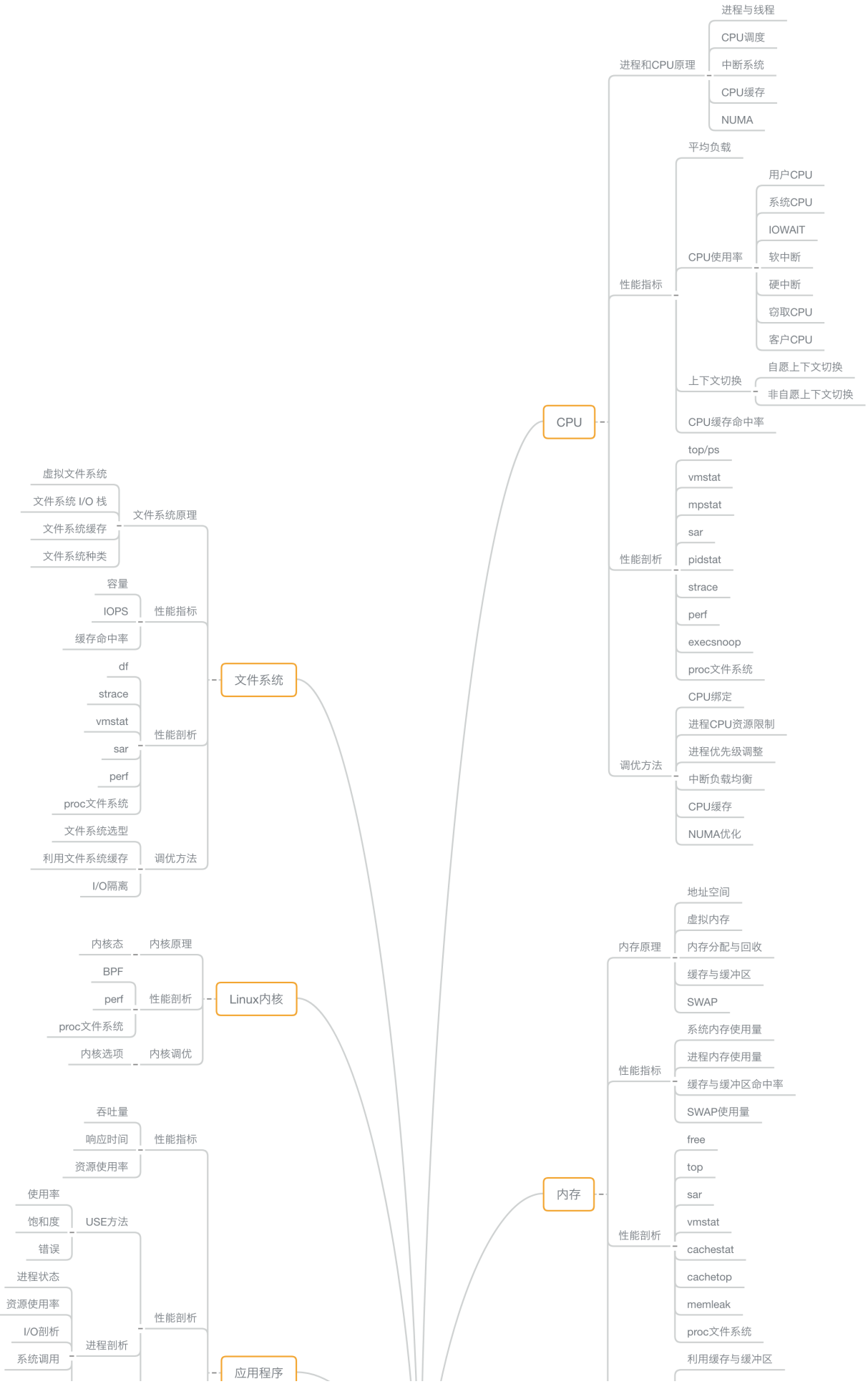
这个图是 Linux 性能分析最重要的参考资料之一，它告诉你，在 Linux 不同子系统出现性能问题后，应该用什么样的工具来观测和分析。

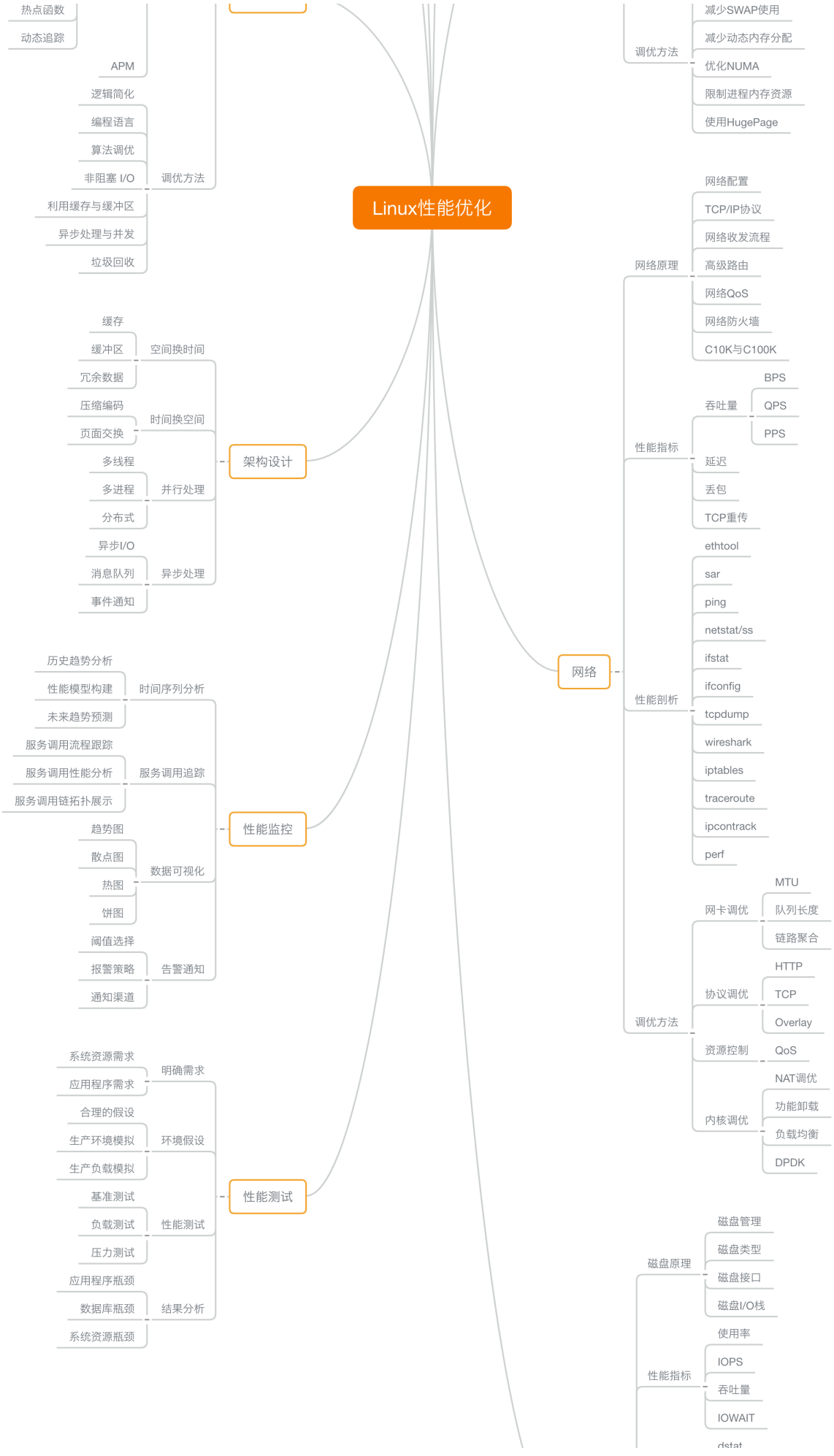
比如，当遇到 I/O 性能问题时，可以参考图片最下方的 I/O 子系统，使用 iostat、iotop、blktrace 等工具分析磁盘 I/O 的瓶颈。你可以把这个图保存下来，在需要的时候参考查询。

另外，我还要特别强调一点，就是**性能工具的选用**。有句话是这么说的，一个正确的选择胜过千百次的努力。虽然夸张了些，但是选用合适的性能工具，确实可以大大简化整个性能优化过程。在什么场景选用什么样的工具、以及怎么学会选择合适工具，都是我想教给你的东西。

但是切记，**千万不要把性能工具当成学习的全部**。工具只是解决问题的手段，关键在于你的用法。只有真正理解了它们背后的原理，并且结合具体场景，融会贯通系统的不同组件，你才能真正掌握它们。

最后，为了让你对性能有个全面的认识，我画了一张思维导图，里面涵盖了大部分性能分析和优化都会包含的知识，专栏中也基本都会讲到。你可以保存或者打印下来，每学会一部分就标记出来，记录并把握自己的学习进度。


CPU
进程和CPU原理
进程与线程
CPU调度
中断系统
CPU缓存
NUMA
性能指标
平均负载
CPU使用率
用户CPU
系统CPU
IOWAIT
软中断
硬中断
窃取CPU
客户CPU
上下文切换
自愿上下文切换
非自愿上下文切换
CPU缓存命中率
性能剖析
top/ps
vmstat
mpstat
sar
pidstat
strace
perf
execsnoop
proc文件系统
调优方法
CPU绑定
进程CPU资源限制
进程优先级调整
中断负载均衡
CPU缓存
NUMA优化
内存
内存原理
地址空间
虚拟内存
内存分配与回收
缓存与缓冲区
SWAP
性能指标
系统内存使用量
进程内存使用量
缓存与缓冲区命中率
SWAP使用量
性能剖析
free
top
sar
vmstat
cachestat
cachetop
memleak
proc文件系统
利用缓存与缓冲区
文件系统
文件系统原理
虚拟文件系统
文件系统 I/O 栈
文件系统缓存
文件系统种类
性能指标
容量
IOPS
缓存命中率
性能剖析
df
strace
vmstat
sar
perf
proc文件系统
调优方法
文件系统选型
利用文件系统缓存
I/O隔离
Linux内核
内核原理
内核态
性能剖析
BPF
perf
proc文件系统
内核调优
内核选项
应用程序
性能指标
吞吐量
响应时间
资源使用率
性能剖析
USE方法
使用率
饱和度
错误
进程剖析
进程状态
资源使用率
I/O剖析
系统调用


Linux性能优化
架构设计
调优方法
热点函数
动态追踪
APM
逻辑简化
编程语言
算法调优
非阻塞 I/O
利用缓存与缓冲区
异步处理与并发
垃圾回收
空间换时间
缓存
缓冲区
冗余数据
时间换空间
压缩编码
页面交换
并行处理
多线程
多进程
分布式
异步处理
异步I/O
消息队列
事件通知
性能监控
时间序列分析
历史趋势分析
性能模型构建
未来趋势预测
服务调用追踪
服务调用流程跟踪
服务调用性能分析
服务调用链拓扑展示
数据可视化
趋势图
散点图
热图
饼图
告警通知
阈值选择
报警策略
通知渠道
性能测试
明确需求
系统资源需求
应用程序需求
环境假设
合理的假设
生产环境模拟
生产负载模拟
性能测试
基准测试
负载测试
压力测试
结果分析
应用程序瓶颈
数据库瓶颈
系统资源瓶颈
网络
网络原理
网络配置
TCP/IP协议
网络收发流程
高级路由
网络QoS
网络防火墙
C10K与C100K
性能指标
吞吐量
BPS
QPS
PPS
延迟
丢包
TCP重传
性能剖析
ethtool
sar
ping
netstat/ss
ifstat
ifconfig
tcpdump
wireshark
iptables
traceroute
ipcontrack
perf
调优方法
网卡调优
MTU
队列长度
链路聚合
协议调优
HTTP
TCP
Overlay
资源控制
QoS
内核调优
NAT调优
功能卸载
负载均衡
DPDK
减少SWAP使用
减少动态内存分配
优化NUMA
限制进程内存资源
使用HugePage
磁盘
磁盘原理
磁盘管理
磁盘类型
磁盘接口
磁盘I/O栈
性能指标
使用率
IOPS
吞吐量
IOWAIT
dstat

怎么学更高效?

前面我给你讲了 Linux 性能优化的学习重点，接下来我再跟你分享一下，我的几个学习技巧。掌握这些技巧，可以让你学得更轻松。

技巧一：虽然系统的原理很重要，但在刚开始一定不要试图抓住所有的实现细节。

深陷到系统实现的内部，可能会让你丢掉学习的重点，而且繁杂的实现逻辑，很可能会打退你学习的积极性。所以，我个人观点是一定要适度。

你可以先学会我给你讲的这些系统工作原理，但不要去深究 Linux 内核是如何做到的，而是要把你的重点放到如何观察和运用这些原理上，比如：

- 有哪些指标可以衡量性能?
- 使用什么样的性能工具来观察指标?
- 导致这些指标变化的因素等。

技巧二：边学边实践，通过大量的案例演习掌握 Linux 性能的分析和优化。

只有通过在机器上练习，把我讲的知识和案例自己过一遍，这些东西才能转化成你的。我精心设计这些案例，正是为了让你有更好的学习理解和操作体验。

所以我强烈推荐你去实际运行、分析这些案例，或者用学到的知识去分析你自己的系统，这样你会有更直观的感受，获得更好的学习效果。

技巧三：勤思考，多反思，善总结，多问为什么。

想真正学懂一门知识，最好的方法就是问问题。当你能提出好的问题时，就说明你已经深入了解了它。

你可以随时在留言区给我留言，写下自己的疑问、思考和总结，和我还有其他的学习者一起讨论切磋。你也可以写下自己经历过的性能问题，记录你的分析步骤和优化思路，我们一起互动探讨。

学习之前，你的准备

作为一个包含大量案例实践的课程，我会在每篇文章中，使用一到两台 Ubuntu 18.04 虚拟机，作为案例运行和分析的环境。如果你只是单纯听音频的讲解，却从不动手实践，学习的效果一定会大打折扣。

所以，你是不是可以准备好一台 Linux 机器，用于课程案例的实践呢？任意的虚拟机或物理机都可以，并不局限于 Ubuntu 系统。

思考

今天的内容是我们后续学习的热身准备。从下篇文章开始，我们就要正式进入 Linux 性能分析和优化了。所以，我想请你来聊一聊，你之前在解决 Linux 性能问题时，有遇到过什么样的困难或者疑惑吗？或者是之前自己学习 Linux 性能优化时，有哪些问题吗？参考我今天所讲的内容，你又打算怎么来学这个专栏？

欢迎在留言区和我分享。

笔记本： Linux
创建时间： 2019/1/8 14:32 更新时间： 2019/1/16 9:43
标签： 性能优化
URL： https://time.geekbang.org/column/article/69618

02 | 基础篇：到底应该怎么理解“平均负载”？

每次发现系统变慢时，我们通常做的第一件事，就是执行 top 或者 uptime 命令，来了解系统的负载情况。比如像下面这样，我在命令行里输入了 uptime 命令，系统也随即给出了结果。

复制代码

```
$ uptime
02:34:03 up 2 days, 20:14, 1 user, load average: 0.63, 0.83, 0.88
```

但我想问的是，你真的知道这里每列输出的含义吗？

我相信你对前面的几列比较熟悉，它们分别是当前时间、系统运行时间以及正在登录用户数。

复制代码

```
02:34:03 // 当前时间
up 2 days, 20:14 // 系统运行时间
1 user // 正在登录用户数
```

而最后三个数字呢，依次则是过去 1 分钟、5 分钟、15 分钟的平均负载（Load Average）。

平均负载？这个词对很多人来说，可能既熟悉又陌生，我们每天的工作中，也都会提到这个词，但你真正理解它背后的含义吗？如果你们团队来了一个实习生，他揪住你不放，你能给他讲清楚什么是平均负载吗？

其实，6 年前，我就遇到过这样的一个场景。公司一个实习生一直追问我，什么是平均负载，我支支吾吾半天，最后也没能解释明白。明明总看到也总会用到，怎么就说不明白呢？后来我静下来想想，其实还是自己的功底不够。

于是，这几年，我遇到问题，特别是基础问题，都会多问自己几个“为什么”，以求能够彻底理解现象背后的本质原理，用起来更灵活，也更有底气。

今天，我就带你来学习下，如何观测和理解这个最常见、也是最重要的系统指标。

我猜一定有人会说，平均负载不就是单位时间内的 CPU 使用率吗？上面的 0.63，就代表 CPU 使用率是 63%。其实并不是这样，如果你方便的话，可以通过执行 man uptime 命令，来了解平均负载的详细解释。

简单来说，平均负载是指单位时间内，系统处于**可运行状态**和**不可中断状态**的平均进程数，也就是**平均活跃进程数**，它和 CPU 使用率并没有直接关系。这里我先解释下，可运行状态和不可中断状态这俩词儿。

所谓可运行状态的进程，是指正在使用 CPU 或者正在等待 CPU 的进程，也就是我们常用 ps 命令看到的，处于 R 状态（Running 或 Runnable）的进程。

不可中断状态的进程则是正处于内核态关键流程中的进程，并且这些流程是不可打断的，比如最常见的是等待硬件设备的 I/O 响应，也就是我们在 ps 命令中看到的 D 状态（Uninterruptible Sleep，也称为 Disk Sleep）的进程。

比如，当一个进程向磁盘读写数据时，为了保证数据的一致性，在得到磁盘回复前，它是不能被其他进程或者中断打断的，这个时候的进程就处于不可中断状态。如果此时的进程被打断了，就容易出现磁盘数据与进程数据不一致的问题。

所以，不可中断状态实际上是系统对进程和硬件设备的一种保护机制。

因此，你可以简单理解为，平均负载其实就是平均活跃进程数。平均活跃进程数，直观上的理解就是单位时间内的活跃进程数，但它实际上是活跃进程数的指数衰减平均值。这个“指数衰减平均”的详细含义你不用计较，这只是系统的一种更快速的计算方式，你把它直接当成活跃进程数的平均值也没问题。

既然平均的是活跃进程数，那么最理想的，就是每个 CPU 上都刚好运行着一个进程，这样每个 CPU 都得到了充分利用。比如当平均负载为 2 时，意味着什么呢？

- 在只有 2 个 CPU 的系统上，意味着所有的 CPU 都刚好被完全占用。
- 在 4 个 CPU 的系统上，意味着 CPU 有 50% 的空闲。
- 而在只有 1 个 CPU 的系统中，则意味着有一半的进程竞争不到 CPU。

平均负载为多少时合理

讲完了什么是平均负载，现在我们再回到最开始的例子，不知道你能否判断出，在 uptime 命令的结果里，那三个时间段的平均负载数，多大的时候能说明系统负载高？或是多小的时候就能说明系统负载很低呢？

我们知道，平均负载最理想的情况是等于 CPU 个数。所以在评判平均负载时，**首先你要知道系统有几个 CPU**，这可以通过 top 命令或者从文件 /proc/cpuinfo 中读取，比如：

```
# 关于 grep 和 wc 的用法请查询它们的手册或者网络搜索
$ grep 'model name' /proc/cpuinfo | wc -l
2
```

有了 CPU 个数，我们就可以判断出，当平均负载比 CPU 个数还大的时候，系统已经出现了过载。

不过，且慢，新的问题又来了。我们在例子中可以看到，平均负载有三个数值，到底该参考哪一个呢？

实际上，都要看。三个不同时间间隔的平均值，其实给我们提供了，分析**系统负载趋势**的数据来源，让我们能更全面、更立体地理解目前的负载状况。

打个比方，就像初秋时北京的天气，如果只看中午的温度，你可能以为还在 7 月份的大夏天呢。但如果你结合了早上、中午、晚上三个时间点的温度来看，基本就可以全方位了解这一天的天气情况了。

同样的，前面说到的 CPU 的三个负载时间段也是这个道理。

- 如果 1 分钟、5 分钟、15 分钟的三个值基本相同，或者相差不大，那就说明系统负载很平稳。
- 但如果 1 分钟的值远小于 15 分钟的值，就说明系统最近 1 分钟的负载在减少，而过去 15 分钟内却有很大的负载。
- 反过来，如果 1 分钟的值远大于 15 分钟的值，就说明最近 1 分钟的负载在增加，这种增加有可能只是临时性的，也有可能还会持续增加下去，所以就需要持续观察。一旦 1 分钟的平均负载接近或超过了 CPU 的个数，就意味着系统正在发生过载的问题，这时就得分析调查是哪里导致的问题，并要想办法优化了。

这里我再举个例子，假设我们在一个单 CPU 系统上看到平均负载为 1.73，0.60，7.98，那么说明在过去 1 分钟内，系统有 73% 的超载，而在 15 分钟内，有 698% 的超载，从整体趋势来看，系统的负载在降低。

那么，在实际生产环境中，平均负载多高时，需要我们重点关注呢？

在我看来，**当平均负载高于 CPU 数量 70% 的时候**，你就应该分析排查负载高的问题了。一旦负载过高，就可能导致进程响应变慢，进而影响服务的正常功能。

但 70% 这个数字并不是绝对的，最推荐的方法，还是把系统的平均负载监控起来，然后根据更多的历史数据，判断负载的变化趋势。当发现负载有明显升高趋势时，比如说负载翻倍了，你再去做分析和调查。

平均负载与 CPU 使用率

现实工作中，我们经常容易把平均负载和 CPU 使用率混淆，所以在这里，我也做一个区分。

可能你会疑惑，既然平均负载代表的是活跃进程数，那平均负载高了，不就意味着 CPU 使用率高吗?

我们还是要回到平均负载的含义上来，平均负载是指单位时间内，处于可运行状态和不可中断状态的进程数。所以，它不仅包括了**正在使用 CPU** 的进程，还包括**等待 CPU** 和**等待 I/O** 的进程。

而 CPU 使用率，是单位时间内 CPU 繁忙情况的统计，跟平均负载并不一定完全对应。比如：

- CPU 密集型进程，使用大量 CPU 会导致平均负载升高，此时这两者是一致的；
- I/O 密集型进程，等待 I/O 也会导致平均负载升高，但 CPU 使用率不一定很高；
- 大量等待 CPU 的进程调度也会导致平均负载升高，此时的 CPU 使用率也会比较高。

平均负载案例分析

下面，我们以三个示例分别来看这三种情况，并用 iostat、mpstat、pidstat 等工具，找出平均负载升高的根源。

因为案例分析都是基于机器上的操作，所以不要只是听听、看看就够了，最好还是跟着我实际操作一下。

你的准备

下面的案例都是基于 Ubuntu 18.04，当然，同样适用于其他 Linux 系统。我使用的案例环境如下所示。

- 机器配置：2 CPU，8GB 内存。
- 预先安装 stress 和 sysstat 包，如 apt install stress sysstat。

在这里，我先简单介绍一下 stress 和 sysstat。

stress 是一个 Linux 系统压力测试工具，这里我们用作异常进程模拟平均负载升高的场景。

而 sysstat 包含了常用的 Linux 性能工具，用来监控和分析系统的性能。我们的案例会用到这个包的两个命令 mpstat 和 pidstat。

- mpstat 是一个常用的多核 CPU 性能分析工具，用来实时查看每个 CPU 的性能指标，以及所有 CPU 的平均指标。
- pidstat 是一个常用的进程性能分析工具，用来实时查看进程的 CPU、内存、I/O 以及上下文切换等性能指标。

此外，每个场景都需要你开三个终端，登录到同一台 Linux 机器中。

实验之前，你先做好上面的准备。如果包的安装有问题，可以先在 Google 一下自行解决，如果还是解决不了，再来留言区找我，这事儿应该不难。

另外要注意，下面的所有命令，我们都是默认以 root 用户运行。所以，如果你是用普通用户登陆的系统，一定要先运行 sudo su root 命令切换到 root 用户。

如果上面的要求都已经完成了，你可以先用 uptime 命令，看一下测试前的平均负载情况：


```
$ uptime
..., load average: 0.11, 0.15, 0.09
```

场景一：CPU 密集型进程

首先，我们在第一个终端运行 stress 命令，模拟一个 CPU 使用率 100% 的场景：


```
$ stress --cpu 1 --timeout 600
```

接着，在第二个终端运行 uptime 查看平均负载的变化情况：


```
# -d 参数表示高亮显示变化的区域
$ watch -d uptime
..., load average: 1.00, 0.75, 0.39
```

最后，在第三个终端运行 mpstat 查看 CPU 使用率的变化情况：

```
# -P ALL 表示监控所有 CPU，后面数字 5 表示间隔 5 秒后输出一组数据
$ mpstat -P ALL 5
Linux 4.15.0 (ubuntu) 09/22/18 _x86_64_ (2 CPU)
13:30:06 CPU %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle
13:30:11 all 50.05 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 49.95
13:30:11 0 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 100.00
13:30:11 1 100.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

从终端二中可以看到，1 分钟的平均负载会慢慢增加到 1.00，而从终端三中还可以看到，正好有一个 CPU 的使用率为 100%，但它的 iowait 只有 0。这说明，平均负载的升高正是由于 CPU 使用率为 100% 。

那么，到底是哪个进程导致了 CPU 使用率为 100% 呢？你可以使用 pidstat 来查询：

```
# 间隔 5 秒后输出一组数据
$ pidstat -u 5 1
13:37:07 UID PID %usr %system %guest %wait %CPU CPU Command
13:37:12 0 2962 100.00 0.00 0.00 0.00 100.00 1 stress
```

从这里可以明显看到，stress 进程的 CPU 使用率为 100%。

场景二：I/O 密集型进程

首先还是运行 stress 命令，但这次模拟 I/O 压力，即不停地执行 sync：

```
$ stress -i 1 --timeout 600
```

还是在第二个终端运行 uptime 查看平均负载的变化情况：

```
$ watch -d uptime
..., load average: 1.06, 0.58, 0.37
```

然后，第三个终端运行 mpstat 查看 CPU 使用率的变化情况：

```
# 显示所有 CPU 的指标，并在间隔 5 秒输出一组数据
$ mpstat -P ALL 5 1
Linux 4.15.0 (ubuntu) 09/22/18 _x86_64_ (2 CPU)
13:41:28 CPU %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle
13:41:33 all 0.21 0.00 12.07 32.67 0.00 0.21 0.00 0.00 0.00 54.84
13:41:33 0 0.43 0.00 23.87 67.53 0.00 0.43 0.00 0.00 0.00 7.74
13:41:33 1 0.00 0.00 0.81 0.20 0.00 0.00 0.00 0.00 0.00 98.99
```

从这里可以看到，1 分钟的平均负载会慢慢增加到 1.06，其中一个 CPU 的系统 CPU 使用率升高到了 23.87，而 iowait 高达 67.53%。这说明，平均负载的升高是由于 iowait 的升高。

那么到底是哪个进程，导致 iowait 这么高呢？我们还是用 pidstat 来查询：

```
# 间隔 5 秒后输出一组数据，-u 表示 CPU 指标
$ pidstat -u 5 1
Linux 4.15.0 (ubuntu) 09/22/18 _x86_64_ (2 CPU)
13:42:08 UID PID %usr %system %guest %wait %CPU CPU Command
13:42:13 0 104 0.00 3.39 0.00 0.00 3.39 1 kworker/1:1H
13:42:13 0 109 0.00 0.40 0.00 0.00 0.40 0 kworker/0:1H
13:42:13 0 2997 2.00 35.53 0.00 3.99 37.52 1 stress
13:42:13 0 3057 0.00 0.40 0.00 0.00 0.40 0 pidstat
```

可以发现，还是 stress 进程导致的。

场景三：大量进程的场景

当系统中运行进程超出 CPU 运行能力时，就会出现等待 CPU 的进程。

比如，我们还是使用 stress，但这次模拟的是 8 个进程：

```
$ stress -c 8 --timeout 600
```

由于系统只有 2 个 CPU，明显比 8 个进程要少得多，因而，系统的 CPU 处于严重过载状态，平均负载高达 7.97：

```
$ uptime
..., load average: 7.97, 5.93, 3.02
```

接着再运行 pidstat 来看一下进程的情况：

```
# 间隔 5 秒后输出一组数据
$ pidstat -u 5 1
14:23:25 UID PID %usr %system %guest %wait %CPU CPU Command
14:23:30 0 3190 25.00 0.00 0.00 74.80 25.00 0 stress
14:23:30 0 3191 25.00 0.00 0.00 75.20 25.00 0 stress
14:23:30 0 3192 25.00 0.00 0.00 74.80 25.00 1 stress
14:23:30 0 3193 25.00 0.00 0.00 75.00 25.00 1 stress
14:23:30 0 3194 24.80 0.00 0.00 74.60 24.80 0 stress
14:23:30 0 3195 24.80 0.00 0.00 75.00 24.80 0 stress
14:23:30 0 3196 24.80 0.00 0.00 74.60 24.80 1 stress
14:23:30 0 3197 24.80 0.00 0.00 74.80 24.80 1 stress
14:23:30 0 3200 0.00 0.20 0.00 0.20 0.20 0 pidstat
```

可以看出，8 个进程在争抢 2 个 CPU，每个进程等待 CPU 的时间（也就是代码块中的 %wait 列）高达 75%。这些超出 CPU 计算能力的进程，最终导致 CPU 过载。

小结

分析完这三个案例，我再来归纳一下平均负载的理解。

平均负载提供了一个快速查看系统整体性能的手段，反映了整体的负载情况。但只看平均负载本身，我们并不能直接发现，到底是哪里出现了瓶颈。所以，在理解平均负载时，也要注意：

- 平均负载高有可能是 CPU 密集型进程导致的；
- 平均负载高并不一定代表 CPU 使用率高，还有可能是 I/O 更繁忙了；
- 当发现负载高的时候，你可以使用 mpstat、pidstat 等工具，辅助分析负载的来源。

思考

最后，我想邀请你一起来聊聊你所理解的平均负载，当你发现平均负载升高后，又是怎么分析排查的呢？你可以结合我前面的讲解，来总结自己的思考。欢迎在留言区和我讨论。

笔记本： Linux
创建时间： 2019/1/8 10:19 更新时间： 2019/1/16 9:43
标签： 性能优化
URL： https://time.geekbang.org/column/article/69859

03 | 基础篇：经常说的 CPU 上下文切换是什么意思？（上）

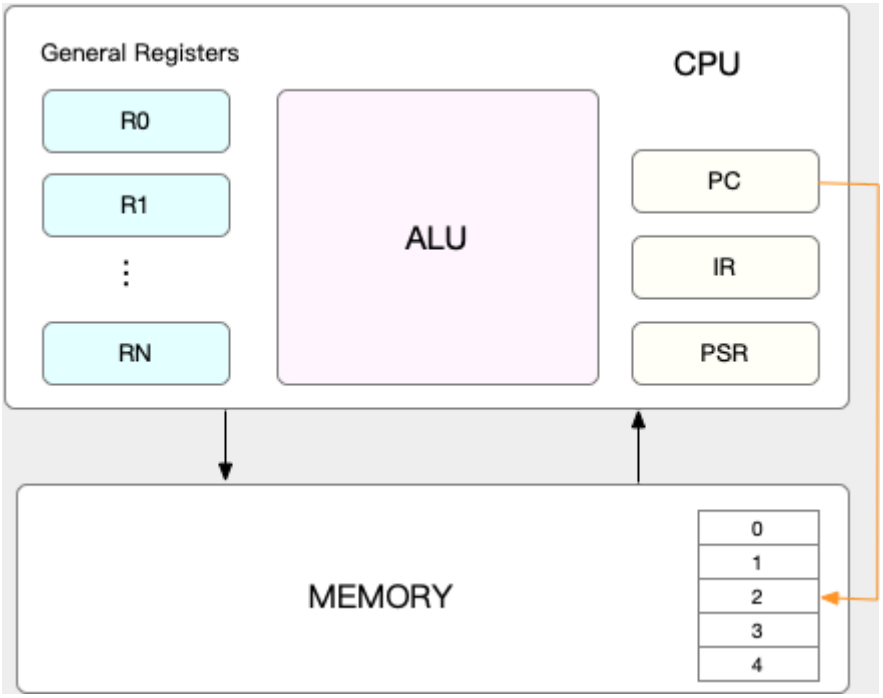
上一节，我给你讲了要怎么理解平均负载（ Load Average），并用三个案例展示了不同场景下平均负载升高的分析方法。这其中，多个进程竞争 CPU 就是一个经常被我们忽视的问题。

我想你一定很好奇，进程在竞争 CPU 的时候并没有真正运行，为什么还会导致系统的负载升高呢？看到今天的主题，你应该已经猜到了，CPU 上下文切换就是罪魁祸首。

我们都知道，Linux 是一个多任务操作系统，它支持远大于 CPU 数量的任务同时运行。当然，这些任务实际上并不是真的在同时运行，而是因为系统在很短的时间内，将 CPU 轮流分配给它们，造成多任务同时运行的错觉。

而在每个任务运行前，CPU 都需要知道任务从哪里加载、又从哪里开始运行，也就是说，需要系统事先帮它设置好 **CPU 寄存器和程序计数器**（Program Counter，PC）。

CPU 寄存器，是 CPU 内置的容量小、但速度极快的内存。而程序计数器，则是用来存储 CPU 正在执行的指令位置、或者即将执行的下一条指令位置。它们都是 CPU 在运行任何任务前，必须的依赖环境，因此也被叫做 **CPU 上下文**。



知道了什么是 CPU 上下文，我想你也很容易理解 **CPU 上下文切换**。CPU 上下文切换，就是先把前一个任务的 CPU 上下文（也就是 CPU 寄存器和程序计数器）保存起来，然后加载新任务的上下文到这些寄存器和程序计数器，最后再跳转到程序计数器所指的新位置，运行新任务。

而这些保存下来的上下文，会存储在系统内核中，并在任务重新调度执行时再次加载进来。这样就能保证任务原来的状态不受影响，让任务看起来还是连续运行。

我猜肯定会有人说，CPU 上下文切换无非就是更新了 CPU 寄存器的值嘛，但这些寄存器，本身就是为了快速运行任务而设计的，为什么会影响系统的 CPU 性能呢？

在回答这个问题前，不知道你有没有想过，操作系统管理的这些“任务”到底是什么呢？

也许你会说，任务就是进程，或者说任务就是线程。是的，进程和线程正是最常见的任务。但是除此之外，还有没有其他的任务呢？

不要忘了，硬件通过触发信号，会导致中断处理程序的调用，也是一种常见的任务。

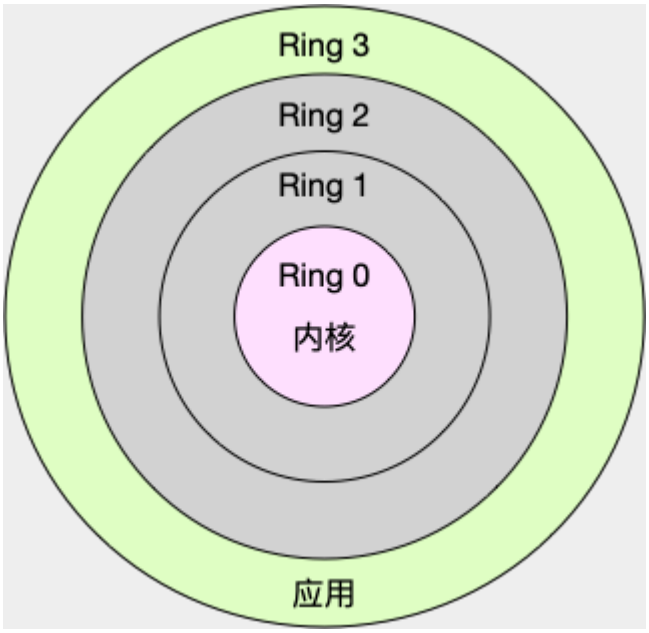
所以，根据任务的不同，CPU 的上下文切换就可以分为几个不同的场景，也就是**进程上下文切换**、**线程上下文切换**以及**中断上下文切换**。

这节课我就带你来看看，怎么理解这几个不同的上下文切换，以及它们为什么会引发 CPU 性能相关问题。

进程上下文切换

Linux 按照特权等级，把进程的运行空间分为内核空间和用户空间，分别对应着下图中， CPU 特权等级的 Ring 0 和 Ring 3。

- 内核空间（Ring 0）具有最高权限，可以直接访问所有资源；
- 用户空间（Ring 3）只能访问受限资源，不能直接访问内存等硬件设备，必须通过系统调用陷入到内核中，才能访问这些特权资源。



换个角度看，也就是说，进程既可以在用户空间运行，又可以在内核空间中运行。进程在用户空间运行时，被称为进程的用户态，而陷入内核空间的时候，被称为进程的内核态。

从用户态到内核态的转变，需要通过**系统调用**来完成。比如，当我们查看文件内容时，就需要多次系统调用来完成：首先调用 open() 打开文件，然后调用 read() 读取文件内容，并调用 write() 将内容写到标准输出，最后再调用 close() 关闭文件。

那么，系统调用的过程有没有发生 CPU 上下文的切换呢？答案自然是肯定的。

CPU 寄存器里原来用户态的指令位置，需要先保存起来。接着，为了执行内核态代码，CPU 寄存器需要更新为内核态指令的新位置。最后才是跳转到内核态运行内核任务。

而系统调用结束后，CPU 寄存器需要**恢复**原来保存的用户态，然后再切换到用户空间，继续运行进程。所以，一次系统调用的过程，其实是发生了两次 CPU 上下文切换。

不过，需要注意的是，系统调用过程中，并不会涉及到虚拟内存等进程用户态的资源，也不会切换进程。这跟我们通常所说的进程上下文切换是不一样的：

- 进程上下文切换，是指从一个进程切换到另一个进程运行。
- 而系统调用过程中一直是同一个进程在运行。

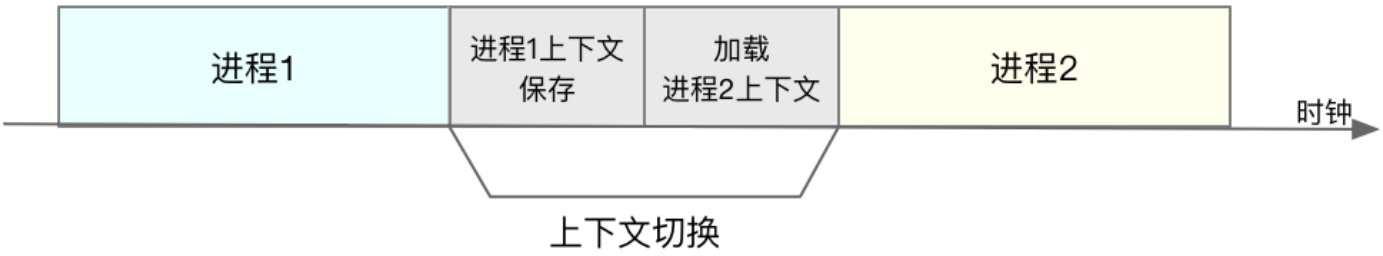
所以，**系统调用过程通常称为特权模式切换，而不是上下文切换**。但实际上，系统调用过程中，CPU 的上下文切换还是无法避免的。

那么，进程上下文切换跟系统调用又有什么区别呢？

首先，你需要知道，进程是由内核来管理和调度的，进程的切换只能发生在内核态。所以，进程的上下文不仅包括了虚拟内存、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状态。

因此，进程的上下文切换就比系统调用时多了一步：在保存当前进程的内核状态和 CPU 寄存器之前，需要先把该进程的虚拟内存、栈等保存下来；而加载了下一进程的内核态后，还需要刷新进程的虚拟内存和用户栈。

如下图所示，保存上下文和恢复上下文的过程并不是“免费”的，需要内核在 CPU 上运行才能完成。



根据 Tsuna 的测试报告，每次上下文切换都需要几十纳秒到数微秒的 CPU 时间。这个时间还是相当可观的，特别是在进程上下文切换次数较多的情况下，很容易导致 CPU 将大量时间耗费在寄存器、内核栈以及虚拟内存等资源的保存和恢复上，进而大大缩短了真正运行进程的时间。这也正是上一节中我们所讲的，导致平均负载升高的一个重要因素。

另外，我们知道， Linux 通过 TLB（Translation Lookaside Buffer）来管理虚拟内存到物理内存的映射关系。当虚拟内存更新后，TLB 也需要刷新，内存的访问也会随之变慢。特别是在多处理器系统上，缓存是被多个处理器共享的，刷新缓存不仅会影响当前处理器的进程，还会影响共享缓存的其他处理器的进程。

知道了进程上下文切换潜在的性能问题后，我们再来看，究竟什么时候会切换进程上下文。

显然，进程切换时才需要切换上下文，换句话说，只有在进程调度的时候，才需要切换上下文。Linux 为每个 CPU 都维护了一个就绪队列，将活跃进程（即正在运行和正在等待 CPU 的进程）按照优先级和等待 CPU 的时间排序，然后选择最需要 CPU 的进程，也就是优先级最高和等待 CPU 时间最长的进程来运行。

那么，进程在什么时候才会被调度到 CPU 上运行呢？

最容易想到的一个时机，就是进程执行完终止了，它之前使用的 CPU 会释放出来，这个时候再从就绪队列里，拿一个新的进程过来运行。其实还有很多其他场景，也会触发进程调度，在这里我给你逐个梳理下。

其一，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，就会被系统挂起，切换到其它正在等待 CPU 的进程运行。

其二，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。

其三，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。

其四，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。

最后一个，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。

了解这几个场景是非常有必要的，因为一旦出现上下文切换的性能问题，它们就是幕后凶手。

线程上下文切换

说完了进程的上下文切换，我们再来看看线程相关的问题。

线程与进程最大的区别在于，**线程是调度的基本单位，而进程则是资源拥有的基本单位**。说白了，所谓内核中的任务调度，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。所以，对于线程和进程，我们可以这么理解：

- 当进程只有一个线程时，可以认为进程就等于线程。
- 当进程拥有多个线程时，这些线程会共享相同的虚拟内存和全局变量等资源。这些资源在上下文切换时是不需要修改的。
- 另外，线程也有自己的私有数据，比如栈和寄存器等，这些在上下文切换时也是需要保存的。

这么一来，线程的上下文切换其实就可以分为两种情况：

第一种， 前后两个线程属于不同进程。此时，因为资源不共享，所以切换过程就跟进程上下文切换是一样。

第二种，前后两个线程属于同一个进程。此时，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。

到这里你应该也发现了，虽然同为上下文切换，但同进程内的线程切换，要比多进程间的切换消耗更少的资源，而这，也正是多线程代替多进程的一个优势。

中断上下文切换

除了前面两种上下文切换，还有一个场景也会切换 CPU 上下文，那就是中断。

为了快速响应硬件的事件，**中断处理会打断进程的正常调度和执行**，转而调用中断处理程序，响应设备事件。而在打断其他进程时，就需要将进程当前的状态保存下来，这样在中断结束后，进程仍然可以从原来的状态恢复运行。

跟进程上下文不同，中断上下文切换并不涉及到进程的用户态。所以，即便中断过程打断了一个正处在用户态的进程，也不需要保存和恢复这个进程的虚拟内存、全局变量等用户态资源。中断上下文，其实只包括内核态中断服务程序执行所必需的状态，包括 CPU 寄存器、内核堆栈、硬件中断参数等。

对同一个 CPU 来说，中断处理比进程拥有更高的优先级，所以中断上下文切换并不会与进程上下文切换同时发生。同样道理，由于中断会打断正常进程的调度和执行，所以大部分中断处理程序都短小精悍，以便尽可能快的执行结束。

另外，跟进程上下文切换一样，中断上下文切换也需要消耗 CPU，切换次数过多也会耗费大量的 CPU，甚至严重降低系统的整体性能。所以，当你发现中断次数过多时，就需要注意去排查它是否会给你的系统带来严重的性能问题。

小结

总结一下，不管是哪种场景导致的上下文切换，你都应该知道：

1. CPU 上下文切换，是保证 Linux 系统正常工作的核心功能之一，一般情况下不需要我们特别关注。
2. 但过多的上下文切换，会把 CPU 时间消耗在寄存器、内核栈以及虚拟内存等数据的保存和恢复上，从而缩短进程真正运行的时间，导致系统的整体性能大幅下降。

今天主要为你介绍这几种上下文切换的工作原理，下一节，我将继续案例实战，说说上下文切换问题的分析方法。

思考

最后，我想邀请你一起来聊聊，你所理解的 CPU 上下文切换。你可以结合今天的内容，总结自己的思路和看法，写下你的学习心得。

欢迎在留言区和我讨论。

笔记本： Linux
创建时间： 2019/1/8 10:33 更新时间： 2019/1/16 9:43
标签： 性能优化
URL： https://time.geekbang.org/column/article/70077

04 | 基础篇：经常说的 CPU 上下文切换是什么意思？（下）

上一节，我给你讲了 CPU 上下文切换的工作原理。简单回顾一下，CPU 上下文切换是保证 Linux 系统正常工作的一个核心功能，按照不同场景，可以分为进程上下文切换、线程上下文切换和中断上下文切换。具体的概念和区别，你也要在脑海中过一遍，忘了的话及时查看上一篇。

今天我们就接着来看，究竟怎么分析 CPU 上下文切换的问题。

怎么查看系统的上下文切换情况

通过前面学习我们知道，过多的上下文切换，会把 CPU 时间消耗在寄存器、内核栈以及虚拟内存等数据的保存和恢复上，缩短进程真正运行的时间，成了系统性能大幅下降的一个元凶。

既然上下文切换对系统性能影响那么大，你肯定迫不及待想知道，到底要怎么查看上下文切换呢？在这里，我们可以使用 vmstat 这个工具，来查询系统的上下文切换情况。

vmstat 是一个常用的系统性能分析工具，主要用来分析系统的内存使用情况，也常用来分析 CPU 上下文切换和中断的次数。

比如，下面就是一个 vmstat 的使用示例：

复制代码

```
# 每隔 5 秒输出 1 组数据
$ vmstat 5
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 0 7005360 91564 818900 0 0 0 0 25 33 0 0 100 0 0
```

我们一起来看这个结果，你可以先试着自己解读每列的含义。在这里，我重点强调下，需要特别关注的四列内容：

- cs（context switch）是每秒上下文切换的次数。
- in（interrupt）则是每秒中断的次数。
- r（Running or Runnable）是就绪队列的长度，也就是正在运行和等待 CPU 的进程数。

- b（Blocked）则是处于不可中断睡眠状态的进程数。

可以看到，这个例子中的上下文切换次数 cs 是 33 次，而系统中断次数 in 则是 25 次，而就绪队列长度 r 和不可中断状态进程数 b 都是 0。

vmstat 只给出了系统总体的上下文切换情况，要想查看每个进程的详细情况，就需要使用我们前面提到过的 pidstat 了。给它加上 -w 选项，你就可以查看每个进程上下文切换的情况了。

比如说：


```
# 每隔 5 秒输出 1 组数据
$ pidstat -w 5
Linux 4.15.0 (ubuntu) 09/23/18 _x86_64_ (2 CPU)

08:18:26 UID PID cswch/s nvcswch/s Command
08:18:31 0 1 0.20 0.00 systemd
08:18:31 0 8 5.40 0.00 rcu_sched
...
```

这个结果中有两列内容是我们的重点关注对象。一个是 cswch ，表示每秒自愿上下文切换（voluntary context switches）的次数，另一个则是 nvcswch ，表示每秒非自愿上下文切换（non voluntary context switches）的次数。

这两个概念你一定要牢牢记住，因为它们意味着不同的性能问题：

- 所谓**自愿上下文切换，是指进程无法获取所需资源，导致的上下文切换**。比如说， I/O、内存等系统资源不足时，就会发生自愿上下文切换。
- 而**非自愿上下文切换，则是指进程由于时间片已到等原因，被系统强制调度，进而发生的上下文切换**。比如说，大量进程都在争抢 CPU 时，就容易发生非自愿上下文切换。

案例分析

知道了怎么查看这些指标，另一个问题又来了，上下文切换频率是多少次才算正常呢？别急着要答案，同样的，我们先来看一个上下文切换的案例。通过案例实战演练，你自己就可以分析并找出这个标准了。

你的准备

今天的案例，我们将使用 sysbench 来模拟系统多线程调度切换的情况。

sysbench 是一个多线程的基准测试工具，一般用来评估不同系统参数下的数据库负载情况。当然，在这次案例中，我们只把它当成一个异常进程来看，作用是模拟上下文切换过多的问题。

下面的案例基于 Ubuntu 18.04，当然，其他的 Linux 系统同样适用。我使用的案例环境如下所示：

- 机器配置：2 CPU，8GB 内存
- 预先安装 sysbench 和 sysstat 包，如 apt install sysbench sysstat

正式操作开始前，你需要打开三个终端，登录到同一台 Linux 机器中，并安装好上面提到的两个软件包。包的安装，可以先 Google 一下自行解决，如果仍然有问题的，在留言区写下你的情况。

另外注意，下面所有命令，都**默认以 root 用户运行**。所以，如果你是用普通用户登陆的系统，记住先运行 sudo su root 命令切换到 root 用户。

安装完成后，你可以先用 vmstat 看一下空闲系统的上下文切换次数：


```
# 间隔 1 秒后输出 1 组数据
$ vmstat 1 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 0 6984064 92668 830896 0 0 2 19 19 35 1 0 99 0 0
```

这里你可以看到，现在的上下文切换次数 cs 是 35，而中断次数 in 是 19，r 和 b 都是 0。因为这会儿我并没有运行其他任务，所以它们就是空闲系统的上下文切换次数。

操作和分析

接下来，我们正式进入实战操作。

首先，在第一个终端里运行 sysbench ，模拟系统多线程调度的瓶颈：


```
# 以 10 个线程运行 5 分钟的基准测试，模拟多线程切换的问题
$ sysbench --threads=10 --max-time=300 threads run
```

接着，在第二个终端运行 vmstat ，观察上下文切换情况：

```
# 每隔 1 秒输出 1 组数据（需要 Ctrl+C 才结束）
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
6 0 0 6487428 118240 1292772 0 0 0 0 9019 1398830 16 84 0 0 0
8 0 0 6487428 118240 1292772 0 0 0 0 10191 1392312 16 84 0 0 0
```

你应该可以发现，cs 列的上下文切换次数从之前的 35 骤然上升到了 139 万。同时，注意观察其他几个指标：

- r 列：就绪队列的长度已经到了 8，远远超过了系统 CPU 的个数 2，所以肯定会有大量的 CPU 竞争。
- us（user）和 sy（system）列：这两列的 CPU 使用率加起来上升到了 100%，其中系统 CPU 使用率，也就是 sy 列高达 84%，说明 CPU 主要是被内核占用了。
- in 列：中断次数也上升到了 1 万左右，说明中断处理也是个潜在的问题。

综合这几个指标，我们可以知道，系统的就绪队列过长，也就是正在运行和等待 CPU 的进程数过多，导致了大量的上下文切换，而上下文切换又导致了系统 CPU 的占用率升高。

那么到底是什么进程导致了这些问题呢？

我们继续分析，在第三个终端再用 pidstat 来看一下， CPU 和进程上下文切换的情况：

```
# 每隔 1 秒输出 1 组数据（需要 Ctrl+C 才结束）
# -w 参数表示输出进程切换指标，而 -u 参数则表示输出 CPU 使用指标
$ pidstat -w -u 1
08:06:33 UID PID %usr %system %guest %wait %CPU CPU Command
08:06:34 0 10488 30.00 100.00 0.00 0.00 100.00 0 sysbench
08:06:34 0 26326 0.00 1.00 0.00 0.00 1.00 0 kworker/u4:2

08:06:33 UID PID cswch/s nvcswch/s Command
08:06:34 0 8 11.00 0.00 rcu_sched
08:06:34 0 16 1.00 0.00 ksoftirqd/1
08:06:34 0 471 1.00 0.00 hv_balloon
08:06:34 0 1230 1.00 0.00 iscsid
08:06:34 0 4089 1.00 0.00 kworker/1:5
08:06:34 0 4333 1.00 0.00 kworker/0:3
08:06:34 0 10499 1.00 224.00 pidstat
08:06:34 0 26326 236.00 0.00 kworker/u4:2
08:06:34 1000 26784 223.00 0.00 sshd
```

从 pidstat 的输出你可以发现，CPU 使用率的升高果然是 sysbench 导致的，它的 CPU 使用率已经达到了 100%。但上下文切换则是来自其他进程，包括非自愿上下文切换频率最高的 pidstat，以及自愿上下文切换频率最高的内核线程 kworker 和 sshd。

不过，细心的你肯定也发现了一个怪异的事儿：pidstat 输出的上下文切换次数，加起来也就几百，比 vmstat 的 139 万明显小了太多。这是怎么回事呢？难道是工具本身出了错吗？

别着急，在怀疑工具之前，我们再来回想一下，前面讲到的几种上下文切换场景。其中有一点提到， Linux 调度的基本单位实际上是线程，而我们的场景 sysbench 模拟的也是线程的调度问题，那么，是不是 pidstat 忽略了线程的数据呢？

通过运行 man pidstat ，你会发现，pidstat 默认显示进程的指标数据，加上 -t 参数后，才会输出线程的指标。

所以，我们可以在第三个终端里， Ctrl+C 停止刚才的 pidstat 命令，再加上 -t 参数，重试一下看看：

```
# 每隔 1 秒输出一组数据（需要 Ctrl+C 才结束）
# -wt 参数表示输出线程的上下文切换指标
$ pidstat -wt 1
08:14:05 UID TGID TID cswch/s nvcswch/s Command
...
08:14:05 0 10551 - 6.00 0.00 sysbench
08:14:05 0 - 10551 6.00 0.00 |__sysbench
08:14:05 0 - 10552 18911.00 103740.00 |__sysbench
08:14:05 0 - 10553 18915.00 100955.00 |__sysbench
08:14:05 0 - 10554 18827.00 103954.00 |__sysbench
...
```

现在你就能看到了，虽然 sysbench 进程（也就是主线程）的上下文切换次数看起来并不多，但它的子线程的上下文切换次数却有很多。看来，上下文切换罪魁祸首，还是过多的 sysbench 线程。

我们已经找到了上下文切换次数增多的根源，那是不是到这儿就可以结束了呢？

当然不是。不知道你还记不记得，前面在观察系统指标时，除了上下文切换频率骤然升高，还有一个指标也有很大的变化。是的，正是中断次数。中断次数也上升到了 1 万，但到底是什么类型的中断上升了，现在还不清楚。我们接下来继续抽丝剥茧找源头。

既然是中断，我们都知道，它只发生在内核态，而 pidstat 只是一个进程的性能分析工具，并不提供任何关于中断的详细信息，怎样才能知道中断发生的类型呢？

没错，那就是从 /proc/interrupts 这个只读文件中读取。/proc 实际上是 Linux 的一个虚拟文件系统，用于内核空间与用户空间之间的通信。/proc/interrupts 就是这种通信机制的一部分，提供了一个只读的中断使用情况。

我们还是在第三个终端里， Ctrl+C 停止刚才的 pidstat 命令，然后运行下面的命令，观察中断的变化情况：

```
# -d 参数表示高亮显示变化的区域
$ watch -d cat /proc/interrupts
CPU0 CPU1
...
RES: 2450431 5279697 Rescheduling interrupts
...
```

观察一段时间，你可以发现，变化速度最快的是**重调度中断**（RES），这个中断类型表示，唤醒空闲状态的 CPU 来调度新的任务运行。这是多处理器系统（SMP）中，调度器用来分散任务到不同 CPU 的机制，通常也被称为**处理器间中断**（Inter-Processor Interrupts，IPI）。

所以，这里的中断升高还是因为过多任务的调度问题，跟前面上下文切换次数的分析结果是一致的。

通过这个案例，你应该也发现了多工具、多方面指标对比观测的好处。如果最开始时，我们只用了 pidstat 观测，这些很严重的上下文切换线程，压根儿就发现不了了。

现在再回到最初的问题，每秒上下文切换多少次才算正常呢？

这个数值其实取决于系统本身的 CPU 性能。在我看来，如果系统的上下文切换次数比较稳定，那么从数百到一万以内，都应该算是正常的。但当上下文切换次数超过一万次，或者切换次数出现数量级的增长时，就很可能已经出现了性能问题。

这时，你还需要根据上下文切换的类型，再做具体分析。比方说：

- 自愿上下文切换变多了，说明进程都在等待资源，有可能发生了 I/O 等其他问题；
- 非自愿上下文切换变多了，说明进程都在被强制调度，也就是都在争抢 CPU，说明 CPU 的确成了瓶颈；

- 中断次数变多了，说明 CPU 被中断处理程序占用，还需要通过查看 /proc/interrupts 文件来分析具体的中断类型。

小结

今天，我通过一个 sysbench 的案例，给你讲了上下文切换问题的分析思路。碰到上下文切换次数过多的问题时，**我们可以借助 vmstat 、 pidstat 和 /proc/interrupts 等工具**，来辅助排查性能问题的根源。

思考

最后，我想请你一起来聊聊，你之前是怎么分析和排查上下文切换问题的。你可以结合这两节的内容和你自己的实际操作，来总结自己的思路。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中学习。

笔记本： Linux
创建时间： 2019/1/8 14:31 更新时间： 2019/1/16 9:43
标签： 性能优化
URL： https://time.geekbang.org/column/article/70476

05 | 基础篇：某个应用的CPU使用率居然达到100%，我该怎么办？

通过前两节对平均负载和 CPU 上下文切换的学习，我相信你对 CPU 的性能已经有了初步了解。不过我还是想问一下，在学这个专栏前，你最常用什么指标来描述系统的 CPU 性能呢？我想你的答案，可能不是平均负载，也不是 CPU 上下文切换，而是另一个更直观的指标—— CPU 使用率。

我们前面说过，CPU 使用率是单位时间内 CPU 使用情况的统计，以百分比的方式展示。那么，作为最常用也是最熟悉的 CPU 指标，你能说出 CPU 使用率到底是怎么算出来的吗？再有，诸如 top、ps 之类的性能工具展示的 %user、%nice、 %system、%iowait 、%steal 等等，你又能弄清楚它们之间的不同吗？

今天我就带你了解 CPU 使用率的内容，同时，我也会以我们最常用的反向代理服务器 Nginx 为例，带你在一步步操作和分析中深入理解。

CPU 使用率

在上一期我曾提到，Linux 作为一个多任务操作系统，将每个 CPU 的时间划分为很短的时间片，再通过调度器轮流分配给各个任务使用，因此造成多任务同时运行的错觉。

为了维护 CPU 时间，Linux 通过事先定义的节拍率（内核中表示为 HZ），触发时间中断，并使用全局变量 Jiffies 记录了开机以来的节拍数。每发生一次时间中断，Jiffies 的值就加 1。

节拍率 HZ 是内核的可配选项，可以设置为 100、250、1000 等。不同的系统可能设置不同数值，你可以通过查询 /boot/config 内核选项来查看它的配置值。比如在我的系统中，节拍率设置成了 250，也就是每秒钟触发 250 次时间中断。

复制代码

```
$ grep 'CONFIG_HZ=' /boot/config-$(uname -r)
CONFIG_HZ=250
```

同时，正因为节拍率 HZ 是内核选项，所以用户空间程序并不能直接访问。为了方便用户空间程序，内核还提供了一个用户空间节拍率 USER_HZ，它总是固定为 100，也就是 1/100 秒。这

样，用户空间程序并不需要关心内核中 HZ 被设置成了多少，因为它看到的总是固定值 USER_HZ。

Linux 通过 /proc 虚拟文件系统，向用户空间提供了系统内部状态的信息，而 /proc/stat 提供的就是系统的 CPU 和任务统计信息。比方说，如果你只关注 CPU 的话，可以执行下面的命令：

复制代码

```
# 只保留各个 CPU 的数据
$ cat /proc/stat | grep ^cpu
cpu 280580 7407 286084 172900810 83602 0 583 0 0 0
cpu0 144745 4181 176701 86423902 52076 0 301 0 0 0
cpu1 135834 3226 109383 86476907 31525 0 282 0 0 0
```

这里的输出结果是一个表格。其中，第一列表示的是 CPU 编号，如 cpu0、cpu1 ，而第一行没有编号的 cpu ，表示的是所有 CPU 的累加。其他列则表示不同场景下 CPU 的累加节拍数，它的单位是 USER_HZ，也就是 10 ms（1/100 秒），所以这其实就是不同场景下的 CPU 时间。

当然，这里每一列的顺序并不需要你背下来。你只要记住，有需要的时候，查询 man proc 就可以。不过，你要清楚 man proc 文档里每一列的涵义，它们都是 CPU 使用率相关的重要指标，你还会在很多其他的性能工具中看到它们。下面，我来依次解读一下。

- user（通常缩写为 us），代表用户态 CPU 时间。注意，它不包括下面的 nice 时间，但包括了 guest 时间。
- nice（通常缩写为 ni），代表低优先级用户态 CPU 时间，也就是进程的 nice 值被调整为 1-19 之间时的 CPU 时间。这里注意，nice 可取值范围是 -20 到 19，数值越大，优先级反而越低。
- system（通常缩写为 sys），代表内核态 CPU 时间。
- idle（通常缩写为 id），代表空闲时间。注意，它不包括等待 I/O 的时间（iowait）。
- iowait（通常缩写为 wa），代表等待 I/O 的 CPU 时间。
- irq（通常缩写为 hi），代表处理硬中断的 CPU 时间。
- softirq（通常缩写为 si），代表处理软中断的 CPU 时间。
- steal（通常缩写为 st），代表当系统运行在虚拟机中的时候，被其他虚拟机占用的 CPU 时间。

- guest（通常缩写为 guest），代表通过虚拟化运行其他操作系统的时间，也就是运行虚拟机的 CPU 时间。
- guest_nice（通常缩写为 gnice），代表以低优先级运行虚拟机的时间。

而我们通常所说的 **CPU 使用率，就是除了空闲时间外的其他时间占总 CPU 时间的百分比**，用公式来表示就是：

$$CPU\ 使用率 = 1 - \frac{空闲时间}{总\ CPU\ 时间}$$

根据这个公式，我们就可以从 /proc/stat 中的数据，很容易地计算出 CPU 使用率。当然，也可以用每一个场景的 CPU 时间，除以总的 CPU 时间，计算出每个场景的 CPU 使用率。

不过先不要着急计算，你能说出，直接用 /proc/stat 的数据，算的是什么时间段的 CPU 使用率吗？

看到这里，你应该想起来了，这是开机以来的节拍数累加值，所以直接算出来的，是开机以来的平均 CPU 使用率，一般没啥参考价值。

事实上，为了计算 CPU 使用率，性能工具一般都会取间隔一段时间（比如 3 秒）的两次值，作差后，再计算出这段时间内的平均 CPU 使用率，即

$$平均CPU使用率 = 1 - \frac{空闲时间_{new} - 空闲时间_{old}}{总CPU时间_{new} - 总CPU时间_{old}}$$

这个公式，就是我们用各种性能工具所看到的 CPU 使用率的实际计算方法。

现在，我们知道了系统 CPU 使用率的计算方法，那进程的呢？跟系统的指标类似，Linux 也给每个进程提供了运行情况的统计信息，也就是 /proc/[pid]/stat。不过，这个文件包含的数据就比较丰富了，总共有 52 列的数据。

当然，不用担心，因为你并不需要掌握每一列的含义。还是那句话，需要的时候，查 man proc 就行。

回过头来看，是不是说要查看 CPU 使用率，就必须先读取 /proc/stat 和 /proc/[pid]/stat 这两个文件，然后再按照上面的公式计算出来呢？

当然不是，各种各样的性能分析工具已经帮我们计算好了。不过要注意的是，**性能分析工具给出的都是间隔一段时间的平均 CPU 使用率，所以要注意间隔时间的设置**，特别是用多个工具对比分析时，你一定要保证它们用的是相同的间隔时间。

比如，对比一下 top 和 ps 这两个工具报告的 CPU 使用率，默认的结果很可能不一样，因为 top 默认使用 3 秒时间间隔，而 ps 使用的却是进程的整个生命周期。

怎么查看 CPU 使用率

知道了 CPU 使用率的含义后，我们再来看看要怎么查看 CPU 使用率。说到查看 CPU 使用率的工具，我猜你第一反应肯定是 top 和 ps。的确，top 和 ps 是最常用的性能分析工具：

- top 显示了系统总体的 CPU 和内存使用情况，以及各个进程的资源使用情况。
- ps 则只显示了每个进程的资源使用情况。

比如，top 的输出格式为：


```
# 默认每 3 秒刷新一次
$ top
top - 11:58:59 up 9 days, 22:47, 1 user, load average: 0.03, 0.02, 0.00
Tasks: 123 total, 1 running, 72 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.3 us, 0.3 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 8169348 total, 5606884 free, 334640 used, 2227824 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 7497908 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
1 root 20 0 78088 9288 6696 S 0.0 0.1 0:16.83 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.05 kthreadd
4 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker/0:0H
...
```

这个输出结果中，第三行 %Cpu 就是系统的 CPU 使用率，具体每一列的含义上一节都讲过，只是把 CPU 时间变换成了 CPU 使用率，我就不再重复讲了。不过需要注意，top 默认显示的是所有 CPU 的平均值，这个时候你只需要按下数字 1 ，就可以切换到每个 CPU 的使用率了。

继续往下看，空白行之后是进程的实时信息，每个进程都有一个 %CPU 列，表示进程的 CPU 使用率。它是用户态和内核态 CPU 使用率的总和，包括进程用户空间使用的 CPU、通过系统调用执行的内核空间 CPU 、以及在就绪队列等待运行的 CPU。在虚拟化环境中，它还包括了运行虚拟机占用的 CPU。

所以，到这里我们可以发现， top 并没有细分进程的用户态 CPU 和内核态 CPU。那要怎么查看每个进程的详细情况呢？你应该还记得上一节用到的 pidstat 吧，它正是一个专门分析每个进程 CPU 使用情况的工具。

比如，下面的 pidstat 命令，就间隔 1 秒展示了进程的 5 组 CPU 使用率，包括：

- 用户态 CPU 使用率 （%usr）；
- 内核态 CPU 使用率（%system）；
- 运行虚拟机 CPU 使用率（%guest）；
- 等待 CPU 使用率（%wait）；
- 以及总的 CPU 使用率（%CPU）。

最后的 Average 部分，还计算了 5 组数据的平均值。

复制代码

```
# 每隔 1 秒输出一组数据，共输出 5 组
$ pidstat 1 5
15:56:02 UID PID %usr %system %guest %wait %CPU CPU Command
15:56:03 0 15006 0.00 0.99 0.00 0.00 0.99 1 dockerd

...

Average: UID PID %usr %system %guest %wait %CPU CPU Command
Average: 0 15006 0.00 0.99 0.00 0.00 0.99 - dockerd
```

CPU 使用率过高怎么办？

通过 top、ps、pidstat 等工具，你能够轻松找到 CPU 使用率较高（比如 100% ）的进程。接下来，你可能又想知道，占用 CPU 的到底是代码里的哪个函数呢？找到它，你才能更高效、更针对性地进行优化。

我猜你第一个想到的，应该是 GDB（The GNU Project Debugger）， 这个功能强大的程序调试利器。的确，GDB 在调试程序错误方面很强大。但是，我又要来“挑刺”了。请你记住，GDB 并不适合在性能分析的早期应用。

为什么呢？因为 GDB 调试程序的过程会中断程序运行，这在线上环境往往是不允许的。所以，GDB 只适合用在性能分析的后期，当你找到了出问题的大致函数后，线下再借助它来进一步调试

函数内部的问题。

那么哪种工具适合在第一时间分析进程的 CPU 问题呢？我的推荐是 perf。perf 是 Linux 2.6.31 以后内置的性能分析工具。它以性能事件采样为基础，不仅可以分析系统的各种事件和内核性能，还可以用来分析指定应用程序的性能问题。

使用 perf 分析 CPU 性能问题，我来说两种最常见、也是我最喜欢的用法。

第一种常见用法是 perf top，类似于 top，它能够实时显示占用 CPU 时钟最多的函数或者指令，因此可以用来查找热点函数，使用界面如下所示：


```
$ perf top
Samples: 833 of event 'cpu-clock', Event count (approx.): 97742399
Overhead Shared Object Symbol
7.28% perf [.] 0x00000000001f78a4
4.72% [kernel] [k] vsnprintf
4.32% [kernel] [k] module_get_kallsym
3.65% [kernel] [k] _raw_spin_unlock_irqrestore
...
```

输出结果中，第一行包含三个数据，分别是采样数（Samples）、事件类型（event）和事件总数量（Event count）。比如这个例子中，perf 总共采集了 833 个 CPU 时钟事件，而总事件数则为 97742399。

另外，**采样数需要我们特别注意**。如果采样数过少（比如只有十几个），那下面的排序和百分比就没什么实际参考价值了。

再往下看是一个表格式样的数据，每一行包含四列，分别是：

- 第一列 Overhead ，是该符号的性能事件在所有采样中的比例，用百分比来表示。
- 第二列 Shared ，是该函数或指令所在的动态共享对象（Dynamic Shared Object），如内核、进程名、动态链接库名、内核模块名等。
- 第三列 Object ，是动态共享对象的类型。比如 [.] 表示用户空间的可执行程序、或者动态链接库，而 [k] 则表示内核空间。
- 最后一列 Symbol 是符号名，也就是函数名。当函数名未知时，用十六进制的地址来表示。

还是以上面的输出为例，我们可以看到，占用 CPU 时钟最多的是 perf 工具自身，不过它的比例也只有 7.28%，说明系统并没有 CPU 性能问题。 perf top 的使用你应该很清楚了吧。

接着再来看第二种常见用法，也就是 perf record 和 perf report。 perf top 虽然实时展示了系统的性能信息，但它的缺点是并不保存数据，也就无法用于离线或者后续的分析。而 perf record 则提供了保存数据的功能，保存后的数据，需要你用 perf report 解析展示。


```
$ perf record # 按 Ctrl+C 终止采样
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.452 MB perf.data (6093 samples) ]

$ perf report # 展示类似于 perf top 的报告
```

在实际使用中，我们还经常为 perf top 和 perf record 加上 -g 参数，开启调用关系的采样，方便我们根据调用链来分析性能问题。

案例

下面我们就以 Nginx + PHP 的 Web 服务为例，来看看当你发现 CPU 使用率过高的问题后，要怎么使用 top 等工具找出异常的进程，又要怎么利用 perf 找出引发性能问题的函数。

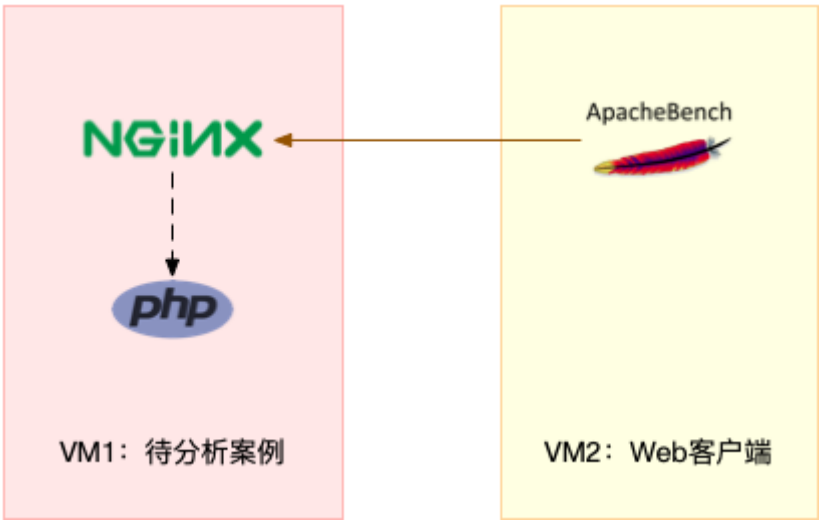
你的准备

以下案例基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

- 机器配置：2 CPU，8GB 内存
- 预先安装 docker、sysstat、perf、ab 等工具，如 apt install docker.io sysstat linux-tools-common apache2-utils

我先简单介绍一下这次新使用的工具 ab。ab（apache bench）是一个常用的 HTTP 服务性能测试工具，这里用来模拟 Ngnix 的客户端。由于 Nginx 和 PHP 的配置比较麻烦，我把它们打包成了两个 Docker 镜像，这样只需要运行两个容器，就可以得到模拟环境。

注意，这个案例要用到两台虚拟机，如下图所示：



你可以看到，其中一台用作 Web 服务器，来模拟性能问题；另一台用作 Web 服务器的客户端，来给 Web 服务增加压力请求。使用两台虚拟机是为了相互隔离，避免“交叉感染”。

接下来，我们打开两个终端，分别 SSH 登录到两台机器上，并安装上面提到的工具。

还是同样的“配方”。下面的所有命令，都默认假设以 root 用户运行，如果你是普通用户身份登陆系统，一定要先运行 sudo su root 命令切换到 root 用户。到这里，准备工作就完成了。

不过，操作之前，我还想再说一点。这次案例中 PHP 应用的核心逻辑比较简单，大部分人一眼就可以看出问题，但你要知道，实际生产环境中的源码就复杂多了。

所以，我希望你在按照步骤操作之前，先不要查看源码（避免先入为主），而是**把它当成一个黑盒来分析。**这样，你可以更好地理解整个解决思路，怎么从系统的资源使用问题出发，分析出瓶颈所在的应用、以及瓶颈在应用中的大概位置。

操作和分析

接下来，我们正式进入操作环节。

首先，在第一个终端执行下面的命令来运行 Nginx 和 PHP 应用：


```
$ docker run --name nginx -p 10000:80 -itd feisky/nginx
$ docker run --name phpfpm -itd --network container:nginx feisky/php-fpm
```

然后，在第二个终端使用 curl 访问 http://[VM1 的 IP]:10000，确认 Nginx 已正常启动。你应该可以看到 It works! 的响应。

```
# 192.168.0.10 是第一台虚拟机的 IP 地址
$ curl http://192.168.0.10:10000/
It works!
```

接着，我们来测试一下这个 Nginx 服务的性能。在第二个终端运行下面的 ab 命令：


```
# 并发 10 个请求测试 Nginx 性能，总共测试 100 个请求
$ ab -c 10 -n 100 http://192.168.0.10:10000/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
...
Requests per second: 11.63 [#/sec] (mean)
Time per request: 859.942 [ms] (mean)
...
```

从 ab 的输出结果我们可以看到，Nginx 能承受的每秒平均请求数只有 11.63。你一定在吐槽，这也太差了吧。那到底是哪里出了问题呢？我们用 top 和 pidstat 再来观察下。

这次，我们在第二个终端，将测试的请求总数增加到 10000。这样当你在第一个终端使用性能分析工具时， Nginx 的压力还是继续。

继续在第二个终端，运行 ab 命令：


```
$ ab -c 10 -n 10000 http://10.240.0.5:10000/
```

接着，回到第一个终端运行 top 命令，并按下数字 1 ，切换到每个 CPU 的使用率：


```
$ top
...
%Cpu0 : 98.7 us, 1.3 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 99.3 us, 0.7 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
...
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
21514 daemon 20 0 336696 16384 8712 R 41.9 0.2 0:06.00 php-fpm
21513 daemon 20 0 336696 13244 5572 R 40.2 0.2 0:06.08 php-fpm
21515 daemon 20 0 336696 16384 8712 R 40.2 0.2 0:05.67 php-fpm
21512 daemon 20 0 336696 13244 5572 R 39.9 0.2 0:05.87 php-fpm
21516 daemon 20 0 336696 16384 8712 R 35.9 0.2 0:05.61 php-fpm
```

这里可以看到，系统中有几个 php-fpm 进程的 CPU 使用率加起来接近 200%；而每个 CPU 的用户使用率（us）也已经超过了 98%，接近饱和。这样，我们就可以确认，正是用户空间的 php-fpm 进程，导致 CPU 使用率骤升。

那再往下走，怎么知道是 php-fpm 的哪个函数导致了 CPU 使用率升高呢？我们来用 perf 分析一下。在第一个终端运行下面的 perf 命令：

复制代码

```
# -g 开启调用关系分析，-p 指定 php-fpm 的进程号 21515
$ perf top -g -p 21515
```

按方向键切换到 php-fpm，再按下回车键展开 php-fpm 的调用关系，你会发现，调用关系最终到了 sqrt 和 add_function。看来，我们需要从这两个函数入手了。

```
Samples: 58K of event 'cpu-clock', Event count (approx.): 6934264349
  Children      Self  Shared Object       Symbol
-   96.94%     3.91%  php-fpm             [.] execute_ex
   - 57.86% execute_ex
      - 19.00% 0x8c4a7c
           3.59% sqrt
           1.18% 0x681b9d
           1.08% 0x681b99
      - 16.60% 0x98dea3
         - 4.83% 0x98dd97
              4.78% add_function
           1.23% 0x98dc03
        1.38% 0x9513cc
        1.31% 0x8cd729
```

我们拷贝出 Nginx 应用的源码，看看是不是调用了这两个函数：

复制代码

```
# 从容器 phpfpm 中将 PHP 源码拷贝出来
$ docker cp phpfpm:/app .

# 使用 grep 查找函数调用
$ grep sqrt -r app/ # 找到了 sqrt 调用
app/index.php: $x += sqrt($x);
$ grep add_function -r app/ # 没找到 add_function 调用，这其实是 PHP 内置函数
```

OK，原来只有 sqrt 函数在 app/index.php 文件中调用了。那最后一步，我们就该看看这个文件的源码了：

复制代码

```
$ cat app/index.php
<?php
// test only.
$x = 0.0001;
for ($i = 0; $i <= 1000000; $i++) {
$x += sqrt($x);
}

echo "It works!"
```

呀，有没有发现问题在哪里呢？我想你要笑话我了，居然犯了一个这么傻的错误，测试代码没删就直接发布应用了。为了方便你验证优化后的效果，我把修复后的应用也打包成了一个 Docker 镜像，你可以在第一个终端中执行下面的命令来运行它：

复制代码

```
# 停止原来的应用
$ docker rm -f nginx phpfpm
# 运行优化后的应用
$ docker run --name nginx -p 10000:80 -itd feisky/nginx:cpu-fix
$ docker run --name phpfpm -itd --network container:nginx feisky/php-fpm:cpu-fix
```

接着，到第二个终端来验证一下修复后的效果。首先 Ctrl+C 停止之前的 ab 命令后，再运行下面的命令：

复制代码

```
$ ab -c 10 -n 10000 http://10.240.0.5:10000/
...
Complete requests:      10000
Failed requests:        0
Total transferred:      1720000 bytes
HTML transferred:       90000 bytes
Requests per second:    2237.04 [#/sec] (mean)
Time per request:       4.470 [ms] (mean)
Time per request:       0.447 [ms] (mean, across all concurrent requests)
Transfer rate:          375.75 [Kbytes/sec] received
...
```

从这里你可以发现，现在每秒的平均请求数，已经从原来的 11 变成了 2237。

你看，就是这么很傻的一个小问题，却会极大的影响性能，并且查找起来也并不容易吧。当然，找到问题后，解决方法就简单多了，删除测试代码就可以了。

小结

CPU 使用率是最直观和最常用的系统性能指标，更是我们在排查性能问题时，通常会关注的第一个指标。所以我们更要熟悉它的含义，尤其要弄清楚用户（%user）、Nice（%nice）、系统（%system）、等待 I/O（%iowait）、中断（%irq）以及软中断（%softirq）这几种不同 CPU 的使用率。比如说：

- 用户 CPU 和 Nice CPU 高，说明用户态进程占用了较多的 CPU，所以应该着重排查进程的性能问题。
- 系统 CPU 高，说明内核态占用了较多的 CPU，所以应该着重排查内核线程或者系统调用的性能问题。
- I/O 等待 CPU 高，说明等待 I/O 的时间比较长，所以应该着重排查系统存储是不是出现了 I/O 问题。
- 软中断和硬中断高，说明软中断或硬中断的处理程序占用了较多的 CPU，所以应该着重排查内核中的中断服务程序。

碰到 CPU 使用率升高的问题，你可以借助 top、pidstat 等工具，确认引发 CPU 性能问题的来源；再使用 perf 等工具，排查出引起性能问题的具体函数。

思考

最后，我想邀请你一起来聊聊，你所理解的 CPU 使用率，以及在发现 CPU 使用率升高时，你又是怎么分析的呢？你可以结合今天的内容，和你自己的操作记录，来总结思路。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

笔记本： Linux
创建时间： 2019/1/9 21:58 更新时间： 2019/1/16 9:43
标签： 性能优化
URL： https://time.geekbang.org/column/article/70822

06 | 案例篇：系统的 CPU 使用率很高，但为啥却找不到高 CPU 的应用?

上一节我讲了 CPU 使用率是什么，并通过一个案例教你使用 top、vmstat、pidstat 等工具，排查高 CPU 使用率的进程，然后再使用 perf top 工具，定位应用内部函数的问题。不过就有人留言了，说似乎感觉高 CPU 使用率的问题，还是挺容易排查的。

那是不是所有 CPU 使用率高的问题，都可以这么分析呢？我想，你的答案应该是否定的。

回顾前面的内容，我们知道，系统的 CPU 使用率，不仅包括进程用户态和内核态的运行，还包括中断处理、等待 I/O 以及内核线程等。所以，**当你发现系统的 CPU 使用率很高的时候，不一定能找到相对应的高 CPU 使用率的进程**。

今天，我就用一个 Nginx + PHP 的 Web 服务的案例，带你来分析这种情况。

案例分析

你的准备

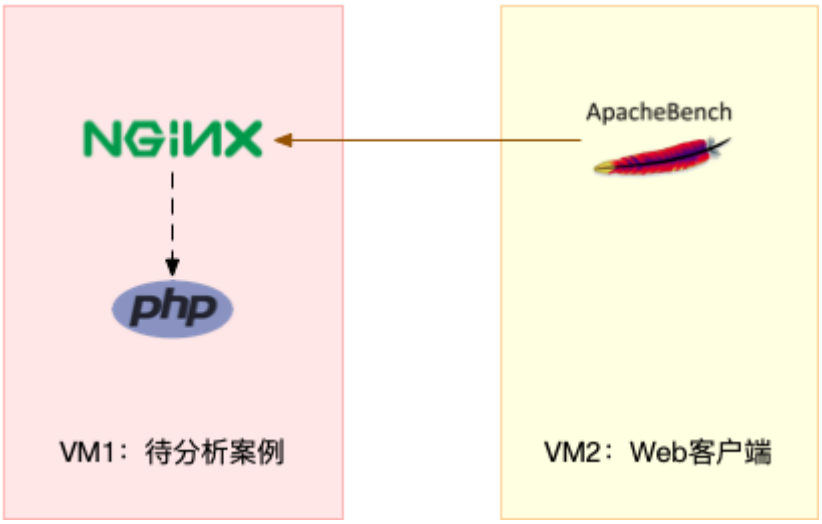
今天依旧探究系统 CPU 使用率高的情况，所以这次实验的准备工作，与上节课的准备工作基本相同，差别在于案例所用的 Docker 镜像不同。

本次案例还是基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

- 机器配置：2 CPU，8GB 内存
- 预先安装 docker、sysstat、perf、ab 等工具，如 apt install docker.io sysstat linux-tools-common apache2-utils

前面我们讲到过，ab（apache bench）是一个常用的 HTTP 服务性能测试工具，这里同样用来模拟 Nginx 的客户端。由于 Nginx 和 PHP 的配置比较麻烦，我把它们打包成了两个 Docker 镜像，这样只需要运行两个容器，就可以得到模拟环境。

注意，这个案例要用到两台虚拟机，如下图所示：



你可以看到，其中一台用作 Web 服务器，来模拟性能问题；另一台用作 Web 服务器的客户端，来给 Web 服务增加压力请求。使用两台虚拟机是为了相互隔离，避免“交叉感染”。

接下来，我们打开两个终端，分别 SSH 登录到两台机器上，并安装上述工具。

同样注意，下面所有命令都默认以 root 用户运行，如果你是用普通用户身份登陆系统，请运行 sudo su root 命令切换到 root 用户。

走到这一步，准备工作就完成了。接下来，我们正式进入操作环节。

> 温馨提示：案例中 PHP 应用的核心逻辑比较简单，你可能一眼就能看出问题，但实际生产环境中的源码就复杂多了。所以，我依旧建议，**操作之前别看源码**，避免先入为主，而要把它当成一个黑盒来分析。这样，你可以更好把握，怎么从系统的资源使用问题出发，分析出瓶颈所在的应用，以及瓶颈在应用中大概的位置。

操作和分析

首先，我们在第一个终端，执行下面的命令运行 Nginx 和 PHP 应用：

```
$ docker run --name nginx -p 10000:80 -itd feisky/nginx:sp
$ docker run --name phpfpm -itd --network container:nginx feisky/php-fpm:sp
```

然后，在第二个终端，使用 curl 访问 http://[VM1 的 IP]:10000，确认 Nginx 已正常启动。你应该可以看到 It works! 的响应。

```
# 192.168.0.10 是第一台虚拟机的 IP 地址
$ curl http://192.168.0.10:10000/
```

```
It works!
```

接着，我们来测试一下这个 Nginx 服务的性能。在第二个终端运行下面的 ab 命令。要注意，与上次操作不同的是，这次我们需要并发 100 个请求测试 Nginx 性能，总共测试 1000 个请求。

复制代码

```
# 并发 100 个请求测试 Nginx 性能，总共测试 1000 个请求
$ ab -c 100 -n 1000 http://192.168.0.10:10000/
This is ApacheBench, Version 2.3 <$Revision: 1706008 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
...
Requests per second: 87.86 [#/sec] (mean)
Time per request: 1138.229 [ms] (mean)
...
```

从 ab 的输出结果我们可以看到，Nginx 能承受的每秒平均请求数，只有 87 多一点，是不是感觉它的性能有点差呀。那么，到底是哪里出了问题呢？我们再用 top 和 pidstat 来观察一下。

这次，我们在第二个终端，将测试的并发请求数改成 5，同时把请求时长设置为 10 分钟（-t 600）。这样，当你在第一个终端使用性能分析工具时， Nginx 的压力还是继续的。

继续在第二个终端运行 ab 命令：

复制代码

```
$ ab -c 5 -t 600 http://192.168.0.10:10000/
```

然后，我们在第一个终端运行 top 命令，观察系统的 CPU 使用情况：

复制代码

```
$ top
...
%Cpu(s): 80.8 us, 15.1 sy, 0.0 ni, 2.8 id, 0.0 wa, 0.0 hi, 1.3 si, 0.0 st
...

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6882 root 20 0 8456 5052 3884 S 2.7 0.1 0:04.78 docker-containe
6947 systemd+ 20 0 33104 3716 2340 S 2.7 0.0 0:04.92 nginx
7494 daemon 20 0 336696 15012 7332 S 2.0 0.2 0:03.55 php-fpm
7495 daemon 20 0 336696 15160 7480 S 2.0 0.2 0:03.55 php-fpm
10547 daemon 20 0 336696 16200 8520 S 2.0 0.2 0:03.13 php-fpm
10155 daemon 20 0 336696 16200 8520 S 1.7 0.2 0:03.12 php-fpm
```

```
10552 daemon 20 0 336696 16200 8520 S 1.7 0.2 0:03.12 php-fpm
15006 root 20 0 1168608 66264 37536 S 1.0 0.8 9:39.51 dockerd
4323 root 20 0 0 0 0 I 0.3 0.0 0:00.87 kworker/u4:1
...
```

观察 top 输出的进程列表可以发现，CPU 使用率最高的进程也只不过才 2.7%，看起来并不高。

然而，再看系统 CPU 使用率（ %Cpu ）这一行，你会发现，系统的整体 CPU 使用率是比较高的：用户 CPU 使用率（us）已经到了 80%，系统 CPU 为 15.1%，而空闲 CPU （id）则只有 2.8%。

为什么用户 CPU 使用率这么高呢？我们再重新分析一下进程列表，看看有没有可疑进程：

- docker-containerd 进程是用来运行容器的，2.7% 的 CPU 使用率看起来正常；
- Nginx 和 php-fpm 是运行 Web 服务的，它们会占用一些 CPU 也不意外，并且 2% 的 CPU 使用率也不算高；
- 再往下看，后面的进程呢，只有 0.3% 的 CPU 使用率，看起来不太像会导致用户 CPU 使用率达到 80%。

那就奇怪了，明明用户 CPU 使用率都 80% 了，可我们挨个分析了一遍进程列表，还是找不到高 CPU 使用率的进程。看来 top 是不管用了，那还有其他工具可以查看进程 CPU 使用情况吗？不知道你记不记得我们的老朋友 pidstat，它可以用来分析进程的 CPU 使用情况。

接下来，我们还是在第一个终端，运行 pidstat 命令：


```
# 间隔 1 秒输出一组数据（按 Ctrl+C 结束）
$ pidstat 1
...
04:36:24 UID PID %usr %system %guest %wait %CPU CPU Command
04:36:25 0 6882 1.00 3.00 0.00 0.00 4.00 0 docker-containe
04:36:25 101 6947 1.00 2.00 0.00 1.00 3.00 1 nginx
04:36:25 1 14834 1.00 1.00 0.00 1.00 2.00 0 php-fpm
04:36:25 1 14835 1.00 1.00 0.00 1.00 2.00 0 php-fpm
04:36:25 1 14845 0.00 2.00 0.00 2.00 2.00 1 php-fpm
04:36:25 1 14855 0.00 1.00 0.00 1.00 1.00 1 php-fpm
04:36:25 1 14857 1.00 2.00 0.00 1.00 3.00 0 php-fpm
04:36:25 0 15006 0.00 1.00 0.00 0.00 1.00 0 dockerd
04:36:25 0 15801 0.00 1.00 0.00 0.00 1.00 1 pidstat
04:36:25 1 17084 1.00 0.00 0.00 2.00 1.00 0 stress
04:36:25 0 31116 0.00 1.00 0.00 0.00 1.00 0 atopacctd
```

```
16 ...
```

观察一会儿，你是不是发现，所有进程的 CPU 使用率也都不高啊，最高的 Docker 和 Nginx 也只有 4% 和 3%，即使所有进程的 CPU 使用率都加起来，也不过是 21%，离 80% 还差得远呢！

最早的时候，我碰到这种问题就完全懵了：明明用户 CPU 使用率已经高达 80%，但我却怎么都找不到是哪个进程的问题。到这里，你也可以想想，你是不是也遇到过这种情况？还能不能再做进一步的分析呢？

后来我发现，会出现这种情况，很可能是因为前面的分析漏了一些关键信息。你可以先暂停一下，自己往上翻，重新操作检查一遍。或者，我们一起返回去分析 top 的输出，看看能不能有新发现。

现在，我们回到第一个终端，重新运行 top 命令，并观察一会儿：

复制代码

```
$ top
top - 04:58:24 up 14 days, 15:47, 1 user, load average: 3.39, 3.82, 2.74
Tasks: 149 total, 6 running, 93 sleeping, 0 stopped, 0 zombie
%Cpu(s): 77.7 us, 19.3 sy, 0.0 ni, 2.0 id, 0.0 wa, 0.0 hi, 1.0 si, 0.0 st
KiB Mem : 8169348 total, 2543916 free, 457976 used, 5167456 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 7363908 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6947 systemd+ 20 0 33104 3764 2340 S 4.0 0.0 0:32.69 nginx
6882 root 20 0 12108 8360 3884 S 2.0 0.1 0:31.40 docker-containe
15465 daemon 20 0 336696 15256 7576 S 2.0 0.2 0:00.62 php-fpm
15466 daemon 20 0 336696 15196 7516 S 2.0 0.2 0:00.62 php-fpm
15489 daemon 20 0 336696 16200 8520 S 2.0 0.2 0:00.62 php-fpm
6948 systemd+ 20 0 33104 3764 2340 S 1.0 0.0 0:00.95 nginx
15006 root 20 0 1168608 65632 37536 S 1.0 0.8 9:51.09 dockerd
15476 daemon 20 0 336696 16200 8520 S 1.0 0.2 0:00.61 php-fpm
15477 daemon 20 0 336696 16200 8520 S 1.0 0.2 0:00.61 php-fpm
24340 daemon 20 0 8184 1616 536 R 1.0 0.0 0:00.01 stress
24342 daemon 20 0 8196 1580 492 R 1.0 0.0 0:00.01 stress
24344 daemon 20 0 8188 1056 492 R 1.0 0.0 0:00.01 stress
24347 daemon 20 0 8184 1356 540 R 1.0 0.0 0:00.01 stress
...
```

这次从头开始看 top 的每行输出，咦？Tasks 这一行看起来有点奇怪，就绪队列中居然有 6 个 Running 状态的进程（6 running），是不是有点多呢？

回想一下 ab 测试的参数，并发请求数是 5。再看进程列表里， php-fpm 的数量也是 5，再加上 Nginx，好像同时有 6 个进程也并不奇怪。但真的是这样吗?

再仔细看进程列表，这次主要看 Running（R） 状态的进程。你有没有发现， Nginx 和所有的 php-fpm 都处于 Sleep（S）状态，而真正处于 Running（R）状态的，却是几个 stress 进程。这几个 stress 进程就比较奇怪了，需要我们做进一步的分析。

我们还是使用 pidstat 来分析这几个进程，并且使用 -p 选项指定进程的 PID。首先，从上面 top 的结果中，找到这几个进程的 PID。比如，先随便找一个 24344，然后用 pidstat 命令看一下它的 CPU 使用情况：

复制代码

```
$ pidstat -p 24344

16:14:55 UID PID %usr %system %guest %wait %CPU CPU Command
```

奇怪，居然没有任何输出。难道是 pidstat 命令出问题了吗？之前我说过，**在怀疑性能工具出问题前，最好还是先用其他工具交叉确认一下**。那用什么工具呢？ ps 应该是最简单易用的。我们在终端里运行下面的命令，看看 24344 进程的状态：

复制代码

```
# 从所有进程中查找 PID 是 24344 的进程
$ ps aux | grep 24344
root 9628 0.0 0.0 14856 1096 pts/0 S+ 16:15 0:00 grep --color=auto 24344
```

还是没有输出。现在终于发现问题，原来这个进程已经不存在了，所以 pidstat 就没有任何输出。既然进程都没了，那性能问题应该也跟着没了吧。我们再用 top 命令确认一下：

复制代码

```
$ top
...
%Cpu(s): 80.9 us, 14.9 sy, 0.0 ni, 2.8 id, 0.0 wa, 0.0 hi, 1.3 si, 0.0 st
...

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
6882 root 20 0 12108 8360 3884 S 2.7 0.1 0:45.63 docker-containe
6947 systemd+ 20 0 33104 3764 2340 R 2.7 0.0 0:47.79 nginx
3865 daemon 20 0 336696 15056 7376 S 2.0 0.2 0:00.15 php-fpm
6779 daemon 20 0 8184 1112 556 R 0.3 0.0 0:00.01 stress
...
```

好像又错了。结果还跟原来一样，用户 CPU 使用率还是高达 80.9%，系统 CPU 接近 15%，而空闲 CPU 只有 2.8%，Running 状态的进程有 Nginx、stress 等。

可是，刚刚我们看到 stress 进程不存在了，怎么现在还在运行呢？再细看一下 top 的输出，原来，这次 stress 进程的 PID 跟前面不一样了，原来的 PID 24344 不见了，现在的是 6779。

进程的 PID 在变，这说明什么呢？在我看来，要么是这些进程在不停地重启，要么就是全新的进程，这无非也就两个原因：

- 第一个原因，进程在不停地崩溃重启，比如因为段错误、配置错误等等，这时，进程在退出后可能又被监控系统自动重启了。
- 第二个原因，这些进程都是短时进程，也就是在其他应用内部通过 exec 调用的外面命令。这些命令一般都只运行很短的时间就会结束，你很难用 top 这种间隔时间比较长的工具发现（上面的案例，我们碰巧发现了）。

至于 stress，我们前面提到过，它是一个常用的压力测试工具。它的 PID 在不断变化中，看起来像是被其他进程调用的短时进程。要想继续分析下去，还得找到它们的父进程。

要怎么查找一个进程的父进程呢？没错，用 pstree 就可以用树状形式显示所有进程之间的关系：


```
$ pstree | grep stress
|-docker-containe-+-php-fpm-+-php-fpm---sh---stress
| |-3*[php-fpm---sh---stress---stress]
```

从这里可以看到，stress 是被 php-fpm 调用的子进程，并且进程数量不止一个（这里是 3 个）。找到父进程后，我们能进入 app 的内部分析了。

首先，当然应该去看看它的源码。运行下面的命令，把案例应用的源码拷贝到 app 目录，然后再执行 grep 查找是不是有代码再调用 stress 命令：


```
# 拷贝源码到本地
$ docker cp phpfpm:/app .

# grep 查找看看是不是有代码在调用 stress 命令
$ grep stress -r app
app/index.php:// fake I/O with stress (via write()/unlink()).
```

```
app/index.php:$result = exec("/usr/local/bin/stress -t 1 -d 1 2>&1", $output, $status);
```

找到了，果然是 app/index.php 文件中直接调用了 stress 命令。

再来看看 app/index.php 的源代码：

复制代码

```
$ cat app/index.php
<?php
// fake I/O with stress (via write()/unlink()).
$result = exec("/usr/local/bin/stress -t 1 -d 1 2>&1", $output, $status);
if (isset($_GET["verbose"]) && $_GET["verbose"]==1 && $status != 0) {
echo "Server internal error: ";
print_r($output);
} else {
echo "It works!";
}
?>
```

可以看到，源码里对每个请求都会调用一个 stress 命令，模拟 I/O 压力。从注释上看，stress 会通过 write() 和 unlink() 对 I/O 进程进行压测，看来，这应该就是系统 CPU 使用率升高的根源了。

不过，stress 模拟的是 I/O 压力，而之前在 top 的输出中看到的，却一直是用户 CPU 和系统 CPU 升高，并没见到 iowait 升高。这又是怎么回事呢？stress 到底是不是 CPU 使用率升高的原因呢？

我们还得继续往下走。从代码中可以看到，给请求加入 verbose=1 参数后，就可以查看 stress 的输出。你先试试看，在第二个终端运行：

复制代码

```
$ curl http://192.168.0.10:10000?verbose=1
Server internal error: Array
(
[0] => stress: info: [19607] dispatching hogs: 0 cpu, 0 io, 0 vm, 1 hdd
[1] => stress: FAIL: [19608] (563) mkstemp failed: Permission denied
[2] => stress: FAIL: [19607] (394) <-- worker 19608 returned error 1
[3] => stress: WARN: [19607] (396) now reaping child worker processes
[4] => stress: FAIL: [19607] (400) kill error: No such process
[5] => stress: FAIL: [19607] (451) failed run completed in 0s
)
```

看错误消息 mkstemp failed: Permission denied ，以及 failed run completed in 0s。原来 stress 命令并没有成功，它因为权限问题失败退出了。看来，我们发现了一个 PHP 调用外部 stress 命令的 bug：没有权限创建临时文件。

从这里我们可以猜测，正是由于权限错误，大量的 stress 进程在启动时初始化失败，进而导致用户 CPU 使用率的升高。

分析出问题来源，下一步是不是就要开始优化了呢？当然不是！既然只是猜测，那就需要再确认一下，这个猜测到底对不对，是不是真的有大量的 stress 进程。该用什么工具或指标呢？

我们前面已经用了 top、pidstat、pstree 等工具，没有发现大量的 stress 进程。那么，还有什么其他的工具可以用吗？

还记得上一期提到的 perf 吗？它可以用来分析 CPU 性能事件，用在这里就很合适。依旧在第一个终端中运行 perf record -g 命令 ，并等待一会儿（比如 15 秒）后按 Ctrl+C 退出。然后再运行 perf report 查看报告：

复制代码

```
# 记录性能事件，等待大约 15 秒后按 Ctrl+C 退出
$ perf record -g

# 查看报告
$ perf report
```

这样，你就可以看到下图这个性能报告：

```
Samples: 137K of event 'cpu-clock', Event count (approx.): 34267500000
  Children      Self  Command          Shared Object                Symbol
-   77.13%     0.00%  stress           stress                       [.] 0x000000000000168d
   - 0x168d
      + 25.97% random_r
      + 18.62% random
        5.68% rand
      + 4.07% 0x2f25
        3.55% 0x2eff
        3.02% 0x2ee5
        2.26% 0xe80
        2.19% 0x2ef3
        2.16% 0x2f09
        1.87% 0x2f18
        1.56% 0x2f29
        1.36% 0x2f1e
        1.29% 0x2f1b
        1.22% 0x2f14
        0.97% 0x2f10
        0.71% 0x2f0d
+   25.97%    24.84%  stress           libc-2.24.so                 [.] random_r
+   18.62%    17.86%  stress           libc-2.24.so                 [.] random
+    5.68%     5.43%  stress           libc-2.24.so                 [.] rand
+    5.66%     0.00%  swapper          [kernel.vmlinux]             [k] 0x00000000002000d5
+    5.66%     0.00%  swapper          [kernel.vmlinux]             [k] cpu_startup_entry
+    5.66%     0.00%  swapper          [kernel.vmlinux]             [k] do_idle
+    5.60%     0.00%  swapper          [kernel.vmlinux]             [k] default_idle_call
+    5.60%     0.00%  swapper          [kernel.vmlinux]             [k] arch_cpu_idle
+    5.59%     0.00%  swapper          [kernel.vmlinux]             [k] default_idle
+    5.59%     5.42%  swapper          [kernel.vmlinux]             [k] native_safe_halt
+    4.37%     0.00%  php-fpm          [kernel.vmlinux]             [k] entry_SYSCALL_64
```

你看，stress 占了所有 CPU 时钟事件的 77%，而 stress 调用调用栈中比例最高的，是随机数生成函数 random()，看来它的确就是 CPU 使用率升高的元凶了。随后的优化就很简单了，只要修复权限问题，并减少或删除 stress 的调用，就可以减轻系统的 CPU 压力。

当然，实际生产环境中的问题一般都要比这个案例复杂，在你找到触发瓶颈的命令行后，却可能发现，这个外部命令的调用过程是应用核心逻辑的一部分，并不能轻易减少或者删除。

这时，你就得继续排查，为什么被调用的命令，会导致 CPU 使用率升高或 I/O 升高等问题。这些复杂场景的案例，我会在后面的综合实战里详细分析。

最后，在案例结束时，不要忘了清理环境，执行下面的 Docker 命令，停止案例中用到的 Nginx 进程：

复制代码

```
$ docker rm -f nginx phpfpm
```

execsnoop

在这个案例中，我们使用了 top、pidstat、pstree 等工具分析了系统 CPU 使用率高的问题，并发现 CPU 升高是短时进程 stress 导致的，但是整个分析过程还是比较复杂的。对于这类问题，有没有更好的方法监控呢？

execsnoop 就是一个专为短时进程设计的工具。它通过 ftrace 实时监控进程的 exec() 行为，并输出短时进程的基本信息，包括进程 PID、父进程 PID、命令行参数以及执行的结果。

比如，用 execsnoop 监控上述案例，就可以直接得到 stress 进程的父进程 PID 以及它的命令行参数，并可以发现大量的 stress 进程在不停启动：

```
# 按 Ctrl+C 结束
$ execsnoop
PCOMM PID PPID RET ARGS
sh 30394 30393 0
stress 30396 30394 0 /usr/local/bin/stress -t 1 -d 1
sh 30398 30393 0
stress 30399 30398 0 /usr/local/bin/stress -t 1 -d 1
sh 30402 30400 0
stress 30403 30402 0 /usr/local/bin/stress -t 1 -d 1
sh 30405 30393 0
stress 30407 30405 0 /usr/local/bin/stress -t 1 -d 1
...
```

execsnoop 所用的 ftrace 是一种常用的动态追踪技术，一般用于分析 Linux 内核的运行时行为，后面课程我也会详细介绍并带你使用。

小结

碰到常规问题无法解释的 CPU 使用率情况时，首先要想到有可能是短时应用导致的问题，比如有可能是下面这两种情况。

- 第一，**应用里直接调用了其他二进制程序，这些程序通常运行时间比较短，通过 top 等工具也不容易发现。**
- 第二，**应用本身在不停地崩溃重启，而启动过程的资源初始化，很可能会占用相当多的 CPU。**

对于这类进程，我们可以用 pstree 或者 execsnoop 找到它们的父进程，再从父进程所在的应用入手，排查问题的根源。

思考

最后，我想邀请你一起来聊聊，你所碰到的 CPU 性能问题。有没有哪个印象深刻的经历可以跟我分享呢？或者，在今天的案例操作中，你遇到了什么问题，又解决了哪些呢？你可以结合我的讲述，总结自己的思路。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

笔记本： Linux
创建时间： 2019/1/10 11:07 更新时间： 2019/1/16 9:42
标签： 性能优化
URL： https://time.geekbang.org/column/article/71064

07 | 案例篇：系统中出现大量不可中断进程和僵尸进程怎么办？（上）

上一节，我用一个 Nginx+PHP 的案例，给你讲了服务器 CPU 使用率高的分析和应对方法。这里你一定要记得，当碰到无法解释的 CPU 使用率问题时，先要检查一下是不是短时应用在捣鬼。

短时应用的运行时间比较短，很难在 top 或者 ps 这类展示系统概要和进程快照的工具中发现，你需要使用记录事件的工具来配合诊断，比如 execsnoop 或者 perf top。

这些思路你不用刻意去背，多练习几次，多在操作中思考，你便能灵活运用。

另外，我们还讲到 CPU 使用率的类型。除了上一节提到的用户 CPU 之外，它还包括系统 CPU（比如上下文切换）、等待 I/O 的 CPU（比如等待磁盘的响应）以及中断 CPU（包括软中断和硬中断）等。

我们已经在上下文切换的文章中，一起分析了系统 CPU 使用率高的问题，剩下的等待 I/O 的 CPU 使用率（以下简称为 iowait）升高，也是最常见的一个服务器性能问题。今天我们就来看一个多进程 I/O 的案例，并分析这种情况。

进程状态

当 iowait 升高时，进程很可能因为得不到硬件的响应，而长时间处于不可中断状态。从 ps 或者 top 命令的输出中，你可以发现它们都处于 D 状态，也就是不可中断状态（Uninterruptible Sleep）。既然说到了进程的状态，进程有哪些状态你还记得吗？我们先来回顾一下。

top 和 ps 是最常用的查看进程状态的工具，我们就从 top 的输出开始。下面是一个 top 命令输出的示例，S 列（也就是 Status 列）表示进程的状态。从这个示例里，你可以看到 R、D、Z、S、I 等几个状态，它们分别是什么意思呢？

复制代码

```
$ top
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
28961 root 20 0 43816 3148 4040 R 3.2 0.0 0:00.01 top
```

```
4 620 root 20 0 37280 33676 908 D 0.3 0.4 0:00.01 app
5 1 root 20 0 160072 9416 6752 S 0.0 0.1 0:37.64 systemd
6 1896 root 20 0 0 0 0 Z 0.0 0.0 0:00.00 devapp
7 2 root 20 0 0 0 0 S 0.0 0.0 0:00.10 kthreadd
8 4 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 kworker/0:0H
9 6 root 0 -20 0 0 0 I 0.0 0.0 0:00.00 mm_percpu_wq
10 7 root 20 0 0 0 0 S 0.0 0.0 0:06.37 ksoftirqd/0
```

我们挨个来看一下：

- **R** 是 Running 或 Runnable 的缩写，表示进程在 CPU 的就绪队列中，正在运行或者正在等待运行。
- **D** 是 Disk Sleep 的缩写，也就是不可中断状态睡眠（Uninterruptible Sleep），一般表示进程正在跟硬件交互，并且交互过程不允许被其他进程或中断打断。
- **Z** 是 Zombie 的缩写，如果你玩过“植物大战僵尸”这款游戏，应该知道它的意思。它表示僵尸进程，也就是进程实际上已经结束了，但是父进程还没有回收它的资源（比如进程的描述符、PID 等）。
- **S** 是 Interruptible Sleep 的缩写，也就是可中断状态睡眠，表示进程因为等待某个事件而被系统挂起。当进程等待的事件发生时，它会被唤醒并进入 R 状态。
- **I** 是 Idle 的缩写，也就是空闲状态，用在不可中断睡眠的内核线程上。前面说了，硬件交互导致的不可中断进程用 D 表示，但对某些内核线程来说，它们有可能实际上并没有任何负载，用 Idle 正是为了区分这种情况。要注意，D 状态的进程会导致平均负载升高， I 状态的进程却不会。

当然了，上面的示例并没有包括进程的所有状态。除了以上 5 个状态，进程还包括下面这 2 个状态。

第一个是 **T 或者 t**，也就是 Stopped 或 Traced 的缩写，表示进程处于暂停或者跟踪状态。

向一个进程发送 SIGSTOP 信号，它就会因响应这个信号变成暂停状态（Stopped）；再向它发送 SIGCONT 信号，进程又会恢复运行（如果进程是终端里直接启动的，则需要你用 fg 命令，恢复

到前台运行）。

而当你用调试器（如 gdb）调试一个进程时，在使用断点中断进程后，进程就会变成跟踪状态，这其实也是一种特殊的暂停状态，只不过你可以用调试器来跟踪并按需要控制进程的运行。

另一个是 **X**，也就是 Dead 的缩写，表示进程已经消亡，所以你不会在 top 或者 ps 命令中看到它。

了解了这些，我们再回到今天的主题。先看不可中断状态，这其实是为了保证进程数据与硬件状态一致，并且正常情况下，不可中断状态在很短时间内就会结束。所以，短时的不可中断状态进程，我们一般可以忽略。

但如果系统或硬件发生了故障，进程可能会在不可中断状态保持很久，甚至导致系统中出现大量不可中断进程。这时，你就得注意下，系统是不是出现了 I/O 等性能问题。

再看僵尸进程，这是多进程应用很容易碰到的问题。正常情况下，当一个进程创建了子进程后，它应该通过系统调用 wait() 或者 waitpid() 等待子进程结束，回收子进程的资源；而子进程在结束时，会向它的父进程发送 SIGCHLD 信号，所以，父进程还可以注册 SIGCHLD 信号的处理函数，异步回收资源。

如果父进程没这么做，或是子进程执行太快，父进程还没来得及处理子进程状态，子进程就已经提前退出，那这时的子进程就会变成僵尸进程。换句话说，父亲应该一直对儿子负责，善始善终，如果不作为或者跟不上，都会导致“问题少年”的出现。

通常，僵尸进程持续的时间都比较短，在父进程回收它的资源后就会消亡；或者在父进程退出后，由 init 进程回收后也会消亡。

一旦父进程没有处理子进程的终止，还一直保持运行状态，那么子进程就会一直处于僵尸状态。大量的僵尸进程会用尽 PID 进程号，导致新进程不能创建，所以这种情况一定要避免。

案例分析

接下来，我将用一个多进程应用的案例，带你分析大量不可中断状态和僵尸状态进程的问题。这个应用基于 C 开发，由于它的编译和运行步骤比较麻烦，我把它打包成了一个 Docker 镜像。这样，你只需要运行一个 Docker 容器就可以得到模拟环境。

你的准备

下面的案例仍然基于 Ubuntu 18.04，同样适用于其他的 Linux 系统。我使用的案例环境如下所示：

- 机器配置：2 CPU，8GB 内存
- 预先安装 docker、sysstat、dstat 等工具，如 apt install docker.io dstat sysstat

这里，dstat 是一个新的性能工具，它吸收了 vmstat、iostat、ifstat 等几种工具的优点，可以同时观察系统的 CPU、磁盘 I/O、网络以及内存使用情况。

接下来，我们打开一个终端，SSH 登录到机器上，并安装上述工具。

注意，以下所有命令都默认以 root 用户运行，如果你用普通用户身份登陆系统，请运行 sudo su root 命令切换到 root 用户。

如果安装过程有问题，你可以先上网搜索解决，实在解决不了的，记得在留言区向我提问。

> 温馨提示：案例应用的核心代码逻辑比较简单，你可能一眼就能看出问题，但实际生产环境中的源码就复杂多了。所以，我依旧建议，操作之前别看源码，避免先入为主，而要把它当成一个黑盒来分析，这样你可以更好地根据现象分析问题。你姑且当成你工作中的一次演练，这样效果更佳。

操作和分析

安装完成后，我们首先执行下面的命令运行案例应用：

复制代码

```
$ docker run --privileged --name=app -itd feisky/app:iowait
```

然后，输入 ps 命令，确认案例应用已正常启动。如果一切正常，你应该可以看到如下所示的输出：

复制代码

```
$ ps aux | grep /app
root 4009 0.0 0.0 4376 1008 pts/0 Ss+ 05:51 0:00 /app
root 4287 0.6 0.4 37280 33660 pts/0 D+ 05:54 0:00 /app
root 4288 0.6 0.4 37280 33668 pts/0 D+ 05:54 0:00 /app
```

从这个界面，我们可以发现多个 app 进程已经启动，并且它们的状态分别是 Ss+ 和 D+。其中，S 表示可中断睡眠状态，D 表示不可中断睡眠状态，我们在前面刚学过，那后面的 s 和 + 是什么意思呢？不知道也没关系，查一下 man ps 就可以。现在记住，s 表示这个进程是一个会话的领导进程，而 + 表示前台进程组。

这里又出现了两个新概念，**进程组**和**会话**。它们用来管理一组相互关联的进程，意思其实很好理解。

- 进程组表示一组相互关联的进程，比如每个子进程都是父进程所在组的成员；
- 而会话是指共享同一个控制终端的一个或多个进程组。

比如，我们通过 SSH 登录服务器，就会打开一个控制终端（TTY），这个控制终端就对应一个会话。而我们在终端中运行的命令以及它们的子进程，就构成了一个个的进程组，其中，在后台运行的命令，构成后台进程组；在前台运行的命令，构成前台进程组。

明白了这些，我们再用 top 看一下系统的资源使用情况：


```
# 按下数字 1 切换到所有 CPU 的使用情况，观察一会儿按 Ctrl+C 结束
$ top
top - 05:56:23 up 17 days, 16:45, 2 users, load average: 2.00, 1.68, 1.39
Tasks: 247 total, 1 running, 79 sleeping, 0 stopped, 115 zombie
%Cpu0 : 0.0 us, 0.7 sy, 0.0 ni, 38.9 id, 60.5 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.0 us, 0.7 sy, 0.0 ni, 4.7 id, 94.6 wa, 0.0 hi, 0.0 si, 0.0 st
...

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
4340 root 20 0 44676 4048 3432 R 0.3 0.0 0:00.05 top
4345 root 20 0 37280 33624 860 D 0.3 0.0 0:00.01 app
4344 root 20 0 37280 33624 860 D 0.3 0.4 0:00.01 app
1 root 20 0 160072 9416 6752 S 0.0 0.1 0:38.59 systemd
...
```

从这里你能看出什么问题吗？细心一点，逐行观察，别放过任何一个地方。忘了哪行参数意思的话，也要及时返回去复习。

好的，如果你已经有了答案，那就继续往下走，看看跟我找的问题是否一样。这里，我发现了四个可疑的地方。

- 先看第一行的平均负载（ Load Average），过去 1 分钟、5 分钟和 15 分钟内的平均负载在依次减小，说明平均负载正在升高；而 1 分钟内的平均负载已经达到系统的 CPU 个数，说明系统很可能已经有了性能瓶颈。
- 再看第二行的 Tasks，有 1 个正在运行的进程，但僵尸进程比较多，而且还在不停增加，说明有子进程在退出时没被清理。
- 接下来看两个 CPU 的使用率情况，用户 CPU 和系统 CPU 都不高，但 iowait 分别是 60.5% 和 94.6%，好像有点儿不正常。
- 最后再看每个进程的情况， CPU 使用率最高的进程只有 0.3%，看起来并不高；但有两个进程处于 D 状态，它们可能在等待 I/O，但光凭这里并不能确定是它们导致了 iowait 升高。

我们把这四个问题再汇总一下，就可以得到很明确的两点：

- 第一点，iowait 太高了，导致系统的平均负载升高，甚至达到了系统 CPU 的个数。
- 第二点，僵尸进程在不断增多，说明有程序没能正确清理子进程的资源。

那么，碰到这两个问题该怎么办呢？结合我们前面分析问题的思路，你先自己想想，动手试试，下节课我来继续“分解”。

小结

今天我们主要通过简单的操作，熟悉了几个必备的进程状态。用我们最熟悉的 ps 或者 top ，可以查看进程的状态，这些状态包括运行（R）、空闲（I）、不可中断睡眠（D）、可中断睡眠（S）、僵尸（Z）以及暂停（T）等。

其中，不可中断状态和僵尸状态，是我们今天学习的重点。

- 不可中断状态，表示进程正在跟硬件交互，为了保护进程数据和硬件的一致性，系统不允许其他进程或中断打断这个进程。进程长时间处于不可中断状态，通常表示系统有 I/O 性能问题。

- 僵尸进程表示进程已经退出，但它的父进程还没有回收子进程占用的资源。短暂的僵尸状态我们通常不必理会，但进程长时间处于僵尸状态，就应该注意了，可能有应用程序没有正常处理子进程的退出。

思考

最后，我想请你思考一下今天的课后题，案例中发现的这两个问题，你会怎么分析呢？又应该怎么解决呢？你可以结合前面我们做过的案例分析，总结自己的思路，提出自己的问题。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

笔记本： Linux
创建时间： 2019/1/16 9:41 更新时间： 2019/1/16 9:42
标签： 性能优化
URL： https://time.geekbang.org/column/article/71382

08 | 案例篇：系统中出现大量不可中断进程和僵尸进程怎么办？（下）

上一节，我给你讲了 Linux 进程状态的含义，以及不可中断进程和僵尸进程产生的原因，我们先来简单复习下。

使用 ps 或者 top 可以查看进程的状态，这些状态包括运行、空闲、不可中断睡眠、可中断睡眠、僵尸以及暂停等。其中，我们重点学习了不可中断状态和僵尸进程：

- 不可中断状态，一般表示进程正在跟硬件交互，为了保护进程数据与硬件一致，系统不允许其他进程或中断打断该进程。
- 僵尸进程表示进程已经退出，但它的父进程没有回收该进程所占用的资源。

上一节的最后，我用一个案例展示了处于这两种状态的进程。通过分析 top 命令的输出，我们发现了两个问题：

- 第一，iowait 太高了，导致系统平均负载升高，并且已经达到了系统 CPU 的个数。
- 第二，僵尸进程在不断增多，看起来是应用程序没有正确清理子进程的资源。

相信你一定认真思考过这两个问题，那么，真相到底是什么呢？接下来，我们一起顺着这两个问题继续分析，找出根源。

首先，请你打开一个终端，登录到上次的机器中。然后执行下面的命令，重新运行这个案例：

复制代码

```
# 先删除上次启动的案例
$ docker rm -f app
# 重新运行案例
$ docker run --privileged --name=app -itd feisky/app:iowait
```

iowait 分析

我们先来看一下 iowait 升高的问题。

我相信，一提到 iowait 升高，你首先会想要查询系统的 I/O 情况。我一般也是这种思路，那么什么工具可以查询系统的 I/O 情况呢？

这里，我推荐的正是上节课要求安装的 dstat ，它的好处是，可以同时查看 CPU 和 I/O 这两种资源的使用情况，便于对比分析。

那么，我们在终端中运行 dstat 命令，观察 CPU 和 I/O 的使用情况：

复制代码

```
# 间隔 1 秒输出 10 组数据
$ dstat 1 10
You did not select any stats, using -cdngy by default.
--total-cpu-usage-- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai stl| read writ| recv send| in out | int csw
0 0 96 4 0|1219k 408k| 0 0 | 0 0 | 42 885
0 0 2 98 0| 34M 0 | 198B 790B| 0 0 | 42 138
0 0 0 100 0| 34M 0 | 66B 342B| 0 0 | 42 135
0 0 84 16 0|5633k 0 | 66B 342B| 0 0 | 52 177
0 3 39 58 0| 22M 0 | 66B 342B| 0 0 | 43 144
0 0 0 100 0| 34M 0 | 200B 450B| 0 0 | 46 147
0 0 2 98 0| 34M 0 | 66B 342B| 0 0 | 45 134
0 0 0 100 0| 34M 0 | 66B 342B| 0 0 | 39 131
0 0 83 17 0|5633k 0 | 66B 342B| 0 0 | 46 168
0 3 39 59 0| 22M 0 | 66B 342B| 0 0 | 37 134
```

从 dstat 的输出，我们可以看到，每当 iowait 升高（wai）时，磁盘的读请求（read）都会很大。这说明 iowait 的升高跟磁盘的读请求有关，很可能就是磁盘读导致的。

那到底是哪个进程在读磁盘呢？不知道你还记不记得，上节在 top 里看到的不可中断状态进程，我觉得它就很可疑，我们试着来分析下。

我们继续在刚才的终端中，运行 top 命令，观察 D 状态的进程：

复制代码

```
# 观察一会儿按 Ctrl+C 结束
$ top
...
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
4340 root 20 0 44676 4048 3432 R 0.3 0.0 0:00.05 top
4345 root 20 0 37280 33624 860 D 0.3 0.0 0:00.01 app
4344 root 20 0 37280 33624 860 D 0.3 0.4 0:00.01 app
...

```

我们从 top 的输出找到 D 状态进程的 PID，你可以发现，这个界面里有两个 D 状态的进程，PID 分别是 4344 和 4345。

接着，我们查看这些进程的磁盘读写情况。对了，别忘了工具是什么。一般要查看某一个进程的资源使用情况，都可以用我们的老朋友 pidstat，不过这次记得加上 -d 参数，以便输出 I/O 使用情况。

比如，以 4344 为例，我们在终端里运行下面的 pidstat 命令，并用 -p 4344 参数指定进程号：

```
# -d 展示 I/O 统计数据，-p 指定进程号，间隔 1 秒输出 3 组数据
$ pidstat -d -p 4344 1 3
06:38:50 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:38:51 0 4344 0.00 0.00 0.00 0 app
06:38:52 0 4344 0.00 0.00 0.00 0 app
06:38:53 0 4344 0.00 0.00 0.00 0 app
```

在这个输出中， kB_rd 表示每秒读的 KB 数， kB_wr 表示每秒写的 KB 数，iodelay 表示 I/O 的延迟（单位是时钟周期）。它们都是 0，那就表示此时没有任何的读写，说明问题不是 4344 进程导致的。

可是，用同样的方法分析进程 4345，你会发现，它也没有任何磁盘读写。

那要怎么知道，到底是哪个进程在进行磁盘读写呢？我们继续使用 pidstat，但这次去掉进程号，干脆就来观察所有进程的 I/O 使用情况。

在终端中运行下面的 pidstat 命令：

```
# 间隔 1 秒输出多组数据 (这里是 20 组)
$ pidstat -d 1 20
...
06:48:46 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:48:47 0 4615 0.00 0.00 0.00 1 kworker/u4:1
06:48:47 0 6080 32768.00 0.00 0.00 170 app
06:48:47 0 6081 32768.00 0.00 0.00 184 app

06:48:47 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:48:48 0 6080 0.00 0.00 0.00 110 app

06:48:48 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
```

```
06:48:49 0 6081 0.00 0.00 0.00 191 app

06:48:49 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command

06:48:50 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:48:51 0 6082 32768.00 0.00 0.00 0 app
06:48:51 0 6083 32768.00 0.00 0.00 0 app

06:48:51 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:48:52 0 6082 32768.00 0.00 0.00 184 app
06:48:52 0 6083 32768.00 0.00 0.00 175 app

06:48:52 UID PID kB_rd/s kB_wr/s kB_ccwr/s iodelay Command
06:48:53 0 6083 0.00 0.00 0.00 105 app
...
```

观察一会儿可以发现，的确是 app 进程在进行磁盘读，并且每秒读的数据有 32 MB，看来就是 app 的问题。不过，app 进程到底在执行啥 I/O 操作呢？

这里，我们需要回顾一下进程用户态和内核态的区别。进程想要访问磁盘，就必须使用系统调用，所以接下来，重点就是找出 app 进程的系统调用了。

strace 正是最常用的跟踪进程系统调用的工具。所以，我们从 pidstat 的输出中拿到进程的 PID 号，比如 6082，然后在终端中运行 strace 命令，并用 -p 参数指定 PID 号：


```
$ strace -p 6082
strace: attach: ptrace(PTRACE_SEIZE, 6082): Operation not permitted
```

这儿出现了一个奇怪的错误，strace 命令居然失败了，并且命令报出的错误是没有权限。按理来说，我们所有操作都已经是以 root 用户运行了，为什么还会没有权限呢？你也可以先想一下，碰到这种情况，你会怎么处理呢？

一般遇到这种问题时，我会先检查一下进程的状态是否正常。比如，继续在终端中运行 ps 命令，并使用 grep 找出刚才的 6082 号进程：


```
$ ps aux | grep 6082
root 6082 0.0 0.0 0 0 pts/0 Z+ 13:43 0:00 [app] <defunct>
```

果然，进程 6082 已经变成了 Z 状态，也就是僵尸进程。僵尸进程都是已经退出的进程，所以就没法儿继续分析它的系统调用。关于僵尸进程的处理方法，我们一会儿再说，现在还是继续分析 iowait 的问题。

到这一步，你应该注意到了，系统 iowait 的问题还在继续，但是 top、pidstat 这类工具已经不能给出更多的信息了。这时，我们就应该求助那些基于事件记录的动态追踪工具了。

你可以用 perf top 看看有没有新发现。再或者，可以像我一样，在终端中运行 perf record，持续一会儿（例如 15 秒），然后按 Ctrl+C 退出，再运行 perf report 查看报告：


```
$ perf record -g
$ perf report
```

接着，找到我们关注的 app 进程，按回车键展开调用栈，你就会得到下面这张调用关系图：

```
Samples: 143K of event 'cpu-clock', Event count (approx.): 35954750000
  Children      Self  Command          Shared Object               Symbol
+   99.17%     0.00%  swapper          [kernel.vmlinux]            [k] 0x00000000002000d5
+   99.17%     0.00%  swapper          [kernel.vmlinux]            [k] cpu_startup_entry
+   99.17%     0.00%  swapper          [kernel.vmlinux]            [k] do_idle
+   99.15%     0.00%  swapper          [kernel.vmlinux]            [k] default_idle_call
+   99.15%     0.00%  swapper          [kernel.vmlinux]            [k] arch_cpu_idle
+   99.15%     0.00%  swapper          [kernel.vmlinux]            [k] default_idle
+   99.15%    99.11%  swapper          [kernel.vmlinux]            [k] native_safe_halt
+   67.79%     0.00%  swapper          [kernel.vmlinux]            [k] start_secondary
+   31.38%     0.00%  swapper          [kernel.vmlinux].init.text  [k] x86_64_start_kernel
+   31.38%     0.00%  swapper          [kernel.vmlinux].init.text  [k] x86_64_start_reservations
+   31.38%     0.00%  swapper          [kernel.vmlinux].init.text  [k] start_kernel
+   31.38%     0.00%  swapper          [kernel.vmlinux]            [k] rest_init
-    0.66%     0.00%  app              [kernel.vmlinux]            [k] entry_SYSCALL_64
     entry_SYSCALL_64
   - do_syscall_64
      - 0.64% sys_read
           vfs_read
           __vfs_read
           new_sync_read
           blkdev_read_iter
           generic_file_read_iter
         - blkdev_direct_IO
            - 0.57% bio_iov_iter_get_pages
               - iov_iter_get_pages
               + get_user_pages_fast
-    0.66%     0.00%  app              [kernel.vmlinux]            [k] do_syscall_64
   - do_syscall_64
      - 0.64% sys_read
           vfs_read
           __vfs_read
           new_sync_read
           blkdev_read_iter
           generic_file_read_iter
         - blkdev_direct_IO
            + 0.57% bio_iov_iter_get_pages
+    0.65%     0.00%  app              [unknown]                   [k] 0x10be258d4c544155
+    0.65%     0.00%  app              libc-2.27.so                [.] __libc_start_main
```

这个图里的 swapper 是内核中的调度进程，你可以先忽略掉。

我们来看其他信息，你可以发现， app 的确在通过系统调用 sys_read() 读取数据。并且从 new_sync_read 和 blkdev_direct_IO 能看出，进程正在对磁盘进行**直接读**，也就是绕过了系统缓存，每个读请求都会从磁盘直接读，这就可以解释我们观察到的 iowait 升高了。

看来，罪魁祸首是 app 内部进行了磁盘的直接 I/O 啊！

下面的问题就容易解决了。我们接下来应该从代码层面分析，究竟是哪里出现了直接读请求。查看源码文件 app.c，你会发现它果然使用了 O_DIRECT 选项打开磁盘，于是绕过了系统缓存，直接对磁盘进行读写。

复制代码

```
1 open(disk, O_RDONLY|O_DIRECT|O_LARGEFILE, 0755)
```

直接读写磁盘，对 I/O 敏感型应用（比如数据库系统）是很友好的，因为你可以在应用中，直接控制磁盘的读写。但在大部分情况下，我们最好还是通过系统缓存来优化磁盘 I/O，换句话说，删除 O_DIRECT 这个选项就是了。

app-fix1.c 就是修改后的文件，我也打包成了一个镜像文件，运行下面的命令，你就可以启动它了：

```
# 首先删除原来的应用
$ docker rm -f app
# 运行新的应用
$ docker run --privileged --name=app -itd feisky/app:iowait-fix1
```

最后，再用 top 检查一下：

```
$ top
top - 14:59:32 up 19 min, 1 user, load average: 0.15, 0.07, 0.05
Tasks: 137 total, 1 running, 72 sleeping, 0 stopped, 12 zombie
%Cpu0 : 0.0 us, 1.7 sy, 0.0 ni, 98.0 id, 0.3 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.0 us, 1.3 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
...

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3084 root 20 0 0 0 0 Z 1.3 0.0 0:00.04 app
3085 root 20 0 0 0 0 Z 1.3 0.0 0:00.04 app
1 root 20 0 159848 9120 6724 S 0.0 0.1 0:09.03 systemd
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 20 0 0 0 0 I 0.0 0.0 0:00.40 kworker/0:0
...
```

你会发现， iowait 已经非常低了，只有 0.3%，说明刚才的改动已经成功修复了 iowait 高的问题，大功告成！不过，别忘了，僵尸进程还在等着你。仔细观察僵尸进程的数量，你会郁闷地发现，僵尸进程还在不断的增长中。

僵尸进程

接下来，我们就来处理僵尸进程的问题。既然僵尸进程是因为父进程没有回收子进程的资源而出现的，那么，要解决掉它们，就要找到它们的根儿，**也就是找出父进程，然后在父进程里解决。**

父进程的找法我们前面讲过，最简单的就是运行 pstree 命令：

```
# -a 表示输出命令行选项
# p 表 PID
# s 表示指定进程的父进程
$ pstree -aps 3084
systemd,1
  └─dockerd,15006 -H fd://
      └─docker-containe,15024 --config /var/run/docker/containerd/containerd.toml
          └─docker-containe,3991 -namespace moby -workdir...
              └─app,4009
                  └─(app,3084)
```

运行完，你会发现 3084 号进程的父进程是 4009，也就是 app 应用。

所以，我们接着查看 app 应用程序的代码，看看子进程结束的处理是否正确，比如有没有调用 wait() 或 waitpid()，抑或是，有没有注册 SIGCHLD 信号的处理函数。

现在我们查看修复 iowait 后的源码文件 app-fix1.c ，找到子进程的创建和清理的地方：

```
int status = 0;
for (;;) {
for (int i = 0; i < 2; i++) {
if(fork()== 0) {
sub_process();
}
}
sleep(5);
}

while(wait(&status)>0);
```

循环语句本来就容易出错，你能找到这里的问题吗？这段代码虽然看起来调用了 wait() 函数等待子进程结束，但却错误地把 wait() 放到了 for 死循环的外面，也就是说，wait() 函数实际上并没被调用到，我们把它挪到 for 循环的里面就可以了。

修改后的文件我放到了 app-fix2.c 中，也打包成了一个 Docker 镜像，运行下面的命令，你就可以启动它：

```
# 先停止产生僵尸进程的 app
$ docker rm -f app
```

```
# 然后启动新的 app
$ docker run --privileged --name=app -itd feisky/app:iowait-fix2
```

启动后，再用 top 最后来检查一遍：

复制代码

```
$ top
top - 15:00:44 up 20 min, 1 user, load average: 0.05, 0.05, 0.04
Tasks: 125 total, 1 running, 72 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.0 us, 1.7 sy, 0.0 ni, 98.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.0 us, 1.3 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
...

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
3198 root 20 0 4376 840 780 S 0.3 0.0 0:00.01 app
2 root 20 0 0 0 0 S 0.0 0.0 0:00.00 kthreadd
3 root 20 0 0 0 0 I 0.0 0.0 0:00.41 kworker/0:0
...
```

好了，僵尸进程（Z 状态）没有了， iowait 也是 0，问题终于全部解决了。

小结

今天我用一个多进程的案例，带你分析系统等待 I/O 的 CPU 使用率（也就是 iowait%）升高的情况。

虽然这个案例是磁盘 I/O 导致了 iowait 升高，不过， **iowait 高不一定代表 I/O 有性能瓶颈。当系统中只有 I/O 类型的进程在运行时，iowait 也会很高，但实际上，磁盘的读写远没有达到性能瓶颈的程度。**

因此，碰到 iowait 升高时，需要先用 dstat、pidstat 等工具，确认是不是磁盘 I/O 的问题，然后再找是哪些进程导致了 I/O。

等待 I/O 的进程一般是不可中断状态，所以用 ps 命令找到的 D 状态（即不可中断状态）的进程，多为可疑进程。但这个案例中，在 I/O 操作后，进程又变成了僵尸进程，所以不能用 strace 直接分析这个进程的系统调用。

这种情况下，我们用了 perf 工具，来分析系统的 CPU 时钟事件，最终发现是直接 I/O 导致的问题。这时，再检查源码中对应位置的问题，就很轻松了。

而僵尸进程的问题相对容易排查，使用 pstree 找出父进程后，去查看父进程的代码，检查 wait() / waitpid() 的调用，或是 SIGCHLD 信号处理函数的注册就行了。

思考

最后，我想邀请你一起来聊聊，你碰到过的不可中断状态进程和僵尸进程问题。你是怎么分析它们的根源？又是怎么解决的？在今天的案例操作中，你又有什么新的发现吗？你可以结合我的讲述，总结自己的思路。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。

笔记本： Linux
创建时间： 2019/1/13 21:49 更新时间： 2019/1/16 9:36
标签： 性能优化
URL： https://time.geekbang.org/column/article/71868

09 | 基础篇：怎么理解Linux软中断?

上一期，我用一个不可中断进程的案例，带你学习了 iowait（也就是等待 I/O 的 CPU 使用率）升高时的分析方法。这里你要记住，进程的不可中断状态是系统的一种保护机制，可以保证硬件的交互过程不被意外打断。所以，短时间的不可中断状态是很正常的。

但是，当进程长时间都处于不可中断状态时，你就得当心了。这时，你可以使用 dstat、pidstat 等工具，确认是不是磁盘 I/O 的问题，进而排查相关的进程和磁盘设备。关于磁盘 I/O 的性能问题，你暂且不用专门去背，我会在后续的 I/O 部分详细介绍，到时候理解了也就记住了。

其实除了 iowait，软中断（softirq）CPU 使用率升高也是最常见的一种性能问题。接下来的两节课，我们就来学习软中断的内容，我还会以最常见的反向代理服务器 Nginx 的案例，带你分析这种情况。

从“取外卖”看中断

说到中断，我在前面关于“上下文切换”的文章，简单说过中断的含义，先来回顾一下。中断是系统用来响应硬件设备请求的一种机制，它会打断进程的正常调度和执行，然后调用内核中的中断处理程序来响应设备的请求。

你可能要问了，为什么要有中断呢？我可以举个生活中的例子，让你感受一下中断的魅力。

比如说你订了一份外卖，但是不确定外卖什么时候送到，也没有别的方法了解外卖的进度，但是，配送员送外卖是不等人的，到了你这儿没人取的话，就直接走人了。所以你只能苦苦等着，时不时去门口看看外卖送到没，而不能干其他事情。

不过呢，如果在订外卖的时候，你就跟配送员约定好，让他送到后给你打个电话，那你就不用苦苦等待了，就可以去忙别的事情，直到电话一响，接电话、取外卖就可以了。

这里的“打电话”，其实就是一个中断。没接到电话的时候，你可以做其他的事情；只有接到了电话（也就是发生中断），你才要进行另一个动作：取外卖。

这个例子你就可以发现，**中断其实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。**

由于中断处理程序会打断其他进程的运行，所以，**为了减少对正常进程运行调度的影响，中断处理程序就需要尽可能快地运行**。如果中断本身要做的事情不多，那么处理起来也不会有太大问题；但如果中断要处理的事情很多，中断服务程序就有可能要运行很长时间。

特别是，中断处理程序在响应中断时，还会临时关闭中断。这就会导致上一次中断处理完成之前，其他中断都不能响应，也就是说中断有可能会丢失。

那么还是以取外卖为例。假如你订了 2 份外卖，一份主食和一份饮料，并且是由 2 个不同的配送员来配送。这次你不用时时等待着，两份外卖都约定了电话取外卖的方式。但是，问题又来了。

当第一份外卖送到时，配送员给你打了个长长的电话，商量发票的处理方式。与此同时，第二个配送员也到了，也想给你打电话。

但是很明显，因为电话占线（也就是关闭了中断响应），第二个配送员的电话是打不通的。所以，第二个配送员很可能试几次后就走掉了（也就是丢失了一次中断）。

软中断

如果你弄清楚了“取外卖”的模式，那对系统的中断机制就很容易理解了。事实上，为了解决中断处理程序执行过长和中断丢失的问题，Linux 将中断处理过程分成了两个阶段，也就是**上半部和下半部**：

- **上半部用来快速处理中断**，它在中断禁止模式下运行，主要处理跟硬件紧密相关的或时间敏感的工作。
- **下半部用来延迟处理上半部未完成的工作，通常以内核线程的方式运行**。

比如说前面取外卖的例子，上半部就是你接听电话，告诉配送员你已经知道了，其他事儿见面再说，然后电话就可以挂断了；下半部才是取外卖的动作，以及见面后商量发票处理的动作。

这样，第一个配送员不会占用你太多时间，当第二个配送员过来时，照样能正常打通你的电话。

除了取外卖，我再举个最常见的网卡接收数据包的例子，让你更好地理解。

网卡接收到数据包后，会通过**硬件中断**的方式，通知内核有新的数据到了。这时，内核就应该调用中断处理程序来响应它。你可以自己先想一下，这种情况下的上半部和下半部分别负责什么工作呢？

对上半部来说，既然是快速处理，其实就是要把网卡的数据读到内存中，然后更新一下硬件寄存器的状态（表示数据已经读好了），最后再发送一个**软中断**信号，通知下半部做进一步的处理。

而下半部被软中断信号唤醒后，需要从内存中找到网络数据，再按照网络协议栈，对数据进行逐层解析和处理，直到把它送给应用程序。

所以，这两个阶段你也可以这样理解：

- 上半部直接处理硬件请求，也就是我们常说的硬中断，特点是快速执行；
- 而下半部则是由内核触发，也就是我们常说的软中断，特点是延迟执行。

实际上，上半部会打断 CPU 正在执行的任务，然后立即执行中断处理程序。而下半部以内核线程的方式执行，并且每个 CPU 都对应一个软中断内核线程，名字为 “ksoftirqd/CPU 编号”，比如说， 0 号 CPU 对应的软中断内核线程的名字就是 ksoftirqd/0。

不过要注意的是，软中断不只包括了刚刚所讲的硬件设备中断处理程序的下半部，一些内核自定义的事件也属于软中断，比如内核调度和 RCU 锁（Read-Copy Update 的缩写，RCU 是 Linux 内核中最常用的锁之一）等。

那要怎么知道你的系统里有哪些软中断呢？

查看软中断和内核线程

不知道你还记不记得，前面提到过的 proc 文件系统。它是一种内核空间和用户空间进行通信的机制，可以用来查看内核的数据结构，或者用来动态修改内核的配置。其中：

- /proc/softirqs 提供了软中断的运行情况；
- /proc/interrupts 提供了硬中断的运行情况。

运行下面的命令，查看 /proc/softirqs 文件的内容，你就可以看到各种类型软中断在不同 CPU 上的累积运行次数：


```
$ cat /proc/softirqs
CPU0 CPU1
HI: 0 0
TIMER: 811613 1972736
NET_TX: 49 7
NET_RX: 1136736 1506885
BLOCK: 0 0
IRQ_POLL: 0 0
TASKLET: 304787 3691
SCHED: 689718 1897539
HRTIMER: 0 0
```

```
RCU: 1330771 1354737
```

在查看 /proc/softirqs 文件内容时，你要特别注意以下这两点。

第一，要注意软中断的类型，也就是这个界面中第一列的内容。从第一列你可以看到，软中断包括了 10 个类别，分别对应不同的工作类型。比如 NET_RX 表示网络接收中断，而 NET_TX 表示网络发送中断。

第二，要注意同一种软中断在不同 CPU 上的分布情况，也就是同一行的内容。正常情况下，同一种中断在不同 CPU 上的累积次数应该差不多。比如这个界面中，NET_RX 在 CPU0 和 CPU1 上的中断次数基本是同一个数量级，相差不大。

不过你可能发现，TASKLET 在不同 CPU 上的分布并不均匀。TASKLET 是最常用的软中断实现机制，每个 TASKLET 只运行一次就会结束 ，并且只在调用它的函数所在的 CPU 上运行。

因此，使用 TASKLET 特别简便，当然也会存在一些问题，比如说由于只在一个 CPU 上运行导致的调度不均衡，再比如因为不能在多个 CPU 上并行运行带来了性能限制。

另外，刚刚提到过，软中断实际上是以内核线程的方式运行的，每个 CPU 都对应一个软中断内核线程，这个软中断内核线程就叫做 ksoftirqd/CPU 编号。那要怎么查看这些线程的运行状况呢?

其实用 ps 命令就可以做到，比如执行下面的指令：


```
$ ps aux | grep softirq
root 7 0.0 0.0 0 0 ? S Oct10 0:01 [ksoftirqd/0]
root 16 0.0 0.0 0 0 ? S Oct10 0:01 [ksoftirqd/1]
```

注意，这些线程的名字外面都有中括号，这说明 ps 无法获取它们的命令行参数（cmline）。一般来说，ps 的输出中，名字括在中括号里的，一般都是内核线程。

小结

Linux 中的中断处理程序分为上半部和下半部：

- 上半部对应硬件中断，用来快速处理中断。
- 下半部对应软中断，用来异步处理上半部未完成的工作。

Linux 中的软中断包括网络收发、定时、调度、RCU 锁等各种类型，可以通过查看 /proc/softirqs 来观察软中断的运行情况。

思考

最后，我想请你一起聊聊，你是怎么理解软中断的？你有没有碰到过因为软中断出现的性能问题？你又是怎么分析它们的瓶颈的呢？你可以结合今天的内容，总结自己的思路，写下自己的问题。

欢迎在留言区和我讨论，也欢迎把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。