03 | 基础篇：经常说的 CPU 上下文切换是什么意思？（上）

2018-11-26 倪朋飞

03 | 基础篇：经常说的 CPU 上下文切换是什么意思？（上）
朗读人：冯永吉　12′38″ | 5.80M

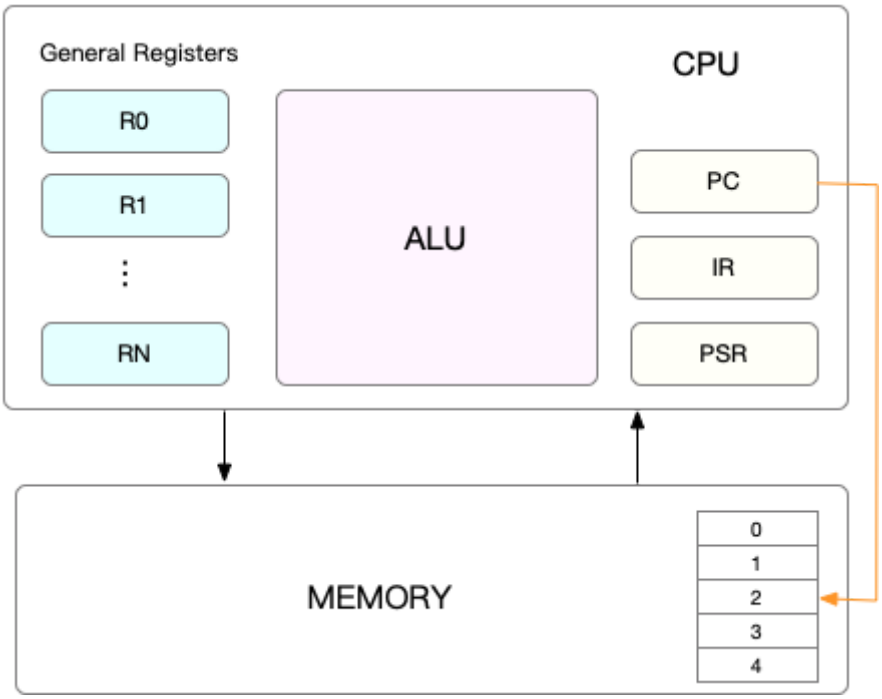
你好，我是倪朋飞。

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。
多个进程竞争CPU时，虽然竞争的CPU没有真正运行，但是会导致CPU上下文频繁切换，从而导致负载升高。
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**。 CPU上下文包括CPU寄存器和程序计数器



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。

CPU的上下文切换分为几个场景，进程上下文切换，线程上下文切换，中断上下文切换。

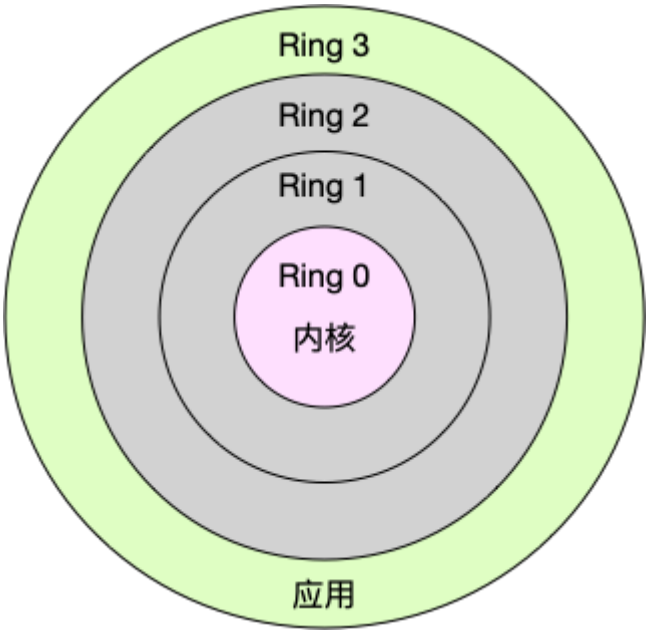
所以，根据任务的不同，CPU 的上下文切换就可以分为几个不同的场景，也就是**进程上下文切换**、**线程上下文切换**以及**中断上下文切换**。

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。

系统调用过程通常称为特权模式切换，而不是上下文切换。

那么，进程上下文切换跟系统调用又有什么区别呢？

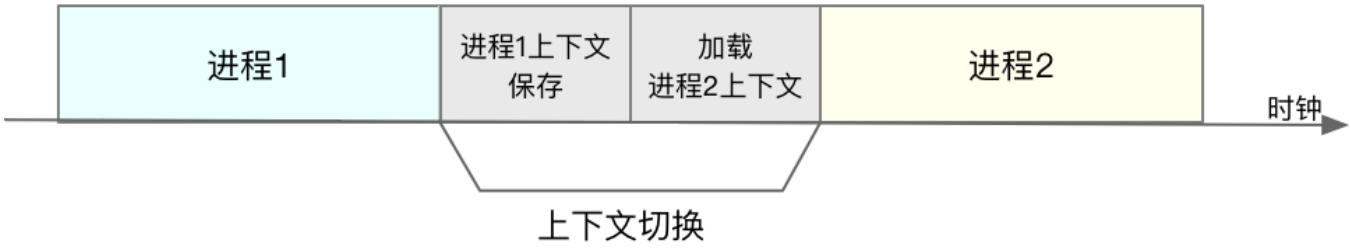
进程是由内核来管理和调度的，进程的切换只能发生在内核态。

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。

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。

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。



每次上下文切换都需要几十纳秒到数微秒的CPU时间。

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妙的 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。

Linux通过TLB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，内存的访问也随之变慢。

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。 进程切换场景:时间片轮询，系统资源不足，sleep函数挂起，更高的进程运行，硬件中断。

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。

线程是调度的基本单位，而进程则是资源拥有的基本单位。

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。

中断上下文切换并不涉及到进程的用户态。

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。

上一篇 02 | 基础篇：到底应该怎么理解“平均负载”？

写留言

精选留言

MoFanDon 2

以前的确是没怎么注意 CPU上下文切换，进程上下文切换 是不同的。学习了。

2018-11-26

C家族的忠实粉儿 1

『D4打卡』

今天学了进程上下文切换、线程上下文切换、中断上下文切换。进程上下文切换和系统调用的区别。
另外，刚刚发现，原来进程、线程我一直都不是特别清楚，尴尬，白学白用了这么久的感觉

2018-11-26

王崧霁 0

进程是资源分配的最小单位，线程是任务调度执行的最小单位，每个进程都有一个主线程，从这个角度说是不是只有线程的上下文切换，而所谓进程上下文切换实则是进程之间获取地址空间等资源的系统调用; 中断分硬中断和软中断，他们的上下文切换又有什么区别哪?

2018-11-26

王涛

D3打卡

2018-11-26

0

骑车吃火锅

如果是上下文切换比较吃CPU，那么我理解一秒内，两个线程之间互相切换与几百个甚至上千个线程互相切换的对系统的开销应该是一样的?

2018-11-26

0

Linuxer

进程上下文切换包括，进程之间上下文切换，线程之间上下文切换，中断上下文切换。其中跨进程的线程上下文切换等同进程上下文切换，前面关于cpu寄存器，全局变量等需不需要保存都好理解，只是中断上下文切换为什么不需要保存虚拟内存和全局变量等?

2018-11-26

0

Anker

Linux是多用户多任务操作系统，cpu是共享资源，多进程或线程根据操作系统不同的调度策略轮徇是使用CPU，下一次执行必须要能够知道上次执行到哪里了，因此需要CPU内部存储需要保存每个进程或线程的状态。

2018-11-26

0

Amark

打卡

2018-11-26

0

马殿军

好👌

2018-11-26

0

小球就是饭二妞🐷🐷

有些概念是不是只是Linux 有的啊，用户态神马的

2018-11-26

0

solar

上下文切换一直拥有概念，但是没有理解的这么细，受教了~~

2018-11-26

0

耿长学

老师，推荐基本性能相关的书籍吧

2018-11-26

0

Cloud*

mark

0

2018-11-26

湖湘志

0

沙发D3

2018-11-26