01丨程序运行原理：程序是如何运行又是如何崩溃的?

2019-11-18 李智慧

讲述：李智慧
时长 12:54 大小 17.73M

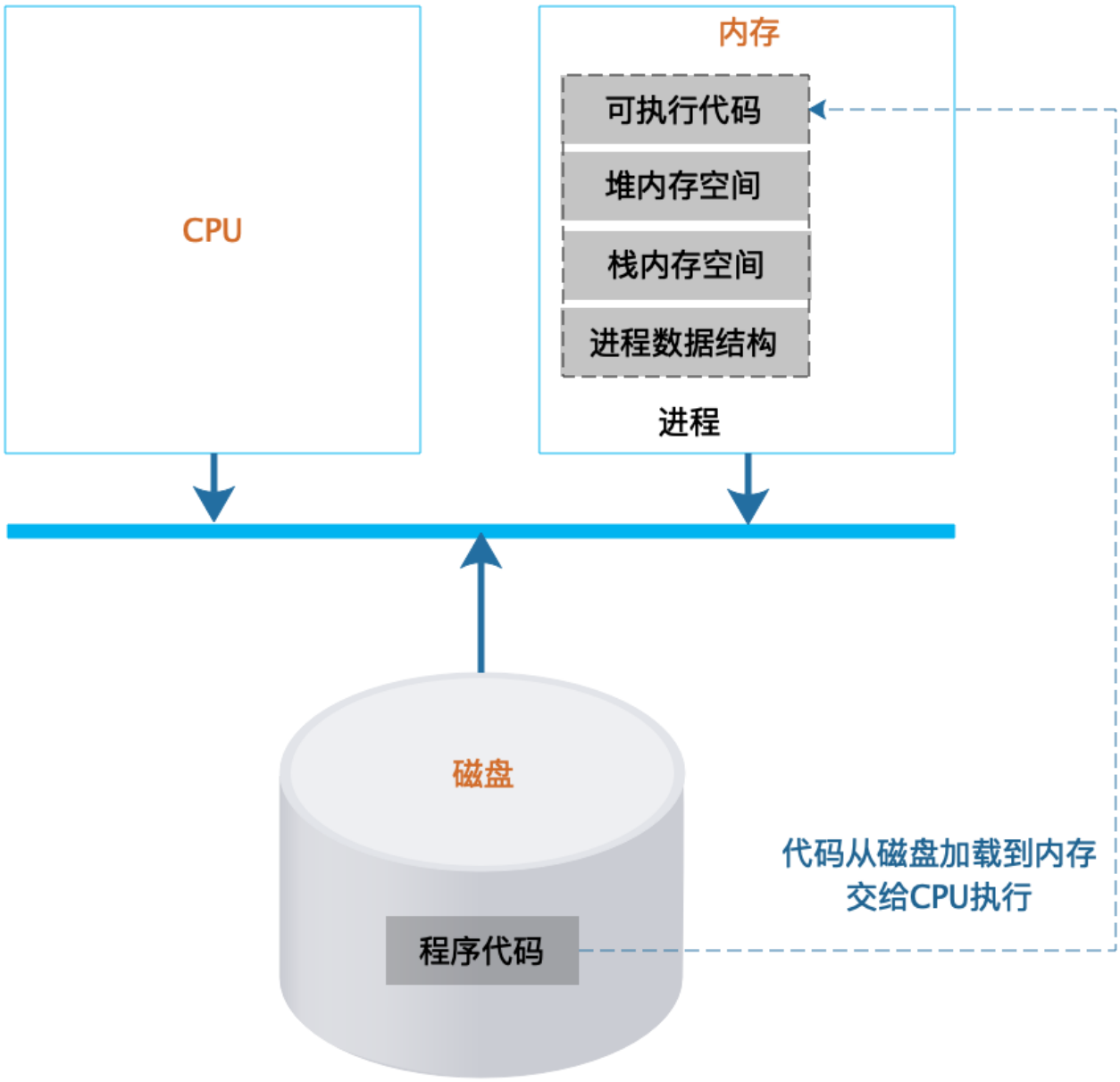
软件的核心载体是程序代码，软件开发的主要工作产出也是代码，但是代码被存储在磁盘上本身没有任何价值，软件要想实现价值，代码就必须运行起来。那么代码是如何运行的？在运行中可能会出现什么问题呢?

程序是如何运行起来的

软件被开发出来，是文本格式的代码，这些代码通常不能直接运行，需要使用编译器编译成操作系统或者虚拟机可以运行的代码，即可执行代码，它们都被存储在文件系统中。不管是文本格式的代码还是可执行的代码，都被称为**程序**，程序是静态的，安静地呆在磁盘上，什么也干不了。要想让程序处理数据，完成计算任务，必须把程序从外部设备加载到内存中，

并在操作系统的管理调度下交给 CPU 去执行，去运行起来，才能真正发挥软件的作用，程序运行起来以后，被称作**进程**。

进程除了包含可执行的程序代码，还包括进程在运行期使用的内存堆空间、栈空间、供操作系统管理用的数据结构。如下图所示：



操作系统把可执行代码加载到内存中，生成相应的数据结构和内存空间后，就从可执行代码的起始位置读取指令交给 CPU 顺序执行。指令执行过程中，可能会遇到一条跳转指令，即 CPU 要执行的下一条指令不是内存中可执行代码顺序的下一条指令。编程中使用的循环 for…，while…和 if…else…最后都被编译成跳转指令。

程序运行时如果需要创建数组等数据结构，操作系统就会在进程的**堆空间**申请一块相应的内存空间，并把这块内存的首地址信息记录在进程的栈中。堆是一块无序的内存空间，任何时候进程需要申请内存，都会从堆空间中分配，分配到的内存地址则记录在栈中。

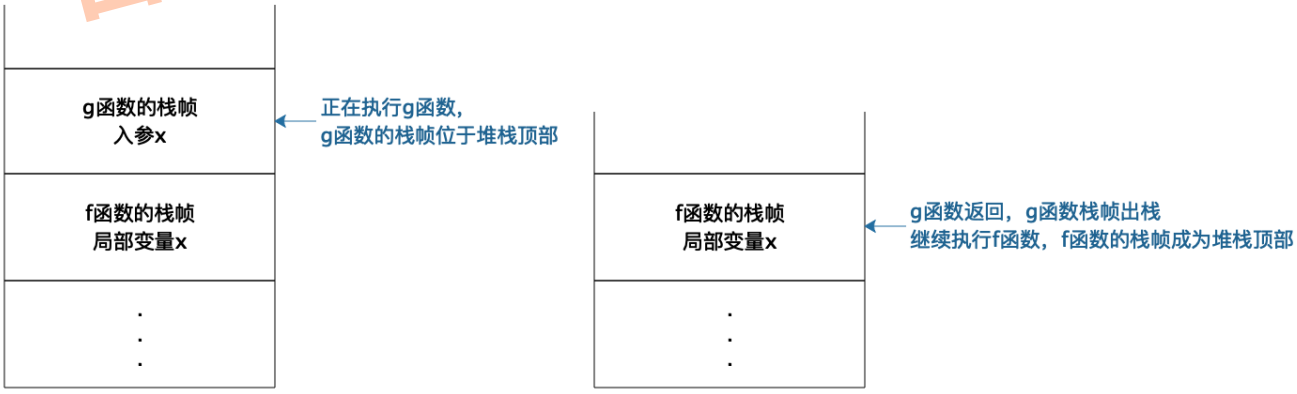
栈是严格的一个后进先出的数据结构，同样由操作系统维护，主要用来记录函数内部的局部变量、堆空间分配的内存空间地址等。

我们以如下代码示例，描述函数调用过程中，栈的操作过程：

复制代码

```
1 void f(){
2   int x = g(1);
3   x++; //g 函数返回，当前堆栈顶部为 f 函数栈帧，在当前栈帧继续执行 f 函数的代码。
4 }
5 int g(int x){
6   return x + 1;
7 }
```


每次函数调用，操作系统都会在栈中创建一个栈帧（stack frame）。正在执行的函数参数、局部变量、申请的内存地址等都在当前栈帧中，也就是堆栈的顶部栈帧中。如下图所示：



当 f 函数执行的时候，f 函数就在栈顶，栈帧中存储着 f 函数的局部变量，输入参数等等。当 f 函数调用 g 函数，当前执行函数就变成 g 函数，操作系统会为 g 函数创建一个栈帧并放置在栈顶。当函数 g() 调用结束，程序返回 f 函数，g 函数对应的栈帧出栈，顶部栈帧变又为 f 函数，继续执行 f 函数的代码，也就是说，真正执行的函数永远都在栈顶。而且因为栈帧是隔离的，所以不同函数可以定义相同的变量而不会发生混乱。

一台计算机如何同时处理数以百计的任务

我们自己日常使用的 PC 计算机通常只是一核或者两核的 CPU，我们部署应用程序的服务器虽然有更多的 CPU 核心，通常也不过几核或者几十核。但是我们的 PC 计算机可以同时编程、听音乐，而且还能执行下载任务，而服务器则可以同时处理数以百计甚至数以千计的**并发**用户请求。

那么为什么一台计算机服务器可以同时处理数以百计，以千计的计算任务呢？这里主要依靠的是操作系统的 CPU 分时共享技术。如果同时有很多个进程在执行，操作系统会将 CPU 的执行时间分成很多份，进程按照某种策略轮流在 CPU 上运行。由于现代 CPU 的计算能力非常强大，虽然每个进程都只被执行了很短一个时间，但是在外部看来却好像是所有的进程都在同时执行，每个进程似乎都独占一个 CPU 执行。

所以虽然从外部看起来，多个进程在同时运行，但是在实际物理上，进程并不总是在 CPU 上运行的，一方面进程共享 CPU，所以需要等待 CPU 运行，另一方面，进程在执行 I/O 操作的时候，也不需要 CPU 运行。进程在生命周期中，主要有三种状态，运行、就绪、阻塞。

运行：当一个进程在 CPU 上运行时，则称该进程处于运行状态。处于运行状态的进程的数目小于等于 CPU 的数目。

就绪：当一个进程获得了除 CPU 以外的一切所需资源，只要得到 CPU 即可运行，则称此进程处于就绪状态，就绪状态有时候也被称为等待运行状态。

阻塞：也称为等待或睡眠状态，当一个进程正在等待某一事件发生（例如等待 I/O 完成，等待锁……）而暂时停止运行，这时即使把 CPU 分配给进程也无法运行，故称该进程处于阻塞状态。

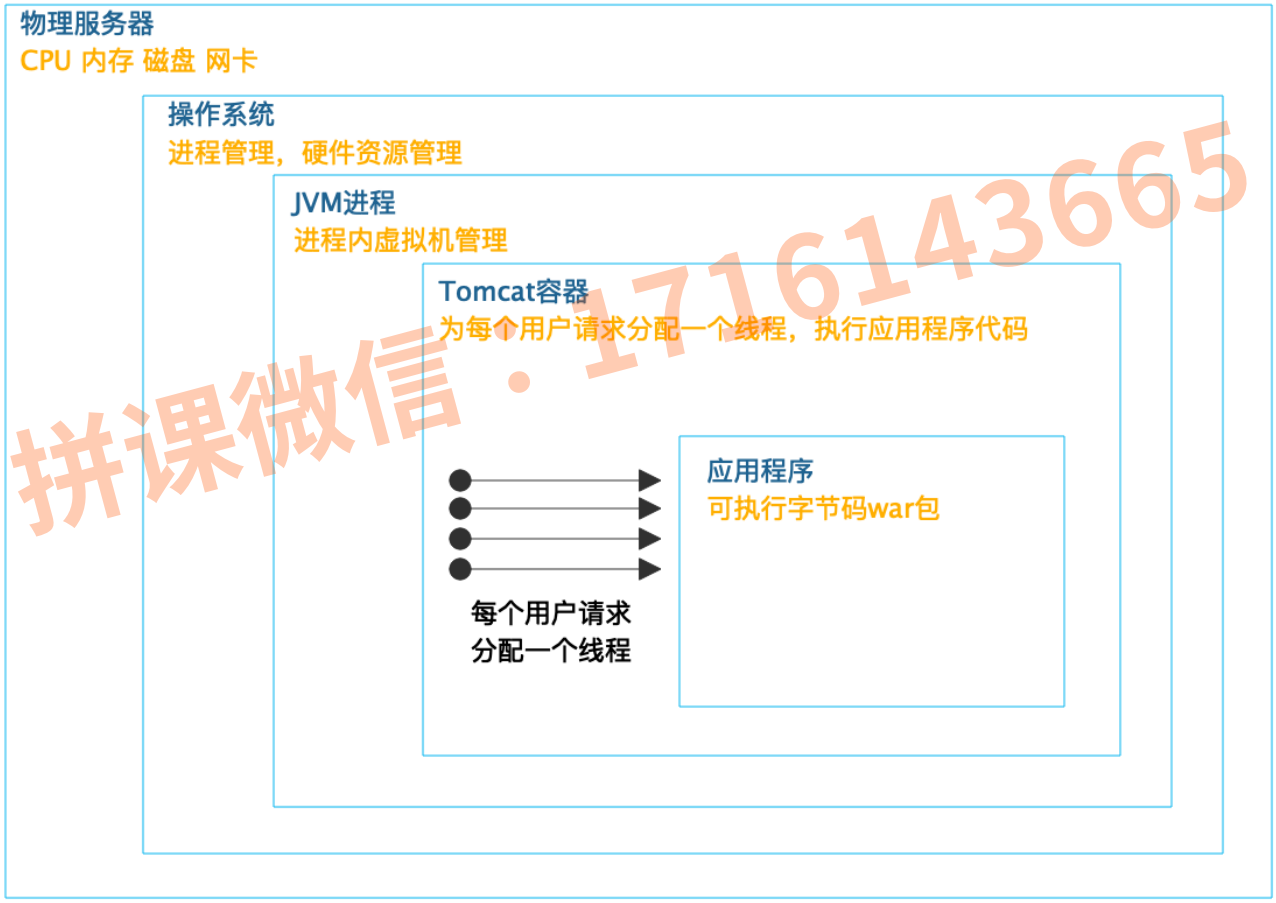
不同进程轮流在 CPU 上执行，每次都要进行进程间 CPU 切换，代价是非常大的，实际上，每个用户请求对应的不是一个进程，而是一个线程。线程可以理解为轻量级的进程，在进程内创建，拥有自己的线程栈，在 CPU 上进行线程切换的代价也更小。线程在运行时，和进程一样，也有三种主要状态，从逻辑上看，进程的主要概念都可以套用到线程上。我们在进行服务器应用开发的时候，通常都是多线程开发，理解线程对我们设计、开发软件更有价值。

系统为什么会变慢，为什么会崩溃

现在的服务器软件系统主要使用多线程技术实现多任务处理，完成对很多用户的并发请求处理。也就是我们开发的应用程序通常以一个进程的方式在操作系统中启动，然后在进程中创建很多线程，每个线程处理一个用户请求。

以 Java 的 web 开发为例，似乎我们编程的时候通常并不需要自己创建和启动线程，那么我们的程序是如何被多线程并发执行，同时处理多个用户请求的呢？实际中，启动多线程，为每个用户请求分配一个处理线程的工作是在 web 容器中完成的，比如常用的 Tomcat 容器。

如下图所示：



Tomcat 启动多个线程，为每个用户请求分配一个线程，调用和请求 URL 路径相对应的 Servlet（或者 Controller）代码，完成用户请求处理。而 Tomcat 则在 JVM 虚拟机进程中，JVM 虚拟机则被操作系统当做一个独立进程管理。真正完成最终计算的，是 CPU、内存等服务器硬件，操作系统将这些硬件进行分时（CPU）、分片（内存）管理，虚拟化成一个独享资源让 JVM 进程在其上运行。

以上就是一个 Java web 应用运行时的主要**架构**，有时也被称作**架构过程视图**。需要注意的是，这里有个很多 web 开发者容易忽略的事情，那就是**不管你是否有意识，你开发的 web 程序都是被多线程执行的，web 开发天然就是多线程开发**。

CPU 以线程为单位进行分时共享执行，可以想象代码被加载到内存空间后，有多个线程在这些代码上执行，这些线程从逻辑上看，是同时在运行的，每个线程有自己的线程栈，所有的线程栈都是完全隔离的，也就是每个方法的参数和方法内的局部变量都是隔离的，一个线程无法访问到其他线程的栈内数据。

但是当某些代码修改内存堆里的数据的时候，如果有多个线程在同时执行，就可能会出现同时修改数据的情况，比如，两个线程同时对一个堆中的数据执行 +1 操作，最终这个数据只会被加一次，这就是人们常说的**线程安全**问题，实际上线程的结果应该是依次加一，即最终的结果应该是 +2。

多个线程访问共享资源的这段代码被称为**临界区**，解决线程安全问题的主要方法是使用锁，将临界区的代码加锁，只有获得锁的线程才能执行临界区代码，如下：

复制代码

```
lock.lock();  // 线程获得锁
i++;  // 临界区代码，i 位于堆中
lock.unlock();  // 线程释放锁
```

如果当前线程执行到第一行，获得锁的代码的时候，锁已经被其他线程获取并没有释放，那么这个线程就会进入阻塞状态，等待前面释放锁的线程将自己唤醒重新获得锁。

锁会引起线程阻塞，如果有很多线程同时在运行，那么就会出现线程排队等待锁的情况，线程无法并行执行，系统响应速度就会变慢。此外 I/O 操作也会引起阻塞，对数据库连接的获取也可能会引起阻塞。目前典型的 web 应用都是基于 RDBMS 关系数据库的，web 应用要想访问数据库，必须获得数据库连接，而受数据库资源限制，每个 web 应用能建立的数据库的连接是有限的，如果并发线程数超过了连接数，那么就会有部分线程无法获得连接而进入阻塞，等待其他线程释放连接后才能访问数据库，并发的线程数越多，等待连接的时间也越多，从 web 请求者角度看，响应时间变长，**系统变慢**。

被阻塞的线程越多，占据的系统资源也越多，这些被阻塞的线程既不能继续执行，也不能释放当前已经占据的资源，在系统中一边等待一边消耗资源，如果阻塞的线程数超过了某个系统资源的极限，就会导致系统宕机，**应用崩溃**。

解决系统因高并发而导致的响应变慢、应用崩溃的主要手段是使用**分布式系统架构**，用更多的服务器构成一个集群，以便共同处理用户的并发请求，保证每台服务器的并发负载不会太高。此外必要时还需要在请求入口处进行**限流**，减小系统的并发请求数；在应用内进行业务**降级**，减小线程的资源消耗。高并发系统架构方案将在专栏的第三模块中进一步探讨。

小结

事实上，现代 CPU 和操作系统的设计远比这篇文章讲的要复杂得多，但是基础原理大致就是如此。为了让程序能很好地被执行，软件开发的时候要考虑很多情况，为了让软件能更好地发挥效能，需要在部署上进行规划和架构。软件是如何运行的，应该是软件工程师和架构师的常识，在设计开发软件的时候，应该时刻以常识去审视自己的工作，保证软件开发在正确的方向上前进。

思考题

线程安全的临界区需要依靠锁，而锁的获取必须也要保证自己是线程安全的，也就是说，不能出现两个线程同时得到锁的情况，那么锁是如何保证自己是线程安全的呢？或者说，在操作系统以及 CPU 层面，锁是如何实现的？

你不妨思考一下这个问题，把你的思考写在下面的评论区里，我会和你一起交流。也欢迎你把这篇文章分享给你的朋友或者同事，一起交流一下。

上一篇　开篇词 | 掌握软件开发技术的第一性原理

下一篇　02 | 数据结构原理：Hash表的时间复杂度为什么是O(1)?

精选留言 (24)

写留言

斐波那契
2019-11-18

在java里 锁是通过cas把当前线程id刷新到对象的头信息里 在获取锁时会去头信息里拿这个信息 如果没有 则会cas刷新进去 刷新成功就获取到锁 刷新失败就表明有别的线程也在尝试刷新这个信息 在操作系统层面 有pv操作保证原子性 而pv操作也是利用cpu中原语指令在获取锁时保证不会被别的指令打断（或被重排序）

展开

作者回复: 👍

2　24

Luciano李鑫
2019-11-18

“不管你是否有意识，你开发的 web 程序都是被多线程执行的，web 开发天然就是多线程开发。”这个会不会绝对了一些，比如go或者c++开发的没有额外创建线程的web程序呢?

作者回复: 这样开发出来的也就不会是一个可以满足高并发的web应用😁 也就不会有我们标题提出的问题~

这个专栏的大部分文章都试图在一篇较小的篇幅内讲完整一个较大的问题场景，这样就会有一些绝对化和理想化，后面的文章也会有类似的情况😢

这也真是我在开篇词想要表达的意图，我们在构建自己的知识体系时，不追求细节的完整和绝对的严谨，而是在大的方向上建起支撑性的知识支柱。细节的缺失和瑕疵在未来的学习和实践中完善。

当然也欢迎大家在评论中指出各种问题，有兴趣的同学参与讨论，我们在评论区就完成部分知识的深入和完善。

感谢👍

4 11

Allen_
2019-11-18

有些地方可能不是很到位，欢迎补充
小结:
1. 我们平时开发出来的程序是文本格式代码，但只是在硬盘中还只是一个程序，只有加载到内存里面通过cpu执行成为进程才是发挥了程序作用。
2.进程里面有堆，栈，可执行代码和进程数据结构。...
展开

作者回复: 清晰👍

2 8

雷咏
2019-11-19

我们用文本格式书写的程序有三种执行方式:
1.解释执行。例子是脚本语言书写的程序或类似于BASIC语言书写的程序。著名的PYTHON也属于这种情况。
2.编译执行。通常C/C++程序属于这种情况。文本格式书写的程序称之为源程序，需要编

译器编译成机器语言代码，称之为可执行程序一或目标程序。...

展开

作者回复: 👍

5

探索无止境

2019-11-19

希望老师在第二节课可以谈谈上一节课留下的思考题，您是怎么理解分析的

作者回复: 也许可以将来以彩蛋的方式专门写一篇文章讲讲~

3

y欧尼酱

2019-11-19

01 程序运行原理

程序是如何运行起来的?

- 1 、程序员被开发出来，文本格式代码，不能直接运行，需要编译器或者虚拟机先编译...

展开

2

2

小美

2019-11-18

李老师好：我想请教下，一个JVM 是一个进程。JVM 上跑 tomcat，tomcat 可以部署多个应用？那每个跑在tomcat 上的应用是一个线程吗？那一个应用crash了，其他应用也会crash.这块感觉有点问题。不知道老师方便解释下吗?

展开

作者回复: JVM是一个进程；tomcat是一个框架；tomcat会启动线程处理应用请求，执行应用的代码；应用自己不能跑起来的，只能被tomcat的线程执行，应用在tomcat线程中执行时，也可以自己启动线程，并发、异步执行自己的某些计算。

tomcat会对不同应用做一些隔离，但是如果某个应用导致JVM crash，所有应用都crash。

可参考后续专栏《JVM虚拟机原理》

1 2

Citizen Z
2019-11-20

粗略思考：
1. 锁在内存中只有 1 份数据（JVM 用堆区中对象的 header 来实现，所以是每个线程都可见的），有开/关两个状态
2. 不同线程访问锁时一定是有先后顺序的（JMM 的 happens-before 原则有规定，这个可能涉及较多手段来保证，问题难点主要出现在多核 CPU 和 CPU cache 的情况下）...
展开

1

Tobe24
2019-11-20

您好，老师，这节课太适合我这种新手了，这里有三个不太明白的地方，希望老师能够解惑。
问题1：CPU 分时共享技术同时执行进程的数量，取决于什么？
问题2：为什么线程切换的代价更小？
问题3：进程切换是不是必须要等到线程切换完毕后进行？如果不是，优先级是由什么决...
展开

作者回复: 1 取决于CPU的核数
2 线程比进程占据的资源少，切换代价小，而且进程内线程可能执行相同的代码，切换线程后也许还可以复用CPU cache数据
3 线程切换和进程切换只依赖CPU是否空闲，和线程进程彼此没关系

PS，上面回复内容其实我也没有确切的资料证实，我是根据第一性原理推导出来的，我觉得仅根据专栏文章内容就可以推导出来，我希望你也可以自己分析推导，这样获得的知识会更加稳固

1

夜里饿煮面
2019-11-18

操作系统中PV原语是最小的原子，就是不知道硬件上是怎么实现的

1

peter
2019-11-24

一直以为tomcat是一个独立的进程。根据本文所述，tomcat只是一个线程，是虚拟机进程

中的一个线程。是这样吗?

展开

小烽

2019-11-21

Java 中，锁的状态存在对象头中，包括线程id 等，因为是在堆中，是共享变量，所以所有的线程都能获取到，并通过cas判断是否能拿到锁。换成其他需要应该也是如此，在共享变量中存储锁相关的信息即可。但要具体实现还是有些困难

李老师，您好。我是非计算机专业的，在从事java 开发，听了课程，感觉收获很大，但...

展开

我爱布丁

2019-11-21

老师，看完文章，联想到两个关于协程的问题：

1. 使用协程在出现IO等待时，程序会自己调度去执行其他的(CPU)任务。理论上这样可以避免额外的IO等待导致的线程间切换。我的问题是从系统的角度上看，使用协程可以抢占到更多的CPU时间片吗？...

展开

作者回复: 1 多个协程通过自我调度复用同一个线程，所以某个协程IO等待的时候，会导致整个线程阻塞，并不能避免线程切换。更好的做法是使用异步IO，不要IO等待。

能否抢到时间片是操作系统调度的，协程自己控制不了，但是协程利用自己更轻量级，配合异步IO等方法，可以提高运行效率，整体性能得到优化。

2 是的。但是用好协程，减少额外的线程调度切换，可以提高整体的系统吞吐能力。

Heidi

2019-11-20

你好，想提个问题。文章中大部分知识点都掌握，但是遇到问题的时候没有从这些角度出发，只是跟着一些关联去分析问题，对遇到的问题反应比较慢。这种情况是不是知识没有成体系？那么怎样建立比较完整的知识体系呢？

展开

作者回复: 思考问题从问题的根源出发思考，如果在解决问题的时候现场比较乱，无法做到。事后做复盘，复盘的时候重新从根源思考。

stephen
2019-11-20

在系统层面分为两种锁:自旋锁（spinlock）这种锁是占用CPU直到获取锁，通常底层采用一个while循环获取来更改一个状态，更改的方式采用的是cas原语，底层CPU指令为rdtsc，这个操作不可中断，语义为，比较并交换，比较两者的值相等就交换目标值为指定值。还有一种锁，这种锁并不会占用CPU，而是在未获取锁之前等待，这个等待将让出CPU，供其他程序使用。而jvm中的线程模型和系统不同，锁的内部情况个人认为基本上是模拟...
展开

超群
2019-11-20

李老师可以写一篇细说并发的文章吗？现在有很多写高并发的书或者文章，但是几乎没有人写过并发是什么。对于并发总感觉一种雾里看花的感觉，有点清楚但是又不是很清楚。希望李老师考虑一下，谢谢

作者回复: 专栏后面还有一些文章讨论高并发，包括反应式编程，分布式架构等等，欢迎继续关注~

y欧尼酱
2019-11-19

空间换时间，时间换空间。
展开

Asura
2019-11-19

老师赶紧更新，无限期待中😄
展开

作者回复:

马超
2019-11-19

个人理解，欢迎指正
1.对地址总线加锁，粒度大，只有一个线程能访问内存，其他线程都会阻塞
2.对一块内存加锁，粒度小
但是感觉很模糊，因为对cpu和内存的底层硬件电路不懂

展开

Cy23
2019-11-19

听着还不错，希望继续试听后边的决定是否购入

展开