关于java线程模型的总结

https://www.bilibili.com/video/BV1ix41137Eu?from=search&seid=10570905734118987025

该发言可能违规，

在Java中，基本我们说的线程（Thread）实际上应该叫作“**用户线程**”，而对应到操作系统，还有另外一种线程叫作“**内核线程**”。

用户线程和内核线程之间必然存在某种关系，多对一模型、一对一模型和多对多模型

多对一线程模型

多个用户线程对应到同一个内核线程上，线程的创建、调度、同步的所有细节全部由进程的用户空间线程库来处理。

优点：

- 用户线程的很多操作对内核来说都是透明的，不需要用户态和内核态的频繁切换，使线程的创建、调度、同步等非常快；

缺点：

- 由于多个用户线程对应到同一个内核线程，如果其中一个用户线程阻塞，那么该其他用户线程也无法执行；
- 内核并不知道用户态有哪些线程，无法像内核线程一样实现较完整的调度、优先级等；

一对一模型

即一个用户线程对应一个内核线程，内核负责每个线程的调度

优点：

- （比如JVM几乎把所有对线程的操作都交给了内核）实现线程模型的容器（jvm）简单，所以我们经常听到在java中使用线程一定要慎重就是这个原因；

缺点：

- 对用户线程的大部分操作都会映射到内核线程上，引起用户态和内核态的频繁切换；
- 内核为每个线程都映射调度实体，如果系统出现大量线程，会对系统性能有影响；

多对多模型

太复杂了，不需要了解

内核态还是用户态

如果有以上的认知，那么一个java的线程在运行的时候是内核态还是用户态呢?

其实这是个伪命题因为一个软件级别的线程用户态和内核态是不确定的

那么什么是内核态什么用户态呢?

这就要说到mmu和mmap了

linux系统的虚拟地址映射

一、物理地址空间是什么

理解虚拟地址空间还得从物理地址空间开始说起。我们知道内存就像一个数组，每个存储单元被分配了一个地址，这个地址就是物理地址，所有物理地址构成的集合就是物理地址空间。物理地址也就是真实的地址，对应真实的那个内存条。

二、虚拟地址空间是什么

引入虚拟地址之后，对于每一个进程，操作系统提供一种假象，让每个进程感觉自己拥有一个巨大的连续的内存可以使用，这个虚拟的空间甚至还可以比内存的容量还大。这个“假象”就是虚拟地址空间。虚拟地址是面向每个进程的，只是一个“假象”罢了。

CPU使用**虚拟地址**向内存寻址，通过专用的内存管理单元（MMU）硬件把虚拟地址转换为真实的物理地址

intel x86 CPU有四种不同的执行级别0-3，linux只使用了其中的0级和3级分别来表示内核态和用户态，所谓的内核态和用户态其实仅仅是CPU的一个权限而已

用户态切换到内核态的3种方式

a. 系统调用

b. 异常（这个异常不是java当中的异常）

c. 外围设备的中断

其实站在java程序员的角度只需要关注系统调用，因为系统调用可以认为是用户进程主动发起的，比如调用线程的park()方法会对应到一个os的一个函数，从而使当前线程进入了内核态；再比如遇到synchronized关键字如果是重量锁则会调用pthread_mutex_lock（）这样我们的线程也会切换到内核态；当执行完系统调用切换到用户态；

什么是切换？有哪些切换

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**

什么是 CPU 上下文

其实和spring上下文差不多，CPU 寄存器和程序计数器就是 CPU 上下文，因为它们都是 CPU 在运行任何任务前，必须的依赖环境

- CPU 寄存器是 CPU 内置的容量小、但速度极快的内存。
- 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。

什么是 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

CPU 上下文切换的类型

根据任务的不同，和java并发编程相关的我们只关心以下两种类型 - 进程上下文切换 - 线程上下文切换

进程上下文切换

1、进程上下文切换之系统调用

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。

在这个过程中就发生了 CPU 上下文切换，整个过程是这样的：
1、保存 CPU 寄存器里原来用户态的指令位
2、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。
3、跳转到内核态运行内核任务。
4、当系统调用结束后，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。系统调用属于同进程内的 CPU 上下文切换

2、真正的进程上下文切换和系统调用有什么区别呢?

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：在保存内核态资源（当前进程的内核状态和 CPU 寄存器）之前，需要先把该进程的用户态资源（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。

发生进程上下文切换的场景：

1. 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。
2. 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。
3. 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。

4. 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

线程上下文切换

特点以及场景：

1. 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。
2. 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

和一个同学的套路（cas不会升级内核态，他仅仅是处理器提供的一个指令，速度非常快）