> *本文由 简悦 SimpRead 转码， 原文地址 hollischuang.gitee.io*

> *Description*

主流的操作系统都提供了线程实现，实现线程主要有 3 种方式：使用内核线程实现、使用用户线程实现和使用用户线程加轻量级进程混合实现。

使用内核线程实现

内核线程（Kernel-Level Thread,KLT）就是直接由操作系统内核（Kernel，下称内核）支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器（Scheduler）对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫做多线程内核（Multi-Threads Kernel）。

程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口——轻量级进程（Light Weight Process,LWP），轻量级进程就是我们通常意义上所讲的线程，由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程。这种轻量级进程与内核线程之间 1:1 的关系称为一对一的线程模型，如图所示。

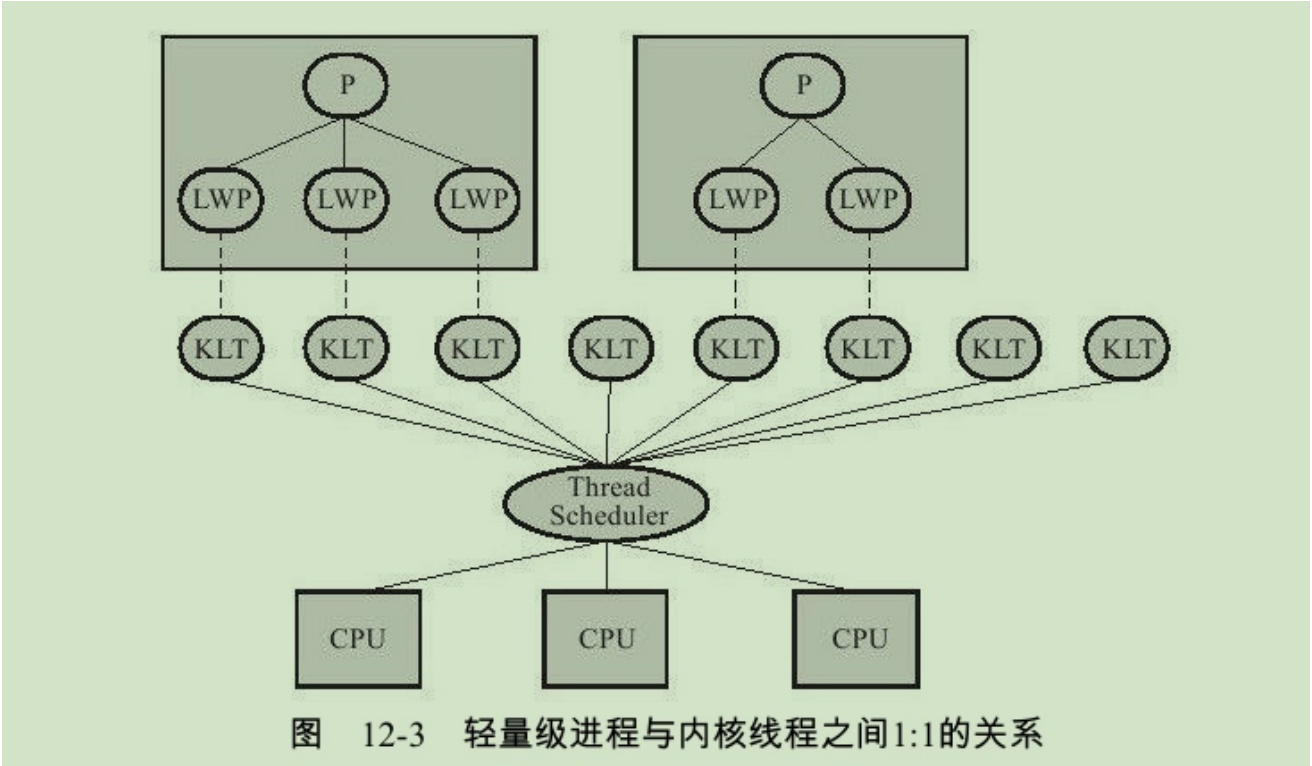

图 12-3 轻量级进程与内核线程之间1:1的关系

由于内核线程的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞了，也不会影响整个进程继续工作，但是轻量级进程具有它的局限性：首先，由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态（User Mode）和内核态（Kernel Mode）

中来回切换。其次，每个轻量级进程都需要有一个内核线程的支持，因此轻量级进程要消耗一定的内核资源（如内核线程的栈空间），因此一个系统支持轻量级进程的数量是有限的。

使用用户线程实现

从广义上来讲，一个线程只要不是内核线程，就可以认为是用户线程（User Thread,UT），因此，从这个定义上来讲，轻量级进程也属于用户线程，但轻量级进程的实现始终是建立在内核之上的，许多操作都要进行系统调用，效率会受到限制。

而狭义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。这种进程与用户线程之间 1：N 的关系称为一对多的线程模型，如图所示。

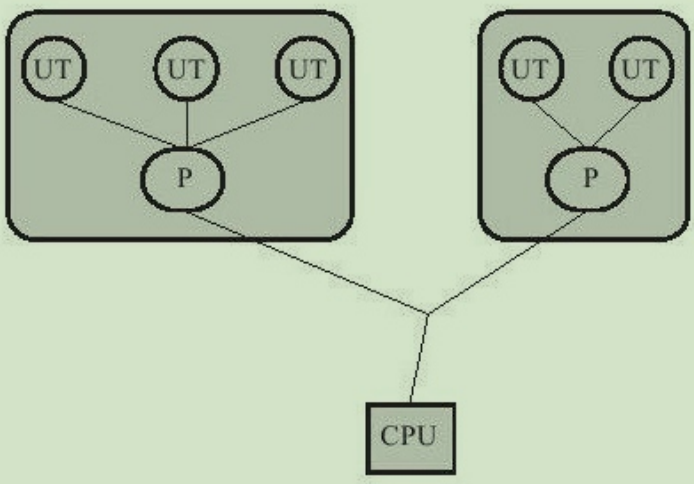

图　12-4　进程与用户线程之间1：N的关系

使用用户线程的优势在于不需要系统内核支援，劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理。线程的创建、切换和调度都是需要考虑的问题，而且由于操作系统只把处理器资源分配到进程，那诸如“阻塞如何处理”、“多处理器系统中如何将线程映射到其他处理器上”这类问题解决起来将会异常困难，甚至不可能完成。因而使用用户线程实现的程序一般都比较复杂，除了以前在不支持多线程的操作系统中（如 DOS）的多线程程序与少数有特殊需求的程序外，现在使用用户线程的程序越来越少了，Java、Ruby 等语言都曾经使用过用户线程，最终又都放弃使用它。

使用用户线程加轻量级进程混合实现

线程除了依赖内核线程实现和完全由用户程序自己实现之外，还有一种将内核线程与用户线程一起使用的实现方式。在这种混合实现下，既存在用户线程，也存在轻量级进程。用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发。而操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁，这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级线程来完成，大大降低了整个进程被完全阻塞的风险。在这种混合模式中，用户线程与

轻量级进程的数量比是不定的，即为 N：M 的关系，如图 12-5 所示，这种就是多对多的线程模型。

许多 UNIX 系列的操作系统，如 Solaris、HP-UX 等都提供了 N：M 的线程模型实现。

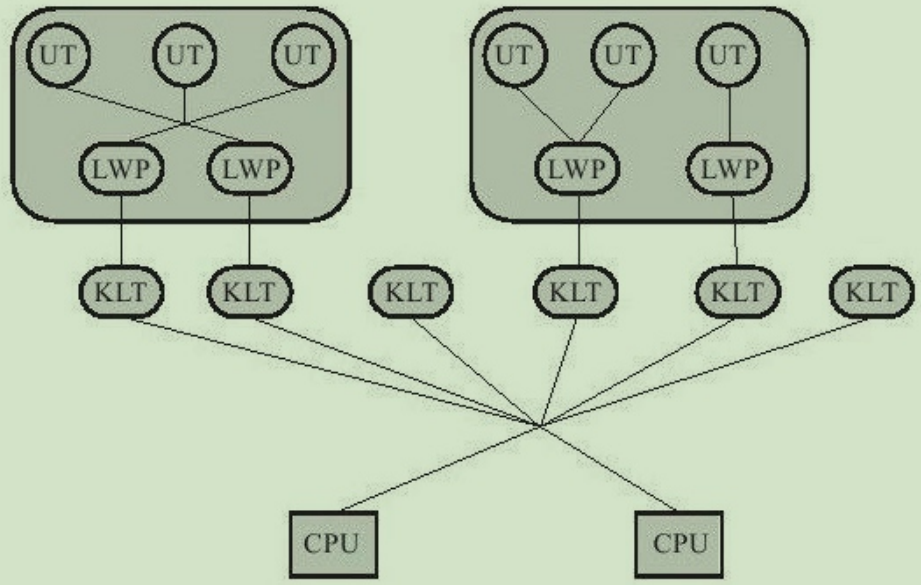

图 12-5 用户线程与轻量级进程之间N：M的关系

Java 线程的实现

Java 线程在 JDK 1.2 之前，是基于称为“绿色线程”（Green Threads）的用户线程实现的，而在 JDK 1.2 中，线程模型替换为基于操作系统原生线程模型来实现。因此，在目前的 JDK 版本中，操作系统支持怎样的线程模型，在很大程度上决定了 Java 虚拟机的线程是怎样映射的，这点在不同的平台上没有办法达成一致，虚拟机规范中也并未限定 Java 线程需要使用哪种线程模型来实现。线程模型只对线程的并发规模和操作成本产生影响，对 Java 程序的编码和运行过程来说，这些差异都是透明的。

对于 Sun JDK 来说，它的 Windows 版与 Linux 版都是使用一对一的线程模型实现的，一条 Java 线程就映射到一条轻量级进程之中，因为 Windows 和 Linux 系统提供的线程模型就是一对一的。

而在 Solaris 平台中，由于操作系统的线程特性可以同时支持一对一（通过 Bound Threads 或 Alternate Libthread 实现）及多对多（通过 LWP/Thread Based Synchronization 实现）的线程模型，因此在 Solaris 版的 JDK 中也对应提供了两个平台专有的虚拟机参数：-XX：+UseLWPSynchronization（默认值）和 - XX：+UseBoundThreads 来明确指定虚拟机使用哪种线程模型。 Java 语言则提供了在不同硬件和操作系统平台下对线程操作的统一处理，每个已经执行 start（）且还未结束的 java.lang.Thread 类的实例就代表了一个线程。我们注意到 Thread 类与大部分的 Java API 有显著的差别，它的所有关键方法都是声明为 Native 的。在 Java API 中，一个 Native 方法往往意味着这个方法没有使用或无法使用平台无关的手段来实现（当然也可能是为了执行效率而使用 Native 方法，不过，通常最高效率的手段也就是平台相关的手段）。

(参考：深入理解 Java 虚拟机)