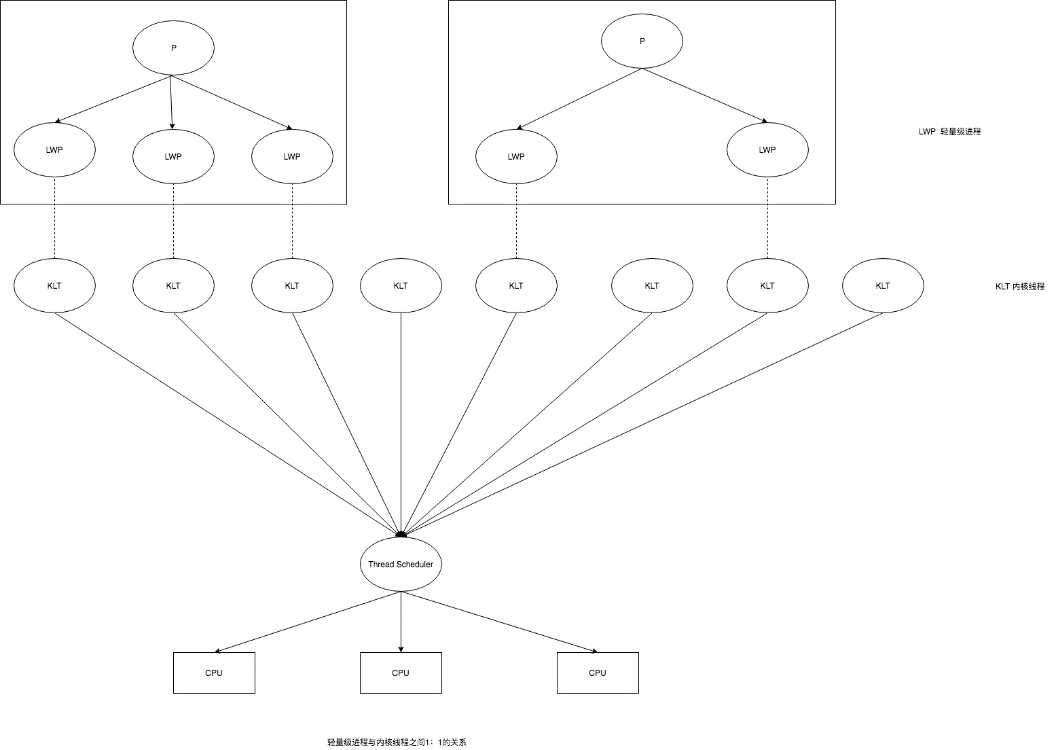
## 线程的实现

线程主要实现方式有三种：使用内核线程实现、用户线程实现、用户线程加轻量级进程实现。

### 内核线程方式

内核线程（Kernel-Level Thread,KLT）就是直接由操作系统内核支持的线程。每个内核线程可以视为内核的一个分身，

    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的关系成为一对一的线程模型。



### 使用用户线程实现

    从广义上讲，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。

    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”、“多处理器系统中如果将线程映射到其他处理器上”这类问题解决起来将会异常的困难，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，即为N:M的关系。

    许多UNIX系列的操作系统，如Solaris、HP-UX等提供了N:M的线程模式实现。

### java线程的实现

    java线程在JDK1.2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。线程模式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。

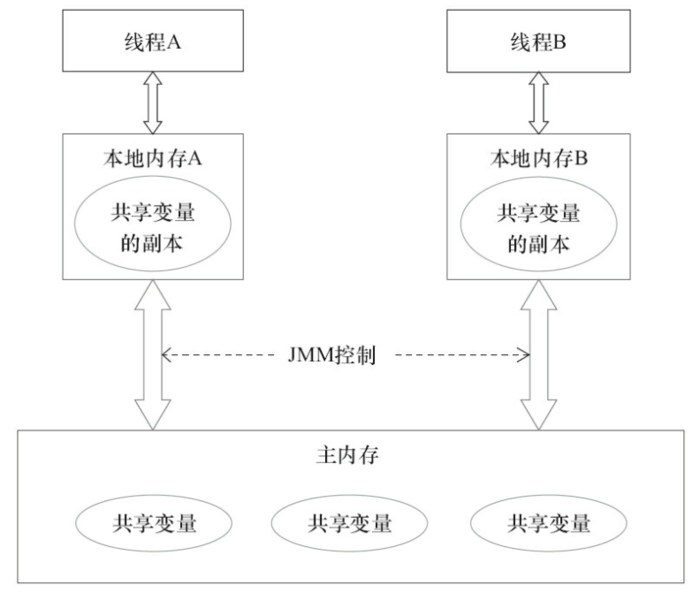
    对于Sun JDK来说，他的Windows版和Linux版都是使用一对一的线程模型来实现的，一条java线程就映射到一条轻量级进程之中，因为Windows和Linux系统提供的线程模型就是就是一对一的。

    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:+UseL WPSynchronization(默认值)和-XX:+UseBoundThreads来明确指定虚拟机使用哪种线程模型。

    java的线程是通过映射到系统的原生线程上来实现的。

### JAVA内存模型

JAVA内存模型（JMM）主要目的是定义程序中各种变量的访问规则，即如何关注虚拟机实现从内存取值和往内存写值的操作（此处不包含局部变量和方法参数线程私有变量）。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。



java内存模型定义了8种操作来完成。这8种操作每一种都是原子操作。8种操作如下：

lock(锁定)：作用于主内存，它把一个变量标记为一条线程独占状态；

read(读取)：作用于主内存，它把变量值从主内存传送到线程的工作内存中，以便随后的load动作使用；

load(载入)：作用于工作内存，它把read操作的值放入工作内存中的变量副本中；

use(使用)：作用于工作内存，它把工作内存中的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量的指令时候，将会执行这个动作；

assign(赋值)：作用于工作内存，它把从执行引擎获取的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的指令时候，执行该操作；

store(存储)：作用于工作内存，它把工作内存中的一个变量传送给主内存中，以备随后的write操作使用；

write(写入)：作用于主内存，它把store传送值放到主内存中的变量中。

unlock(解锁)：作用于主内存，它将一个处于锁定状态的变量释放出来，释放后的变量才能够被其他线程锁定；

Java内存模型还规定了执行上述8种基本操作时必须满足如下规则:

（1）不允许read和load、store和write操作之一单独出现（即不允许一个变量从主存读取了但是工作内存不接受，或者从工作内存发起会写了但是主存不接受的情况），以上两个操作必须按顺序执行，但没有保证必须连续执行，也就是说，read与load之间、store与write之间是可插入其他指令的。

（2）不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。

（3）不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中。

（4）一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

（5）一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。

（6）如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。

（7）如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。

（8）对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作。

as-if-serial:

as-if-serial语义的意思是：不管怎么重排序（编译器和处理器为了提供并行度），（单线程）程序的执行结果不能被改变。编译器，runtime和处理器都必须遵守as-if-serial语义。as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器，runtime和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。

#### happen before：

用happens-before的概念来阐述操作之间的内存可见性。在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系 。

两个操作之间具有happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行！happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前（the first is visible to and ordered before the second） 。

1）如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。(对程序员来说)

2）两个操作之间存在happens-before关系，并不意味着Java平台的具体实现必须要按照happens-before关系指定的顺序来执行。如果重排序之后的执行结果，与按happens-before关系来执行的结果一致，那么这种重排序是允许的(对编译器和处理器 来说)

Java程序员并不关心两个操作的执行顺序，只是希望保持语义不变，因此happens-before关系本质上和as-if-serial语义是一回事。as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。

#### volatile变量自身具有下列特性：

可见性。对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入(内存屏障)。

原子性：对任意单个volatile变量的读/写具有原子性。

LockSupport 、Unsafe工具类

LockSupport 几个方法

void park()：阻塞当前线程，如果调用unpark方法或者当前线程被中断，从能从park()方法中返回

void park(Object blocker)：功能同方法1，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；

void parkNanos(long nanos)：阻塞当前线程，最长不超过nanos纳秒，增加了超时返回的特性；

void parkNanos(Object blocker, long nanos)：功能同方法3，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；

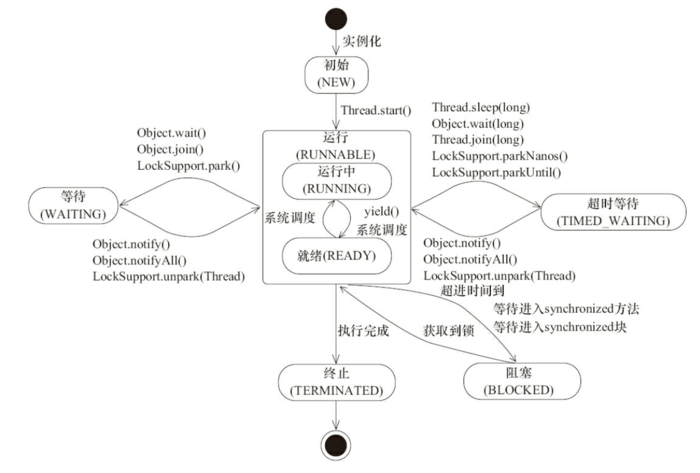
void parkUntil(long deadline)：阻塞当前线程，知道deadline；

void parkUntil(Object blocker, long deadline)：功能同方法5，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；

唤醒线程

void unpark(Thread thread):唤醒处于阻塞状态的指定线程。

Unsafe 的Cas操作



## Java线程安全

只要在多个线程访问下，都能得到正确的结果就是线程安全的。

常见保证安全的方式

1. 不可变对象 域都是final修饰，且正常创建未发生this引用溢出
2. 线程封闭 堆栈封闭：局部变量、threadLocal
3. 线程安全类
4. 锁

## AQS（ AbstractQueuedSynchronizer ）

