1. 什么是线程？

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位，可以使用多线程对 进行运算提速。

比如，如果一个线程完成一个任务要 100 毫秒，那么用十个线程完成改任务只需 10 毫秒

2. 并行和并发有什么区别？

- 并行：多个处理器或多核处理器同时处理多个任务。
- 并发：多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。

3. 线程和进程的区别？

一个程序下至少有一个进程，一个进程下至少有一个线程，一个进程下也可以有多个线程来增加程序的执行速度。

4. 守护线程是什么？

守护线程是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。在 Java 中垃圾回收线程就是特殊的守护线程。

5. 什么是线程安全和线程不安全？

1 、线程安全 线程安全: 就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行 访问，直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

Vector 是用同步方法来实现线程安全的, 而和它相似的 ArrayList 不是线程安全的。

2 、线程不安全 线程不安全：就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据 线程安全问题都是由全局变量及静态变量引起的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个 线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

6. 创建线程有哪几种方式？

创建线程有三种方式：

- 继承 Thread 重写 run 方法；
- 实现 Runnable 接口；
- 实现 Callable 接口。

7. 说一下 runnable 和 callable 有什么区别？

runnable 没有返回值，callable 可以拿到有返回值，callable 可以看作是 runnable 的补充。

8. 线程有哪些状态？

线程的状态：

• NEW 尚未启动

• RUNNABLE 正在执行中

• BLOCKED 阻塞的（被同步锁或者 IO 锁阻塞）

• WAITING 永久等待状态

• TIMED_WAITING 等待指定的时间重新被唤醒的状态

• TERMINATED 执行完成

9. 什么是 CAS?

1 、 CAS （compare and swap）的缩写，中文翻译成比较并交换。

2 、 CAS 不通过 JVM,直接利用java 本地方 JNI （Java Native Interface 为 JAVA 本地调用） ,直接调用 CPU 的 cmpxchg （是 汇编指令）指令。

3、利用CPU 的 CAS 指令，同时借助 JNI 来完成 Java 的非阻塞算法,实现原子操作。其它原子操作都是利用类似的特性完成 的。

4 、整个 java.util.concurrent 都是建立在 CAS 之上的，因此对于 synchronized 阻塞算法，J.U.C 在性能上有了很大的提升。

5 、 CAS 是项乐观锁技术 ，当多个线程尝试使用CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

使用CAS 在线程冲突严重时，会大幅降低程序性能；CAS 只适合于线程冲突较少的情况使用。

synchronized 在 jdk1.6 之后，已经改进优化。synchronized 的底层实现主要依靠 Lock-Free 的队列，基本思路是自旋后阻塞，竞 争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。 在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于 CAS。

10. sleep() 和 wait() 有什么区别？

- 类的不同：sleep() 来自 Thread，wait() 来自 Object。
- 释放锁：sleep() 不释放锁；wait() 释放锁。
- 用法不同：sleep() 时间到会自动恢复；wait() 可以使用 notify()/notifyAll()直接唤醒。

11. notify()和 notifyAll()有什么区别？

notifyAll()会唤醒所有的线程，notify()之后唤醒一个线程。notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而 notify()只会唤醒一个线程，具体唤醒哪一个线程由虚拟机控制。

12. 线程的 run() 和 start() 有什么区别？

start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start() 只能调用一次。

13. 创建线程池有哪几种方式？

线程池创建有七种方式，最核心的是最后一种：

• newSingleThreadExecutor()：它的特点在于工作线程数目被限制为 1，操作一个无界的工作队列，所以它保证了所有任务的都是被顺序执行，最多会有一个任务处于活动状态，并且不允许使用者改动线程池实例，因此可以避免其改变线程数目；

• newCachedThreadPool()：它是一种用来处理大量短时间工作任务的线程池，具有几个鲜明特点：它会试图缓存线程并重用，当无缓存线程可用时，就会创建新的工作线程；如果线程闲置的时间超过 60 秒，则被终止并移出缓存；长时间闲置时，这种线程池，不会消耗什么资源。其内部使用 SynchronousQueue 作为工作队列；

• newFixedThreadPool(int nThreads)：重用指定数目（nThreads）的线程，其背后使用的是无界的工作队列，任何时候最多有 nThreads 个工作线程是活动的。这意味着，如果任务数量超过了活动队列数目，将在工作队列中等待空闲线程出现；如果有工作线程退出，将会有新的工作线程被创建，以补足指定的数目 nThreads；

• newSingleThreadScheduledExecutor()：创建单线程池，返回 ScheduledExecutorService，可以进行定时或周期性的工作调度；

• newScheduledThreadPool(int corePoolSize)：和 newSingleThreadScheduledExecutor() 类似，创建的是个 ScheduledExecutorService，可以进行定时或周期性的工作调度，区别在于单一工作线程还是多个工作线程；

• newWorkStealingPool(int parallelism)：这是一个经常被人忽略的线程池，Java 8 才加入这个创建方法，其内部会构建 ForkJoinPool，利用 Work-Stealing 算法，并行地处理任务，不保证处理顺序；

• ThreadPoolExecutor()：是最原始的线程池创建，上面 1-3 创建方式都是对 ThreadPoolExecutor 的封装。

14. 什么是乐观锁和悲观锁？

1、悲观锁

Java 在 JDK1.5 之前都是靠 synchronized 关键字保证同步的，这种通过使用一致的锁定协议来协调对共享状态的访 问，可以确保无论哪个线程持有共享变量的锁，都采用独占的方式来访问这些变量 。独占锁其实就是一种悲观锁，所以 可以说 synchronized 是悲观锁。

2 、乐观锁

乐观锁（ Optimistic Locking）其实是一种思想。相对悲观锁而言， 乐观锁假设认为数据一般情况下不会造成冲突， 所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测 ，如果发现冲突了，则让返回用户错误的信息， 让用户决定如何去做。

memcached 使用了 cas 乐观锁技术保证数据一致性。

15. 什么是 Executors 框架？

Java 通过 Executors 提供四种线程池，分别为：

1 、 newCachedThreadPool 创建一个可缓存线程池， 如果线程池长度超过处理需要，可灵活回收空闲线程，若无可 回收，则新建线程。

2 、 newFixedThreadPool 创建一个定长线程池， 可控制线程最大并发数，超出的线程会在队列中等待。

3 、 newScheduledThreadPool 创建一个定长线程池， 支持定时及周期性任务执行。

4 、 newSingleThreadExecutor 创建一个单线程化的线程池， 它只会用唯一的工作线程来执行任务，保证所有任务 按照指定顺序(FIFO, LIFO, 优先级)执行。

16. 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？

1 、 JDK7 提供了 7 个阻塞队列 。（也属于并发容器）

i. ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列。

ii. LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列。

iii. PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列。

iv. DelayQueue：一个使用优先级队列实现的无界阻塞队列。

v. SynchronousQueue：一个不存储元素的阻塞队列。

vi. LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

vii. LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

2、概念： 阻塞队列是一个在队列基础上又支持了两个附加操作的队列。

3 、 2 个附加操作：

支持阻塞的插入方法：队列满时，队列会阻塞插入元素的线程，直到队列不满。

支持阻塞的移除方法：队列空时，获取元素的线程会等待队列变为非空。

17. 线程池都有哪些状态？

• RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。

• SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。

• STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。

• TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。

• TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

18. 线程池中 submit() 和 execute() 方法有什么区别？

• execute()：只能执行 Runnable 类型的任务。

• submit()：可以执行 Runnable 和 Callable 类型的任务。 Callable 类型的任务可以获取执行的返回值，而 Runnable 执行无返回值。

19. 在 Java 程序中怎么保证多线程的运行安全？

• 方法一：使用安全类，比如 Java. util. concurrent 下的类。

• 方法二：使用自动锁 synchronized。

• 方法三：使用手动锁 Lock。

手动锁 Java 示例代码如下：

```
Lock lock = new ReentrantLock();
lock. lock();
try {
    System. out. println("获得锁");
} catch (Exception e) {
    // TODO: handle exception
} finally {
    System. out. println("释放锁");
    lock. unlock();
}
```

20. 多线程中 synchronized 锁升级的原理是什么？

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。 锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

21. 什么是死锁？

当线程 A 持有独占锁 a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

22. 怎么防止死锁？

- 尽量使用 tryLock(long timeout, TimeUnit unit)的方法(ReentrantLock、ReentrantReadWriteLock)，设置超时时间，超时可以退出防止死锁。

- 尽量使用 Java. util. concurrent 并发类代替自己手写锁。
- 尽量降低锁的使用粒度，尽量不要几个功能用同一把锁。
- 尽量减少同步的代码块。

23. 什么是 Callable 和 Future?

1、 Callable 和 Future 是比较有趣的一对组合。当我们需要获取线程的执行结果时，就需要用到它们。 Callable用于产生 结果， Future用于获取结果。

2 、 Callable 接口使用泛型去定义它的返回类型。 Executors 类提供了一些有用的方法去在线程池中执行Callable 内的任 务。由于 Callable 任务是并行的，必须等待它返回的结果。 java.util.concurrent.Future 对象解决了这个问题。

3 、 在线程池提交 Callable 任务后返回了一个 Future 对象，使用它可以知道 Callable 任务的状态和得到 Callable 返回的执行 结果。 Future 提供了 get()方法，等待 Callable 结束并获取它的执行结果。

24. ThreadLocal 是什么？有哪些使用场景？

ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal 的经典使用场景是数据库连接和 session 管理等。

25. 说一下 synchronized 底层实现原理？

synchronized 是由一对 monitorenter/monitorexit 指令实现的，monitor 对象是同步的基本实现单元。在 Java 6 之前，monitor 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作，性能也很低。但在 Java 6 的时候，Java 虚拟机 对此进行了大刀阔斧地改进，提供了三种不同的 monitor 实现，也就是常说的三种不同的锁：偏向锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

26. synchronized 和 volatile 的区别是什么？

• volatile 是变量修饰符；synchronized 是修饰类、方法、代码段。

• volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。

• volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

27. synchronized 和 Lock 有什么区别？

• synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。

• synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。

• 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

28. synchronized 和 ReentrantLock 区别是什么？

synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。

主要区别如下：

• ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；

• ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；

• ReentrantLock 只适用于代码块锁，而 synchronized 可用于修饰方法、代码块等。

29. 说一下 atomic 的原理？

atomic 主要利用 CAS (Compare And Wwap) 和 volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

30. 什么是多线程的上下文切换？

1 、多线程：是指从软件或者硬件上实现多个线程的并发技术。

2 、多线程的好处：

i. 使用多线程可以把程序中占据时间长的任务放到后台去处理，如图片、视屏的下载

ii. 发挥多核处理器的优势，并发执行让系统运行的更快、更流畅，用户体验更好

3、多线程的缺点：

1. 大量的线程降低代码的可读性；

2. 更多的线程需要更多的内存空间

3. 当多个线程对同一个资源出现争夺时候要注意线程安全的问题。

4 、多线程的上下文切换：PU 通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保 存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态

31. start()方法和 run()方法的区别？

1 、start()方法来启动一个线程，真正实现了多线程运行。

2 、 如果直接调用 run(),其实就相当于是调用了一个普通函数而已，直接调用 run()方法必须等待 run()方法执行完毕才能执 行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是 run()方法。

32. Runnable 接口和 Callable 接口的区别？

1. Runnable 接口中的 run()方法的返回值是 void，它做的事情只是纯粹地去执行run()方法中的代码而已；
2. Callable 接口中的 call()方法是有返回值的，是一个泛型，和 Future 、 FutureTask 配合可以用来获取异步执行的结果。

33. volatile 关键字的作用？

1. 多线程主要围绕可见性和原子性两个特性而展开， 使用volatile 关键字修饰的变量，保证了其在多线程之间的可见性， 即每次读取到 volatile 变量，一定是最新的数据。
2. 代码底层执行不像我们看到的高级语言——Java 程序这么简单，它的执行是 Java 代码–>字节码–>根据字节码执行对应 的 C/C++代码–>C/C++代码被编译成汇编语言–>和硬件电路交互， 现实中，为了获取更好的性能 JVM 可能会对指令进行 重排序，多线程下可能会出现一些意想不到的问题。使用volatile 则会对禁止语义重排序，当然这也一定程度上降低了代 码执行效率。

34. Java 中如何获取到线程 dump文件？

死循环、死锁、阻塞、页面打开慢等问题，查看线程 dump 是最好的解决问题的途径。所谓线程 dump 也就是线程堆栈， 获取到线程堆栈有两步：

1 、 获取到线程的 pid，可以通过使用jps 命令，在 Linux 环境下还可以使用ps -ef | grep java

2 、打印线程堆栈，可以通过使用jstack pid 命令，在 Linux 环境下还可以使用kill -3 pid

3 、另外提一点， Thread 类提供了一个 getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法 是和具体线程实例绑定的，每次获取到的是具体某个线程当前运行的堆栈。

35. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

1. 高并发、任务执行时间短的业务： 线程池线程数可以设置为 CPU 核数+1 ，减少线程上下文的切换。
2. 并发不高、任务执行时间长的业务要区分开看：

   a. 假如是业务时间长集中在 IO 操作上，也就是 IO 密集型的任务，因为 IO 操作并不占用CPU，所以不要让所有的 CPU 闲下来，可以加大线程池中的线程数目，让 CPU 处理更多的业务

   b. 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（ 1）一样吧，线程池中的线 程数设置得少一些，减少线程上下文的切换

3. 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数 据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（ 2）。最后，业务执行时间长的问题， 也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

36. 如果你提交任务时，线程池队列已满，这时会发生什么？

1 、如果你使用的 LinkedBlockingQueue，也就是无界队列 的话，没关系， 继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务；

2 、如果你使用的是 有界队列 比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中， ArrayBlockingQueue 满了，则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

37. 锁的等级：方法锁、对象锁、类锁?

1. 方法锁（synchronized 修饰方法时）

   a. 通过在方法声明中加入 synchronized 关键字来声明 synchronized 方法。

   b. synchronized 方法控制对类成员变量的访问：

   c. 每个类实例对应一把锁 ，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻 塞， 方法一旦执行，就独占该锁， 直到从

该方法返回时才将锁释放 ，此后被阻塞的线程方能获得该锁，重新进入可 执行状态。 这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态，从而有效避免了类成员变量的访问冲突。

2. 对象锁（synchronized 修饰方法或代码块）

   a. 当一个对象中有 synchronized method 或 synchronized block 的时候调用此对象的同步方法或进入其同步区域时，就必须先获得对象锁。 如果此对象的对象锁已被其他调用者占用，则需要等待此锁被释放。（方法锁也是对象锁）

   b. java 的所有对象都含有 1 个互斥锁，这个锁由 JVM自动获取和释放。线程进入 synchronized方法的时候获取该对象 的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待； synchronized方法正常返回或者抛异常而 终止，JVM 会自动释放对象锁 。 这里也体现了用synchronized 来加锁的 1 个好处，方法抛异常的时候，锁仍然可以由 JVM 来自动释放。

3. 类锁(synchronized 修饰静态的方法或代码块)

   a. 由于一个 class 不论被实例化多少次，其中的静态方法和静态变量在内存中都只有一份。所以，一旦一个静态的方 法被申明为 synchronized。此类所有的实例化对象在调用此方法， 共用同一把锁，我们称之为类锁。

4. 对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步

38. 如果同步块内的线程抛出异常会发生什么？

synchronized方法正常返回或者抛异常而终止， JVM 会自动释放对象锁

39. 并发编程（concurrency）并行编程（parallellism）有什么区别？

1. 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
2. 解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。
3. 解释三：在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如 hadoop 分布式集群 所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

40. 如何保证多线程下 i++ 结果正确？

1. volatile 只能保证你数据的可见性，获取到的是最新的数据，不能保证原子性；
2. 用AtomicInteger 保证原子性。
3. synchronized 既能保证共享变量可见性，也可以保证锁内操作的原子性。

41. 一个线程如果出现了运行时异常会怎么样?

1. 如果这个异常没有被捕获的话，这个线程就停止执行了。
2. 另外重要的一点是：如果这个线程持有某个对象的监视器，那么这个对象监视器会被立即释放.

42. 如何在两个线程之间共享数据?

通过在线程之间共享对象就可以了，然后通过 wait/notify/notifyAll 、await/signal/signalAll 进行唤起和等待，比方说阻塞 队列 BlockingQueue 就是为线程之间共享数据而设计的。

43. 生产者消费者模型的作用是什么?

1. 通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用。
2. 解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不 需要受到相互的制约。

44. 怎么唤醒一个阻塞的线程?

1. 如果线程是因为调用了 wait() 、 sleep()或者 join()方法而导致的阻塞；

1、 suspend 与 resume

Java 废弃 suspend() 去挂起线程的原因，是因为 suspend()在导致线程暂停的同时，并不会去释放任何锁资 源。其他线程都无法访问被它占用的锁。直到对应的线程执行 resume() 方法后，被挂起的线程才能继续，从而其它 被阻塞在这个锁的线程才可以继续执行。

但是，如果 resume() 操作出现在 suspend() 之前执行，那么线程将一直处于挂起状态，同时一直占用锁，这就 产生了死锁。而且，对于被挂起的线程，它的线程状态居然还是 Runnable。

2 、 wait 与 notify

wait 与 notify 必须配合 synchronized 使用，因为调用之前必须持有锁， wait 会立即释放锁， notify 则是同步块执行 完了才释放

3 、await 与 singal

Condition 类提供，而Condition 对象由 new ReentLock().newCondition()获得，与 wait 和 notify 相同，因为使用 Lock 锁后无法使用wait方法

4 、 park 与 unpark

LockSupport 是一个非常方便实用的线程阻塞工具，它可以在线程任意位置让线程阻塞。和 Thread.suspenf()相 比，它弥补了由于 resume()在前发生，导致线程无法继续执行的情况。和 Object.wait()相比，它不需要先获得某个对 象的锁，也不会抛出 IException 异常。可以唤醒指定线程。

2. 如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作系统实现的， Java 代码并没有办法直接接触到操作系统。

45. Java 中用到的线程调度算法是什么

1. 抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一 个时间片给某个线程执行。

46. 单例模式的线程安全性?

老生常谈的问题了，首先要说的是单例模式的线程安全意味着： 某个类的实例在多线程环境下只会被创建一次出来 。单 例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

47. 线程类的构造方法、静态块是被哪个线程调用的?

线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run方法里面的代码才是被线程自身所调用的。

48. 同步方法和同步块，哪个是更好的选择?

同步块是更好的选择，因为它不会锁住整个对象（当然也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕 这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

synchronized(this)以及非static 的 synchronized方法（至于 static synchronized方法请往下看），只能防止多个线程同时 执行同一个对象的同步代码段。

如果要锁住多个对象方法，可以锁住一个固定的对象，或者锁住这个类的 Class 对象。

synchronized 锁住的是括号里的对象，而不是代码。对于非static 的 synchronized方法，锁的就是对象本身也就是 this。

49. 可以运行时 kill 掉一个线程吗？

a. 不可以，线程有 5 种状态，新建（ new）、可运行（runnable）、运行中（running）、阻塞（block）、死亡 （dead）。

b. 只有当线程 run方法或者主线程 main方法结束，又或者抛出异常时，线程才会结束生命周期。

50、线程 a,b,c,d 运行任务，怎么保证当 a,b,c 线程执行完再执行d 线程?

1 、 CountDownLatch 类

一个同步辅助类，常用于某个条件发生后才能执行后续进程。给定计数初始化 CountDownLatch，调用countDown（）方 法，在计数到达零之前， await方法一直受阻塞。

重要方法为 countdown()与 await()；

2 、join方法

将线程 B 加入到线程 A 的尾部，当 A 执行完后 B 才执行。

```
public static void main(String[] args) throws Exception {
```

```
  Th t = new Th("t1");
  Th t2 = new Th("t2");
  t.start();
  t.join();
  t2.start();
}
```

51、高并发系统如何做性能优化？如何防止库存超卖？

1、高并发系统性能优化：

优化程序，优化服务配置，优化系统配置

1.尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。

2.用jprofiler 等工具找出性能瓶颈，减少额外的开销。

3.优化数据库查询语句，减少直接使用hibernate 等工具的直接生成语句（仅耗时较长的查询做优化）。

4.优化数据库结构，多做索引，提高查询效率。

5.统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。

6.能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态 html 来显示）。

7.解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

2、防止库存超卖：

1、悲观锁：在更新库存期间加锁，不允许其它线程修改；

1、数据库锁：select xxx for update；
2、分布式锁；

2、乐观锁：使用带版本号的更新。每个线程都可以并发修改，但在并发时，只有一个线程会修改成功，其它会返回失 败。

1、redis watch：监视键值对，作用时如果事务提交 exec 时发现监视的监视对发生变化，事务将被取消。

3、消息队列：通过 FIFO 队列，使修改库存的操作串行化。

4、总结：总的来说，不能把压力放在数据库上，所以使用 "select xxx for update" 的方式在高并发的场景下是不可行 的。FIFO 同步队列的方式，可以结合库存限制队列长，但是

在库存较多的场景下，又不太适用。所以相对来说，我会倾向于 选择：乐观锁 / 缓存锁 / 分布式锁的方式。