# 内核态|系统调用(system\_call)

1. 安全性
   1. 避免应用肆意妄为
   2. 提供一个中间层，不用硬件种类，驱动的种类，屏蔽隔离。
   3. 保证系统的安全和文档，内核基于权限和规则进行访问控制。
   4. 每个proc好像运行在一个虚拟OS里，控制整个OS一样。并发的前提（PCB）
2. 用户空间应用无法执行内核代码，不能直接调用内核函数。内核在受保护的地址空间运行
   1. 通知到系统，执行内核调用，打开文件->切换到内核态，代表用户线程来运行。
   2. 通知->软中断来通知。引发一个异常，让内核处理该异常->system\_call触发的处理程序。
   3. 图示, 示意图

      描述已自动生成
   4. 从用户空间->内核空间copy数据。
      1. 调用号，调用方法
      2. 执行系统调用的参数，copy\_to\_user，进程空间的目的地址，内核空间的源地址，copy的数据长度。
         1. 是否在用户空间
         2. 是否在用户进程的地址空间内
         3. 是否可读或者可写
      3. Copy\_from\_user
   5. 发生系统系统调用的时候
      1. 寄存器用户态的指令位置->保存起来
      2. 执行内核态的代码，CPU的寄存器PC，更新为内核指令的新位置。
      3. 跳转到内核态执行内核任务。更新CPU的状态字。
      4. 调用完成之后，恢复用户指令的位置，切换到用户空间，继续执行。
      5. 注意：一次系统调用，发生两次上下文切换（硬件上下文切换，CPU上下文切换）。
      6. 系统调用过程中，1个线程在运行。内核线程代替用户线程运行，需要必要的参数，执行数据的copy。零copy->节省来回的copy，提高性能。
3. ULT&KLT
   1. 用户线程
      1. 线程可以进程本身的地址空间去创建，和内核无关，不需要内核参与。在进程内部创建
      2. 用户在用户库帮助下创建的线程，只是对该用户可空间的运行环境可见（内核不知道）。
      3. 协程（go routine）
   2. 内核线程
      1. 由内核创建，对内核可见。用户线程在提供的一些API的帮助下，要求内核创建一个可执行的线程，内核代表该进程创建线程，放入去可执行线程列表。由内核控制。
      2. 当ULT需要内核资源的，KLT和ULT之间的唯一的主要依赖关系。ULT和LWP绑定上。当系统调用时，创建内核线程并调度到LWP上。图示

         描述已自动生成
   3. 线程映射模型
      1. 1v1，linux，windows95-2002
      2. 多v多
4. Linux到底是1v1还是多对多。
   1. LiunxThreads，pthreads，IBM NGPT（next generation POSIX threads）
   2. NPTL(native POSIX thrads library),现代的linux线程模型
   3. 目前是1v1，lwp
5. Java
   1. 栈区
      1. 每个线程包含一个栈区。保存方法中的基础数据类型和自定义对象的引用。对象在堆里面。
      2. 每个方法在执行的时候创建一个栈帧，frame，用于存储局部变量表，操作室栈，动态链接，方法出口信息。每个一方法执行的过程，就对应一个栈帧在vm栈中从入栈到出栈的过程。
      3. 每个栈的数据都是私有的，其他栈不能访问。线程安全的基础。
   2. 三部分
      1. 基本类型变量区
      2. 执行环境上下文
      3. 操作指令区
   3. 堆，所有线程共享。不存放基本类型和对象引用，只放对象本身。
   4. 图形用户界面

      描述已自动生成
6. 图片包含 图形用户界面

   描述已自动生成
7. Juc
   1. reentrantrwLock
      1. 可重入
         1. 获取了读锁还能获取
         2. 获取了写锁，可以获取读|写
      2. 降级
         1. 先写锁，再读锁，最后是否，写锁可以降级为读锁，反之不能。
      3. Condtion条件支持
         1. 线程间协调的一种工具，代替object类同步方法
8. AQS
   1. 资源是共享还是独占？
   2. 访问资源线程如何并发管理？等待队列
   3. 线程等不及，如何退出？
   4. 模板方法，抽象类。
      1. 什么是资源？如何定义资源是否可以被访问？
         1. Reentrantlock，资源表示独占锁，state=0表示可用，1被占用，大于1，表示被占用了几次，重入的次数。
         2. Countdownlatch，资源表示倒数计数器，0，归零，所以线程都可以访问，>0，表示所有现场都需要阻塞。
         3. Semaphore，信号量或者令牌<=0，说明没有令牌可用，所有现场都需要阻塞>0，可用，线程获取一个token，state-1，线程每释放一个token，state+1;
         4. 读写锁，资源代表，共享的读锁和独占的写锁，state逻辑上被分成两个16位的无符号short类型数据，记录读锁被多少线程使用，和写锁被重入的次数。
   5. 大部分都是private|final的，不希望关注细节，只需要重写特定的抽象类即可。
   6. 重写特定的api
9. protected boolean tryAcquire(int arg) {  
    throw new UnsupportedOperationException();  
   }
   * 1. 为什么报异常而不是定义抽象方法体。
   1. CAS
      1. CompareAndSet->UnSafe
   2. 同步状态的管理，state
   3. 阻塞-唤醒线程的操作，locksupport作为工具
   4. 线程等待队列的管理，AQS的核心，如何在并发状态下管理被阻塞的线程。等待队列是严格按照FIFO队列，CLH锁的变种。双向链表实现。
      1. 队列中的节点是对线程状态的封装，两种状态：独占和共享。
      2. 独占
         1. CANCELLED(1)，取消，后驱节点被中断或者超时，需要移除队列
         2. SIGNAL(-1)，后驱节点被阻塞，将prev节点类型改为该状态
         3. CONDITION(-2)，Condition专用，
      3. 共享
         1. CALCELLED(1)
         2. SIGNAL(-1)
         3. PROPAGATE(-3)，传播，共享模式
      4. Next指针，维护队列顺序，当临界区资源被释放时，头节点通过next指针找到队列第一个节点
      5. Prev指针，当节点（线程对象）取消的时候，让当前节点的，前驱直接指向当前节点的后驱完成出队动作
   5. 前一节点的某一个属性，代表当前节点的状态。
      1. 等待队列是AQS的核心。
10. reentrantLock公平策略的原理
    1. 文本

       描述已自动生成
    2. 非公平锁图形用户界面, 文本, 应用程序, 网站

       描述已自动生成
    3. 上来先尝试修改state状态，先抢占一下，抢占成功，那就插队成功，否则去排队（公平锁的策略）
    4. 使用场景
       1. 如果锁时间长，并发比较低，公平锁
       2. 否则，非公平锁，更有吞吐量的优势。
11. 对中断的支持
    1. 检查线程如果被中断国内，炮异常：java.util.concurrent.locks.AbstractQueuedSynchronizer#doAcquireInterruptibly
12. 对超时等待的支持
    1. 锁的时候加入一个时间，在自旋尝试获取锁的时候，加入时间的判断，超时就cancel，将自身从队列中移除，GC掉。
13. Condition
    1. 对Object的wait，notify和notifyAll的增强和补充
    2. ABC三个线程
       1. A先lock成功，调用con.await方法
       2. B获取锁，调用con.signal唤醒A
       3. B释放锁
14. StampedLock,1.8引入，对读写锁的增强
    1. 老读写锁，读锁是共享的，写锁是独占的。
    2. 读远远多余写，导致写线程饥饿。
    3. 特点
       1. 所以获取方法，都返回一个邮戳，0代表获取失败，其余都是成功
       2. 释放锁的方法，需要邮戳，必须和成功获取锁的邮戳一致。
       3. 不可重入，已经获取写锁，在此获取造成死锁。
    4. 三种模式
       1. 读、写、乐观读（优化的读模式）
    5. 不支持Condition等待。
    6. 原理还是使用CLH队列进行线程的管理。
    7. **找个使用案例，琢磨琢磨。**
15. **同步框架**
    1. **countDownLatch，倒数计数器，计数器>0，所有现场都会等待，阻塞，为0，全部释放**
       1. **开关、入口、总协调**
          1. **给一个初始值为1的一个CDL，为0之前，所有调用await的线程都在入口处等待。**
       2. **作为一个完成信号**
          1. **跑步一样，所有队员到达终点，比赛完成**
             1. **初始化一个N的CDL，每当队员到达，countDown一下，如果为0，比赛结束。**
             2. **多线程同步协调的一个过程。多个线程全部完成才算完成。**
    2. **CyclicBarrier，不满足某个条件，阻挡住所有线程**
       1. **可循环使用**
       2. **当线程到底barrier时被阻塞，直到到达栅栏的数量满足数量要求的时候，放开。**
       3. **到达一定的数量才会执行。（countdown也可以有这种作用）**
       4. **原理**
          1. **没有直接封装AQS**
          2. **可重入锁和condition实现**
    3. **信号量，Semaphore**
       * 1. **类似于许可证**
         2. **使用场景？初始化一定数量的token，当有线程需要访问共享资源或者需要线程同步的时候，先获取（acquire），acquire成功，ok，否则就阻塞等待，有了许可之后，就被唤醒。用完了之后，需要release。供其他线程去使用。也支持公平和非公平的策略。**
         3. **原理？和可重入锁原理一样的，实现就是的类似的。**
       1. **Exchanger**
          1. **交换器。**
          2. **结构是什么？**
          3. **原理？使用场景？和countdownLatch和cyclicBarrier的区别是什么？**
       2. **Phaser，1.7引入的一个工具类，用于分阶段任务的出来。一个多阶段的CyclicBarrier。功能更强大**
          1. **中途可以增加或者减少参与者，会进行一个升级或者降级**
          2. **原理？**
    4. **Executor、fork/join**
       1. **线程池？**

maximumPoolSize

corePoolSize

queue

作用？

* + 1. **线程池的大小如何设置？**
       1. **如果CPU密集型，那么就约等于(小于等于)CPU的核树，逻辑CPU的数量。**
       2. **如果IO密集型，约等于 平均响应时间（ms）\*QPS。线程池数量一般远远大于CPU的核树的。因为这个线程等待IO，网络IO，处于D，不可中断睡眠态，CPU其实是空闲，所以要加大线程池的数量，提高CPU的利用率，提高吞吐量。如果线程池的数量=CPU的核树，CPU使用率非常低，大部分时间都是空闲的。可以通过压测的方式得出大概的平均响应时间。**

1. **Future模式**
   1. **异步的来执行任务，在需要的时候获取结果**
   2. **可以有选择的等待获取取消，获取计算结果。**
   3. **Futuretask**
      1. **原理和实现**

CompletableFuture

1. **Fork/join模式**
   1. **Map/reduce**
   2. [**https://blog.csdn.net/dhaibo1986/article/details/108727249**](https://blog.csdn.net/dhaibo1986/article/details/108727249)
2. **Atomic**
   1. **原子类**
      1. **Integer**
      2. **Long**
   2. **AtomicBoolean**
   3. **Reference**
      1. **考察的比较多，原理和使用场景。Unsafe的方法，CAS**
   4. **FiledUpdater**

# 简历（5年以上）

## 切忌

精通，非常擅长，专家。

不要出现多线程，kafka，redis，hashmap，具体的技术名词。显得很low，很工程师。

不要出现跨界非常多的技术，tensorflow,hadoop，k8s，springboot，android。

不要太多管理性的东西。除非找纯管理的工作。

工作内容不要写，排期，协调，调用，根据产品经理，需求设计，开发。->工具人。

## 建议

了解，具有实践经验，了解xx体系的解决方案，并有实践落地的经验。

具备xxx样的能力…..层级设计、开发，具备xxx样规模的xx的系统，根据xx的迭代，流量，重构，支撑了xx的业务量，峰值qpsxxx

服务化，linux下的性能优化排障，全链路的优化排障，重构，解决在xxx的场景下的x的问题。

容器化，服务治理，容器编排，采用xx的流量调度策略，弹性伸缩。。。

自己整个github，将自己的过往的。。。

# github

**Github.com/noskywalker**

**QR 代码

描述已自动生成**