1. 互斥锁，自旋锁，原子操作，CAS

**互斥锁：**一个线程在执行A代码块的时候，另一个线程无法进入，旦另一个线程可以切换去做其他事情

**自旋锁：**同上，但另一个线程会自旋等待，不会线程切换做其他事情

当A代码块耗时少，自旋锁好，耗时多，互斥锁好

**原子操作：**一段指令能完成一系列功能，不可在分割。需要指令集支持

**CAS：**底层提供的一段原子操作 CAS(A,B,C) 如果A等于B，则把C赋值给A。 这段判断是原子性的。 （类似于乐观锁的应用）

1. 多线程单例模式实现

private static SingletonDemo instance = null;

//DCL (Double Check Lock 双端捡锁机制）

public static SingletonDemo getInstance() {

if (instance == null) {

synchronized (SingletonDemo.class) {

if (instance == null) {

instance = new SingletonDemo();

}

}

}

return instance;

}

问题：JVM会进行指令重排，可能会锁不住（指令重排只会保证串行语义执行的一致性（单线程），但并不会关心多线程间的语义一致性）

解决方案：使用volatile修饰instance，禁止指令重排

instance=newSingDemo()；可以分为以下3步完成（伪代码）

memory=allocate(): // 1.分配对象内存空间

instance(memory): // 2.初始化对象

instance=memory; //3. 设置instance指向刚分配的内存地址，此时instance != null

步骤2和步骤3不存在数据依赖关系，而且无论重排前还是重排后程序的执行结果在单线程中并没有改变，因此这种重排优化是允许的。

memory=allocate();//1．分配对象内存空间

instance=memory;//3．设置ins怡nce指向刚分配的内存地址，此时instance != null,但是对象还没有初始化完成！

instance(memory);//2．初始化对象

进程间通信

1.管道：速度慢，容量有限，只有父子进程能通讯

2.FIFO：任何进程间都能通讯，但速度慢

3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

4.信号量：不能传递复杂消息，只能用来同步

5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

6套接字（socket）：套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同设备间的进程通信。

7全双工管道：共享内存、信号量、消息队列、管道和命名管道只适用于本地进程间通信，套接字和全双工管道可用于远程通信，因此可用于网络编程。

线程间通信

　锁机制：包括互斥锁、条件变量、读写锁

　信号量机制（Semaphore）：包括无名进程信号量和命名线程信号量

　信号机制（Signal）：类似进程间的信号处理

Java的锁

Synchronized

1. 当synchronized作用在实例方法时，监视器锁（monitor）便是对象实例（this）；
2. 当synchronized作用在静态方法时，监视器锁（monitor）便是对象的Class实例，因为Class数据存在于永久代，因此静态方法锁相当于该类的一个全局锁；
3. 当synchronized作用在某一个对象实例时，监视器锁（monitor）便是括号括起来的对象实例；

Synchronized底层原理

Synchronized是通过对象内部的一个叫做 监视器锁（Monitor）来实现的 Monitor底层又是依赖操作系统的互斥锁，自旋锁实现的

1synchronized锁对象的升级(膨胀)过程主要如下:1.膨胀过程:无锁(锁对象初始化时)-> 偏向锁(有线程请求锁) -> 轻量级锁(多线程轻度竞争)-> 重量级锁(线程过多或长耗时操作,线程自旋过度... 自旋锁or互斥锁

Lock锁