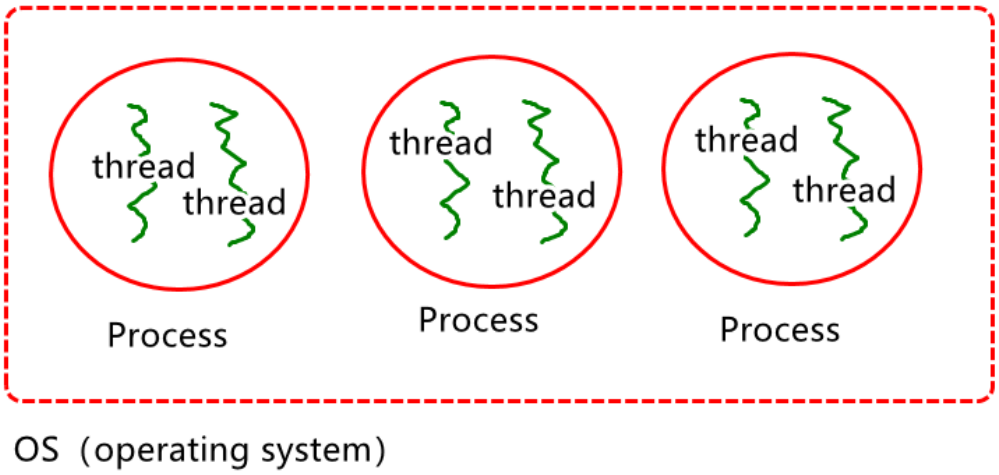
JAVA 核心基础增强
线程应用加强

1. 进程与线程认知强化

1.1. 如何理解进程与线程?

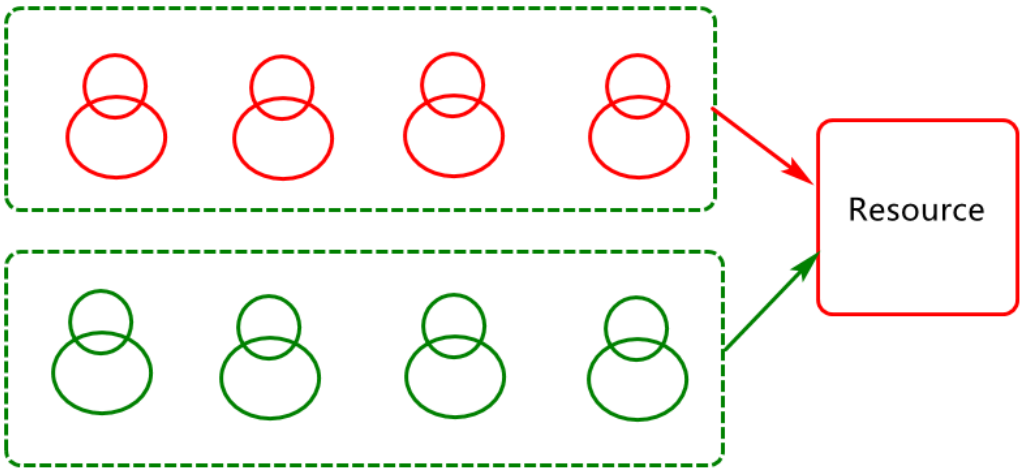
进程：操作系统进行资源调度和分配的基本单位（例如浏览器，APP，JVM）。
线程：进程中的最小执行单位，是 CPU 资源的分配的基本单位（可以理解为一个顺序的执行流）。



说明：多个线程可以共享所属进程的所有资源。

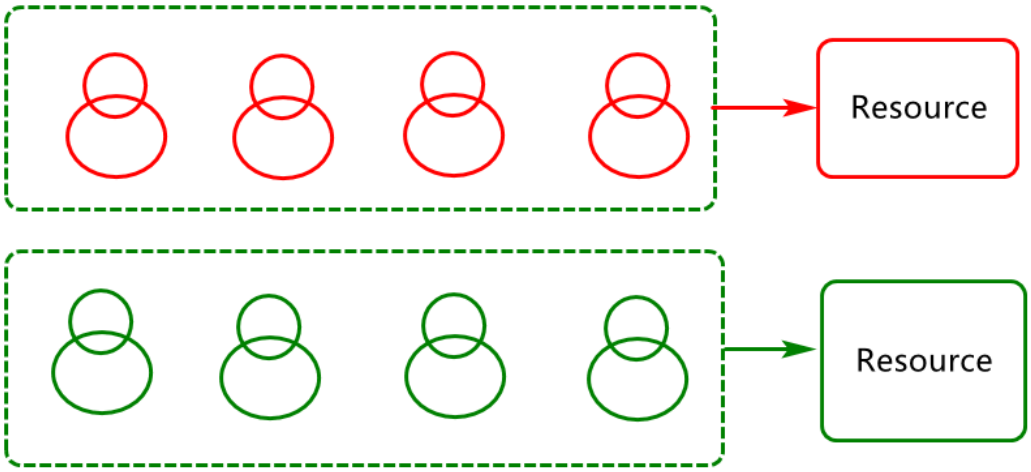
1.2. 如何理解多线程中的并发与并行?

并发：多线程抢占 CPU，可能不同时执行，侧重于多个任务交替执行。



现在的操作系统无论是 windows，linux 还是 macOS 等其实都是多用户多任务分时操作系统，使用这些操作系统的的用户可以“同时”干多件事情。但实际上，对于单机 CPU 的计算机而言，在同一时间只能干一件事，为了看起来像是“同时干多件事”分时操作系统把 CPU 的时间划分成了长短基本相同的时间区间，即“时间片”，通过操作系统的管理，把时间片依次轮流的分配给各个线程任务使用。我们看似的“同时干多件事”，其实是通过 CPU 时间片技术并发完成的。

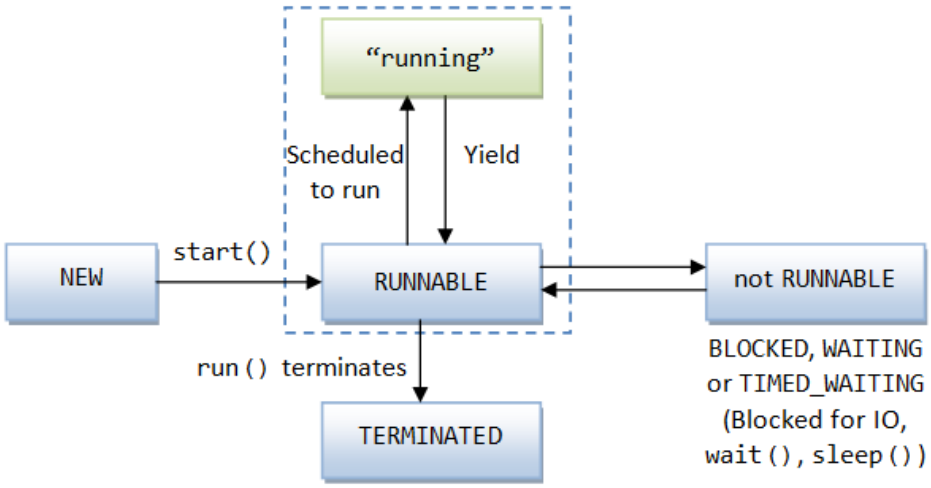
并行：线程可以不共享 CPU，可每个线程一个 CPU 同时执行多个任务。



总之：个人认为并行只出现在多 CPU 或多核 CPU 中，而并发可理解为并行中的一个子集。

1.3. 如何理解线程的生命周期及状态变化?

一个线程从创建，运行，到最后销毁的这个过程称之为线程的生命周期，在这个生命周期过程中线程可能会经历如下几个状态：



这些状态可归纳为：状态分别为新建状态，就绪状态，运行状态，阻塞状态，死亡状态。

2. 线程并发安全问题认知强化

2.1. 如何理解线程安全与不安全?

多个线程并发执行时，仍旧能够保证数据的正确性，这种现象称之为线程安全。
多个线程并发执行时，不能能够保证数据的正确性，这种现象称之为线程不安全。

案例 1：模拟多个线程同时执行售票操作

编写售票任务类：

```
class TicketTask implements Runnable{
      int ticket=10;
      @Override
```

```
        public void run() {
             doTicket();
        }
        public void doTicket() {
           while(true) {
               if(ticket<=0)break;
               System.out.println(ticket--);
           }
        }
    }
```

编写售票测试方法:

```
public static void main(String[] args) {
         TicketTask task=new TicketTask();
         Thread t1=new Thread(task);
         Thread t2=new Thread(task);
         Thread t3=new Thread(task);

         t1.start();
         t2.start();
         t3.start();
    }
```

案例 2：模拟多个线程同时执行计数操作

```
class Counter{
        private int count;
        public void doCount() {
            count++;
        }
}
```

2.2. 导致线程不安全的因素有哪些?

1. 多个线程并发执行。
2. 多个线程并发执行时存在共享数据集(临界资源)。
3. 多个线程在共享数据集上的操作不是原子操作(不可拆分的一个操作)

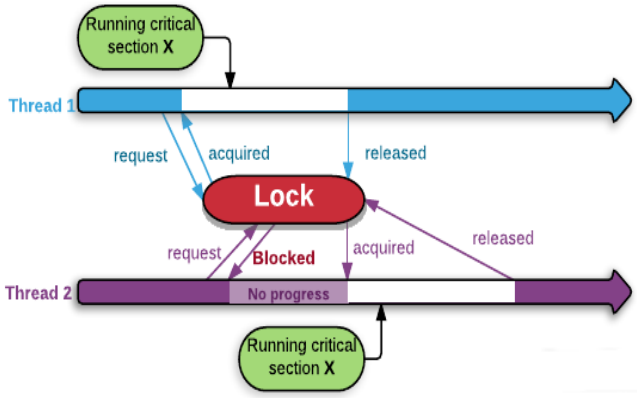
案例:

1. 设计一线程安全的计数器

2. 设计一线程安全的容器（Container）。

2.3. 如何保证并发线程的安全性?

1. 对共享进行限制(阻塞)访问（例如加锁：syncronized，Lock）:多线程在同步方法或同步代码块上排队执行。



2. 基于 CAS(比较和交换)实现非阻塞同步（基于 CPU 硬件技术支持）
   a) 内存地址(V)
   b) 期望数据值(A)
   c) 需要更新的值(B)

   CAS 算法支持无锁状态下的并发更新，但可能会出现 ABA 问题，长时间自旋问题。

3. 取消共享，每个线程一个对象实例（例如 threadlocal）
   a) Connection 允许多线程共享吗？(不允许，每个线程一个)
   b) SimpleDateFormat 允许多线程共享吗？（不允许，每个线程一个）
   c) SqlSession 对象允许共享吗?(不允许，每个线程一个)

说明：Java 中的线程安全问题的主要关注点有 3 个：可见性，有序性，原子性；Java 内存模型（JMM）解决了可见性和有序性问题，而锁解决了原子性问题。

2.4. Synchronized 关键字应用及原理分析?

1. synchronized 简介:

1）synchronized 是排它锁的一种实现，支持可重入性。

2）基于这种机制可以实现多线程在共享数据集上同步(互斥和协作)。

2.1) 互斥：多线程在共享数据集上排队执行。

2.2) 协作：多线程在共享数据集上进行协作执行.(通讯)

说明:

排他性：如果线程 T1 已经持有锁 L，则不允许除 T1 外的任何线程 T 持有该锁 L

重入性：如果线程 T1 已经持有锁 L，则允许线程 T1 多次获取锁 L，更确切的说，获取一次后，可多次进入锁。

2 Synchronized.应用分析:

1） 修饰方法：同步方法（锁为当前实例或 Class 对象）

1.1） 修饰静态方法：默认使用的锁为方法所在类的 Class 对象

1.2） 修饰实例方法：默认使用的所为方法所在类的实例对象

2） 修饰代码块：同步代码块（代码块括号内配置的对象）

3. Synchronized 原理分析：基于 Monitor 对象实现同步。

1）同步代码块采用 monitorenter、monitorexit 指令显式的实现。

2）同步方法则使用 ACC_SYNCHRONIZED 标记符隐式的实现。

4. Synchronized 锁优化：底层

为了减少获得锁和释放锁带来的性能消耗，JDK1.6 以后的锁一共有 4 种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。

说明：锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。

2.5. 如何理解 volatile 关键字的应用?

1.定义：volatile 一般用于修饰属性变量

1) 保证共享变量的可见性.(尤其是多核或多 cpu 场景下)

2) 禁止指令的重排序操作（例如：count++底层会有三个步骤）
3) 不保证原子性(例如不能保证一个线程执行完 count++所有指令其它线程才能执行。)

2.应用场景分析：

1) 状态标记（boolean 类型属性）
2) 安全发布 (线程安全单例中的对象安全发布-双重检测机制)
3) 读写锁策略（一个写，并发读，类似读写锁）

3.代码实现分析：

1. 状态标记代码示例

```
class Looper{
    private volatile boolean isStop;
    public void loop() {
        for(;;) {
            if(isStop)break;
        }
    }
    public void stop() {
        isStop=true;
    }
}
```

```
public class TestVolatile01 {
    public static void main(String[] args)throws Exception {
    Looper looper=new Looper();
        Thread t1=new Thread() {
            public void run() {
                looper.loop();
            };
        };
        t1.start();
        t1.join(1000);
        looper.stop();
    }
}
```

2. 安全发布代码示例

```
class Singleton{
```

```
    private Singleton() {}
    private static volatile  Singleton  instance;
    public static Singleton getSingleton() {//大对象，稀少用
        if(instance==null) {
            synchronized (Singleton.class) {
                instance=new Singleton();//分配空间，属性初始化，instance赋值
            }
        }
        return instance;
    }
}
```

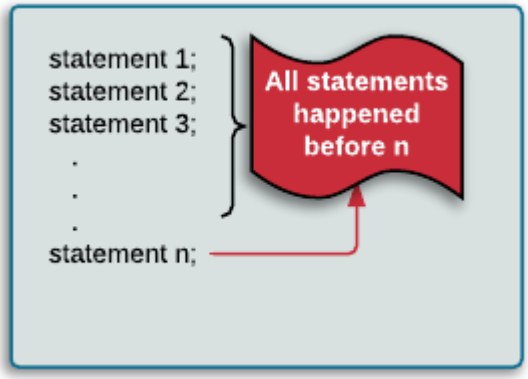
3. 读写锁应用案例：

```
class Counter{
    private volatile int count;
    public int getCount() {//read
        return count;
    }
    public synchronized void doCount() {//write
        count++;
    }
}
```

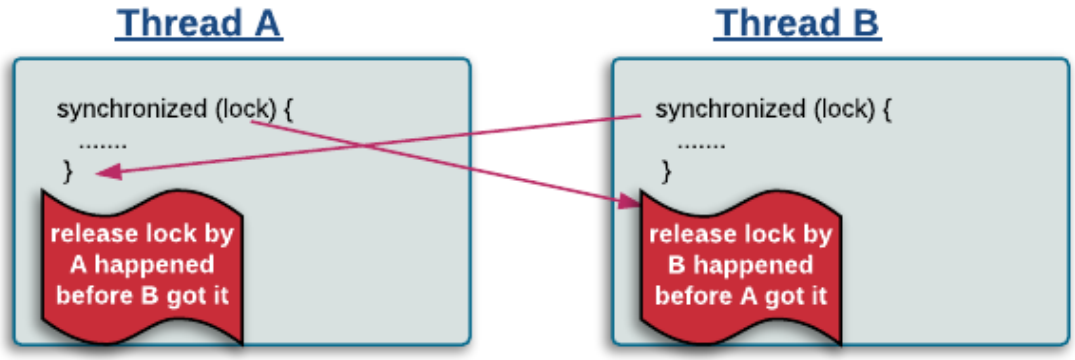
2.6. 如何理解 happen-before 原则应用?

在 JMM 中如果一个操作 A Happened-bofore 另一个操作 B，那么 A 操作的结果对 B 操作的结果是可见的,那么我们称这种方式为 happened-before 原则，例如

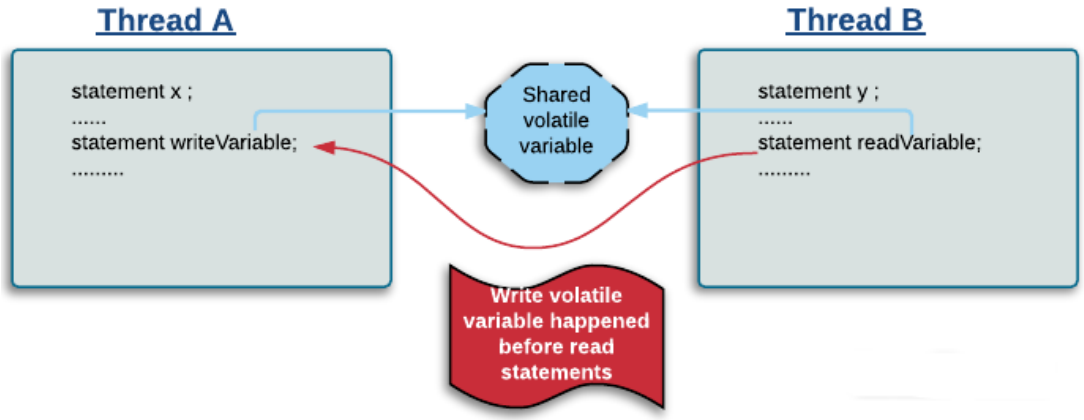
1. Single thread rule:



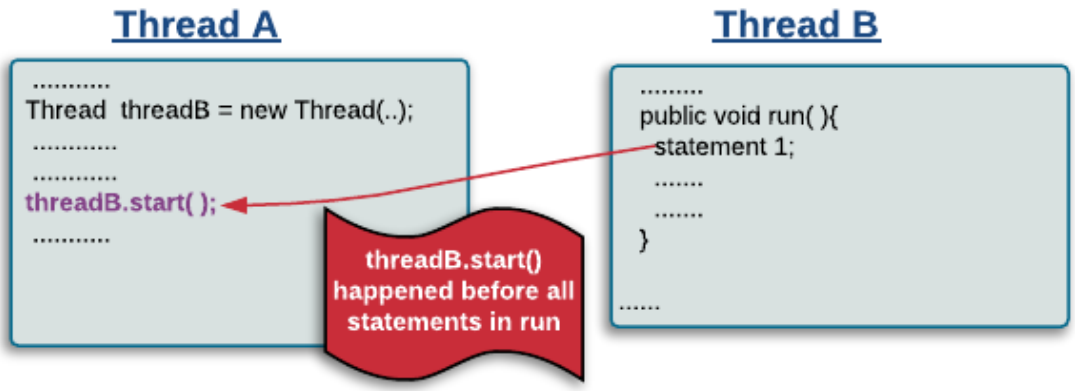
2. Monitor lock rule



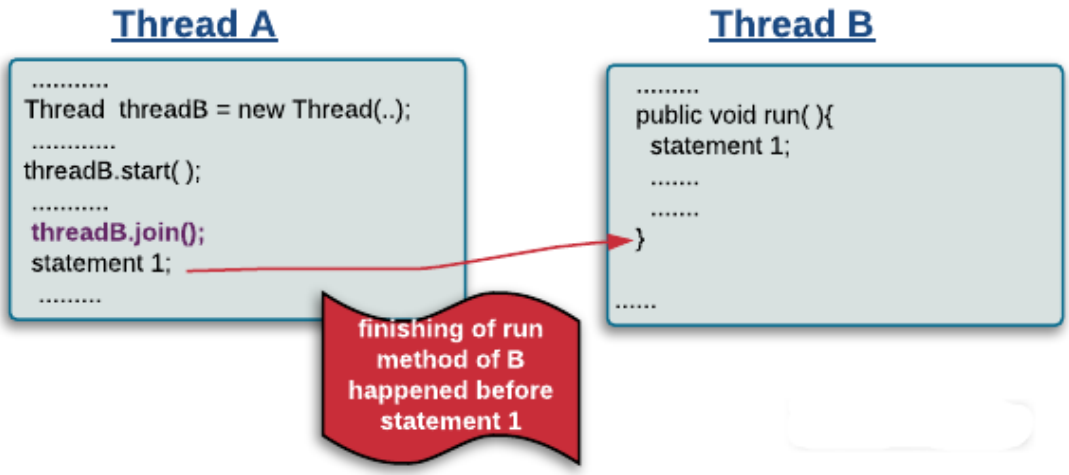
3. Volatile variable rule



4. Thread start rule



5. Thread join rule



说明：JMM 中基于 happened-before 规则，判定数据是否存在竞争，线程是否安全，以及多线程环境下变量值是否是可见的。

2.7. 如何理解 JAVA 中的悲观锁和乐观锁?

JAVA 中为了保证多线程并发访问的安全性，提供了基于锁的应用，大体可归纳为两大类，即悲观锁和乐观锁。

悲观锁&乐观锁定义说明：

1) 悲观锁：假定会发生并发冲突，屏蔽一切可违反数据完整性的操作.同一时刻只能有一个线程执行写操作。
例如 java 中可以基于 syncronized,Lock，ReadWriteLock 等实现。

2) 乐观锁：假设不会发生冲突，只在提交操作时检查是否违反数据完整性.
多个线程可以并发执行写操作但只能有一个线程写操作成功。
例如 java 中可借助 CAS（Compare And Swap）算法实现(此算法依赖硬件 CPU)。

悲观锁&乐观锁应用场景说明：

1)悲观锁适合写操作比较多的场景，写可以保证写操作时数据正确。
2)乐观锁适合读操作比较多的场景，不加锁的特点能够使其读操作的性能大幅提升

悲观锁&乐观锁应用案例分析

悲观锁实现计数器：
方案 1：

```
class Counter{
        private int count;
        public synchronized int count() {
            count++;
            return count;
        }
}
```

方案 2：

```
class Counter{
        private int count;
        private Lock lock=new ReentrantLock();
        public  int count() {
            lock.lock();
            try {
             count++;
            return count;
            }finally {
             lock.unlock();
            }
        }
    }
```

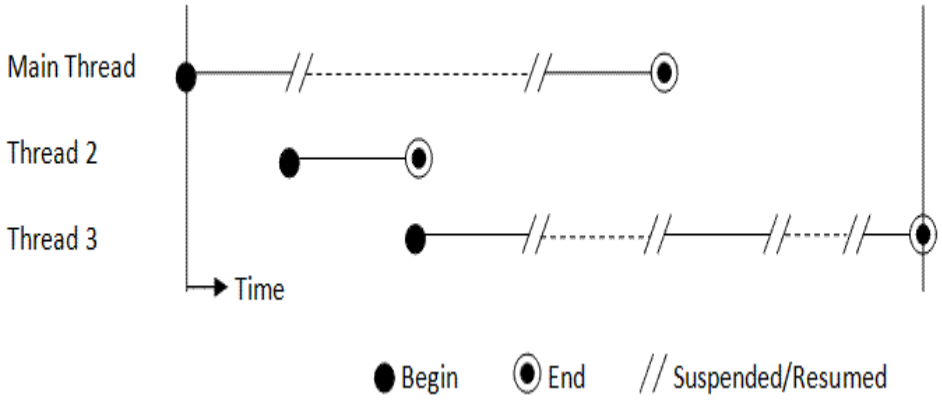
乐观锁实现计数器：

```
class Counter{
    private AtomicInteger at=new AtomicInteger();//CAS
    public  int count() {
        return at.incrementAndGet();
    }
}
```

其中 AtomicInteger 是基于 CAS 算法实现。

2.8. 如何理解线程的上下文切换?

一个线程得到 CPU 执行的时间是有限的。当此线程用完为其分配的 CPU 时间以后，cpu 会切换到下一个线程执行。例如：



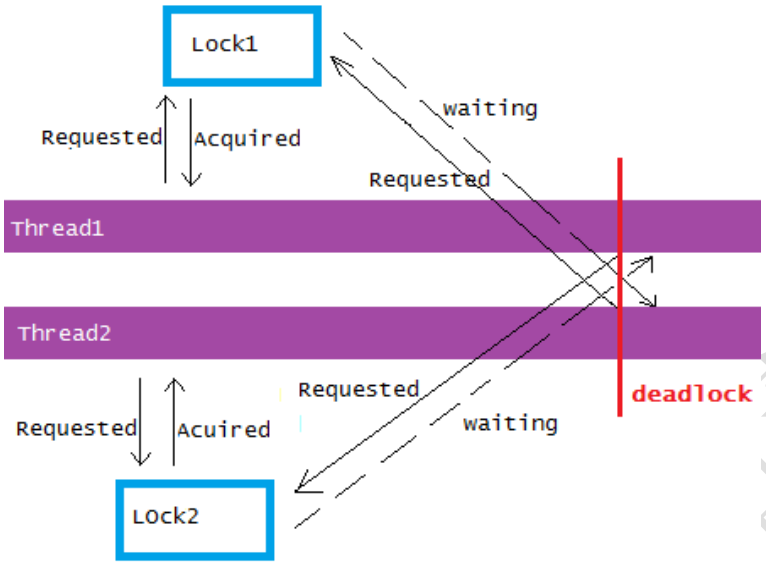
在线程切换之前，线程需要将当前的状态进行保存，以便下次再次获得 CPU 时间片时可以加载对应的状态以继续执行剩下的任务。而这个切换过程是需要耗费时间的，会影响多线程程序的执行效率，所以在在使用多线程时要减少线程的频繁切换。那如何实现呢?

减少多线程上下文切换的方案如下：

1) 无锁并发编程：锁的竞争会带来线程上下文的切换
2) CAS 算法：CAS 算法在数据更新方面，可以达到锁的效果
3) 使用最少线程：避免不必要的线程等待
4) 使用协程：单线程完成多任务的调度和切换，避免多线程

2.7.如何理解死锁以及避免死锁问题?

多个线程互相等待已经被对方线程正在占用的锁,导致陷入彼此等待对方释放锁的状态，这个过程称之为死锁，如图所示:



死锁案例分析-1:

可能出现死锁的案例分享

```
class SyncTask01 implements Runnable {
    private Object obj1;
    private Object obj2;
    public SyncTask01(Object o1, Object o2) {
        this.obj1 = o1;
        this.obj2 = o2;
    }
    @Override
    public void run() {
        synchronized (obj1) {
            work();
            synchronized (obj2) {
                work();
            }
        }
    }
    private void work() {
        try {Thread.sleep(30000);} catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

死锁测试

```
public class TestDeadLock01 {
    public static void main(String[] args)throws Exception {
         Object obj1 = new Object();
         Object obj2 = new Object();
         Thread t1 = new Thread(new SyncTask01(obj1, obj2), "t1");
         Thread t2 = new Thread(new SyncTask01(obj2, obj1), "t2");
         t1.start();
         t2.start();
    }
}
```

死锁案例分析-2：

```
class SyncTask02 implements Runnable{
    private List<Integer> from;
    private List<Integer> to;
    private Integer target;
    public SyncTask02(List<Integer> from,List<Integer> to,Integer target) {
        this.from=from;
        this.to=to;
        this.target=target;
    }
    @Override
    public void run() {
        moveListItem(from, to, target);
    }
     private static void moveListItem (List<Integer> from,
                    List<Integer> to, Integer item) {
       log("attempting lock for list", from);
       synchronized (from) {
          log("lock acquired for list", from);
           try {
               TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            log("attempting lock for list ", to);
            synchronized (to) {
              log("lock acquired for list", to);
               if(from.remove(item)){
```

```
                to.add(item);
            }
            log("moved item to list ", to);
        }
    }
    private static void log (String msg, Object target) {
        System.out.println(Thread.currentThread().getName() +
                                    ": " + msg + " " +
                            System.identityHashCode(target));
    }
}
```

```
public class TestDeadLock02 {

    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>(Arrays.asList(2, 4, 6, 8, 10));
        List<Integer> list2 = new ArrayList<>(Arrays.asList(1, 3, 7, 9, 11));

        Thread thread1 = new Thread(new SyncTask02(list1, list2, 2));
        Thread thread2 = new Thread(new SyncTask02(list2, list1, 9));

        thread1.start();
        thread2.start();
    }
}
```

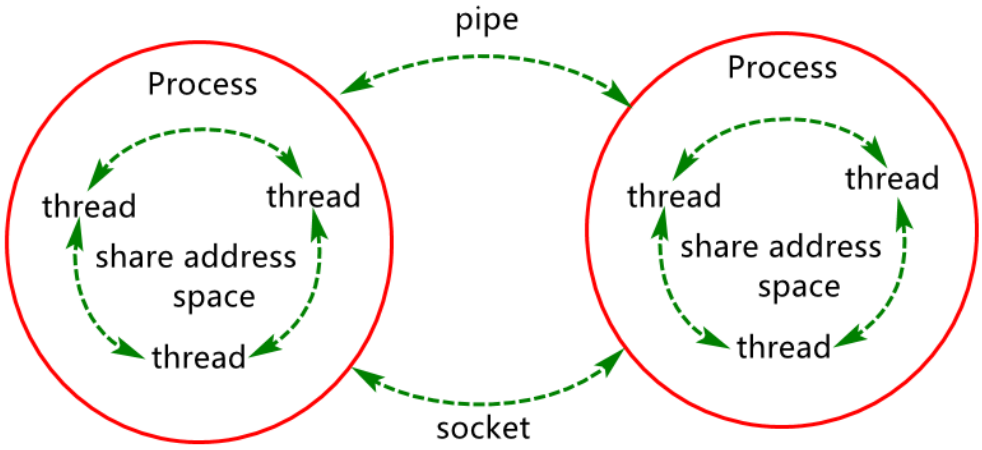
如何避免死锁呢?

1) 避免一个线程中同时获取多个锁
2) 避免一个线程在一个锁中获取其他的锁资源
3) 考虑使用定时锁来替换内部锁机制，如 lock.tryLock(timeout)。

3. 线程通讯与进程通讯应用基础

3.1. 如何理解进程与线程通讯?

线程通讯：java 中的多线程通讯主要是共享内存（变量）等方式。
进程通讯：java 中进程通讯（IPC）主要是 Socket，MQ 等。



3.2. 如何实现进程内部线程之间的通讯?

3.2.1. 基于 wait/nofity/notifyall 实现

1. wait()/notify()/notifyall（）方法定义说明：

1) Wait:阻塞正在使用监视器对象的线程，同时释放监视器对象
2) notify: 唤醒在监视器对象上等待的单个线程，但不释放监视器对象，此时调用该方法的代码继续执行，直到执行结束才释放对象锁
3) notifyAll: 唤醒在监视器对象上等待的所有线程，但不释放监视器对象，此时调用该方法的代码继续执行，直到执行结束才释放对象锁

2. wait()/notify()/notifyall（）方法应用说明

1) 这些方法必须应用在同步代码块或同步方法中

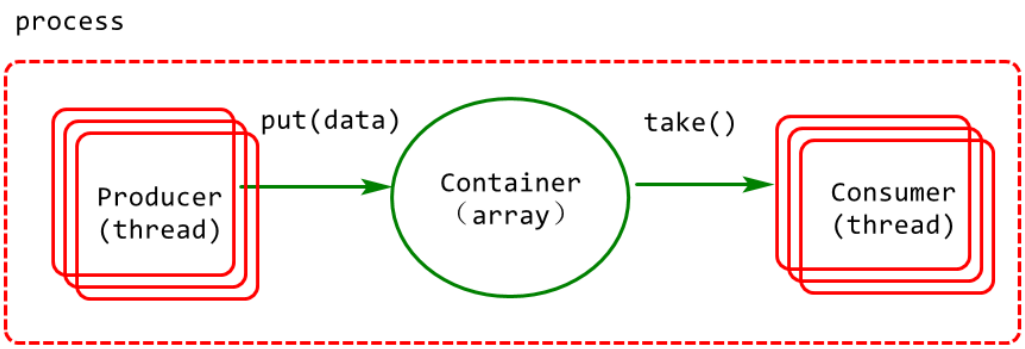
2） 这些方法必须由监视器对象（对象锁）调用

说明：使用 wait/notify/notifyAll 的作用一般是为了避免轮询带来的性能损失。

3. wait()/notify()/notifyall()应用案例实现：

手动实现阻塞式队列，并基于 wait()/notifyAll()方法实现实现线程在队列上的通讯。

案例：现有一生产者消费者模型，生产者和消费者并发操作容器对象。



代码实现：实现一线程安全的容器类(基于数组实现一个阻塞式队列)

```
/**
 * 有界消息队列：用于存取消息
 * 1)数据结构：数组(线性结构)
 * 2)具体算法：FIFO(先进先出)-First in First out
 */
public class BlockContainer<T> {//类泛型
    /**用于存储数据的数组*/
    private Object[] array;
    /**记录有效元素个数*/
    private int size;
    public BlockContainer () {
        this(16);//this(参数列表)表示调用本类指定参数的构造函数
    }
    public BlockContainer (int cap) {
        array=new Object[cap];//每个元素默认值为null
    }
}
```

向容器添加put方法，用于放数据。

```
/**
 * 生产者线程通过put方法向容器放数据
 * 数据永远放在size位置
 * 说明：实例方法内部的this永远指向
 * 调用此方法的当前对象(当前实例)
 * 注意：静态方法中没有this，this只能
 * 应用在实例方法，构造方法，实例代码块中
 */
public synchronized void put(T t){//同步锁：this
//1.判定容器是否已满，满了则等待
while(size==array.length)
try{this.wait();}catch(Exception e){}
//2.放数据
array[size]=t;
//3.有效元素个数加1
size++;
//4.通知消费者取数据
this.notifyAll();
}
```

向容器类添加take方法，用于从容器取数据。

```
/**
 * 消费者通过此方法取数据
 * 位置：永远取下标为0的位置的数据
 * @return
 */
@SuppressWarnings("unchecked")
public synchronized T take(){
//1.判定容器是否为空，空则等待
while(size==0)
try{this.wait();}catch(Exception e){}
//2.取数据
Object obj=array[0];
//3.移动元素
System.arraycopy(
        array,//src 原数组
        1, //srcPos 从哪个位置开始拷贝
        array,  //dest 放到哪个数组
        0, //destPost 从哪个位置开始放
        size-1);//拷贝几个
//4.有效元素个数减1
size--;
```

```
    //5.将size位置为null
    array[size]=null;
    //6.通知生产者放数据
    this.notifyAll();//通知具备相同锁对象正在wait线程
    return (T)obj;
    }
```

3.2.2. 基于 Condition 实现

1. Condition 类定义说明

Condition 是一个用于多线程间协同的工具类，基于此类可以方便的对持有锁的线程进行阻塞或唤醒阻塞的线程。它的强大之处在于它可以为多个线程间建立不同的 Condition，通过 signal()/signalall()方法指定要唤醒的不同线程。

2. Condition 类应用说明

1) 基于 Lock 对象获取 Condition 对象
2) 基于 Condition 对象的 await()/signal()/signalall()方法实现线程阻塞或唤醒。

3. Condition 类对象的应用案例实现：

手动实现阻塞式队列，并基于 wait()/notifyAll()方法实现实现线程在队列上的通讯。

```
/**
 * 有界消息队列：用于存取消息
 * 1)数据结构：数组(线性结构)
 * 2)具体算法：FIFO(先进先出)-First in First out
 */
public class BlockContainer<T> {//类泛型

    /**用于存储数据的数组*/
    private Object[] array;
    /**记录有效元素个数*/
    private int size;
    public BlockContainer() {
        this(16);//this(参数列表)表示调用本类指定参数的构造函数
    }
```

```
    public BlockContainer(int cap) {
        array=new Object[cap];//每个元素默认值为null
    }
    //JDK1.5以后引入的可重入锁(相对于synchronized灵活性更好)
    private ReentrantLock lock=new ReentrantLock(true);// true表示使用公平锁，
默认是非公平锁
    private Condition producerCondition=lock.newCondition();//通讯条件
    private Condition consumerCondition=lock.newCondition();//通讯条件
}
```

向容器中添加put方法，用于向容器放数据

```
    /**
     * 生产者线程通过put方法向容器放数据
     * 数据永远放在size位置
     * 说明：实例方法内部的this永远指向
     * 调用此方法的当前对象(当前实例)
     * 注意：静态方法中没有this，this只能
     * 应用在实例方法，构造方法，实例代码块中
     */
    public void put(T t){//同步锁：this
    System.out.println("put");
    lock.lock();
    try{
    //1.判定容器是否已满，满了则等待
    while(size==array.length)
        //等效于Object类中的wait方法
    try{producerCondition.await();}catch(Exception e){e.printStackTrace();}
    //2.放数据
    array[size]=t;
    //3.有效元素个数加1
    size++;
    //4.通知消费者取数据
    consumerCondition.signalAll();//等效于object类中的notifyall()
    }finally{
    lock.unlock();
    }
    }
```

在容器类中添加take方法用于从容器取数据

```
    /**
     * 消费者通过此方法取数据
     * 位置：永远取下标为0的位置的数据
     * @return
```

```
     */
    @SuppressWarnings("unchecked")
    public  T take(){
    System.out.println("take");
    lock.lock();
    try{
    //1.判定容器是否为空，空则等待
    while(size==0)
    try{consumerCondition.await();}catch(Exception e){}
    //2.取数据
    Object obj=array[0];
    //3.移动元素
    System.arraycopy(
            array,//src 原数组
            1, //srcPos 从哪个位置开始拷贝
            array,  //dest 放到哪个数组
            0, //destPost 从哪个位置开始放
            size-1);//拷贝几个
    //4.有效元素个数减1
    size--;
    //5.将size位置为null
    array[size]=null;
    //6.通知生产者放数据
       producerCondition.signalAll();//通知具备相同锁对象正在wait线程
    return (T)obj;
    }finally{
    lock.unlock();
    }
    }
```

3.3. 如何实现进程之间间通讯（IPC）？

3.3.1. 基于 socket 实现进程间通讯?

基于 BIO 实现的简易 server 服务器

```
public class BioMainServer01 {
    private Logger log=LoggerFactory.getLogger(BioMainServer01.class);
```

```
    private ServerSocket server;
    private volatile boolean isStop=false;
    private int port;
    public BioMainServer01(int port) {
        this.port=port;
    }
    public void doStart()throws Exception {
        server=new ServerSocket(port);
        while(!isStop) {
            Socket socket=server.accept();
            log.info("client connect");
            doService(socket);
        }
        server.close();
    }
    public void doService(Socket socket) throws Exception{
        InputStream in=socket.getInputStream();
        byte[] buf=new byte[1024];
        int len=-1;
        while((len=in.read(buf))!=-1) {
            String content=new String(buf,0,len);
            log.info("client say {}", content);
        }
        in.close();
        socket.close();
    }
    public void doStop() {
        isStop=false;
    }
    public static void main(String[] args)throws Exception {
        BioMainServer01 server=new BioMainServer01(9999);
        server.doStart();
    }
}
```


启动服务，然后打开浏览器进行访问或者通过如下客户端端访问

```
public class BioMainClient {
    public static void main(String[] args) throws Exception{
        Socket socket=new Socket();
        socket.connect(new InetSocketAddress("127.0.0.1", 9999));
        OutputStream out=socket.getOutputStream();
        Scanner sc=new Scanner(System.in);
        System.out.println("client input:");
```

```
        out.write(sc.nextLine().getBytes());
        out.close();
        sc.close();
        socket.close();
    }
}
```

4. 课后练习与加强

4.1. 线程同步应用练习

1. 基于链表结构实现一个线程安全的阻塞队列?

4.2. 线程通讯练习

1. 基于 BIO 方式实现 Socket 跨进程通讯。