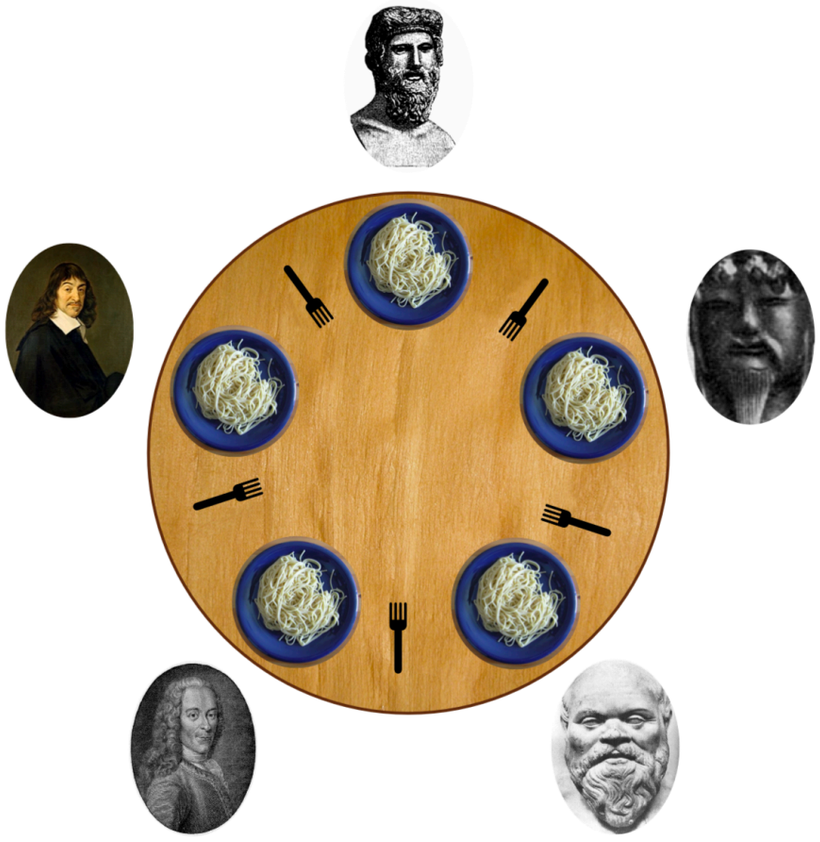
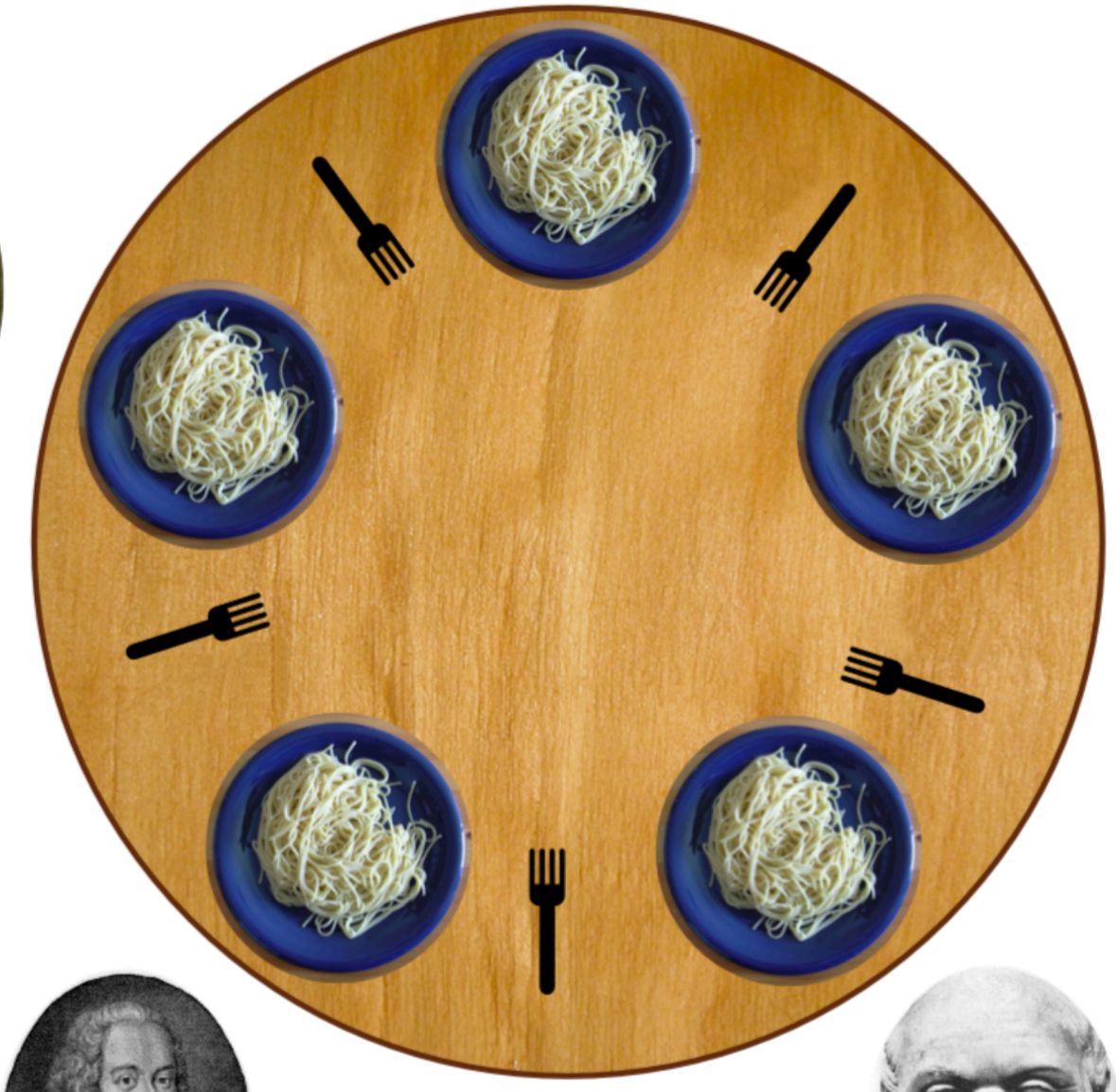
并发与多线程

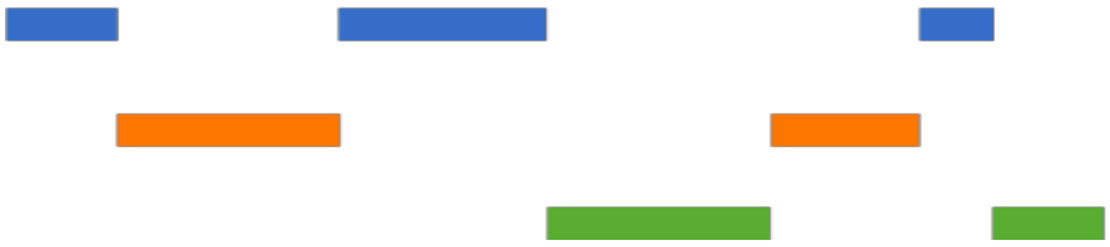
杨亮



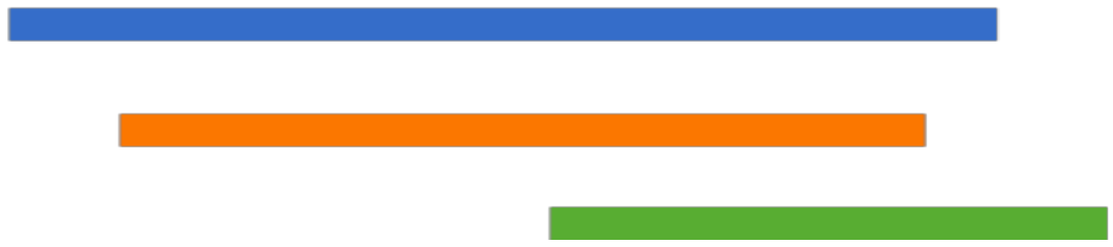
哲学家就餐问题



Tasks start, run and complete in an interleaved fashion

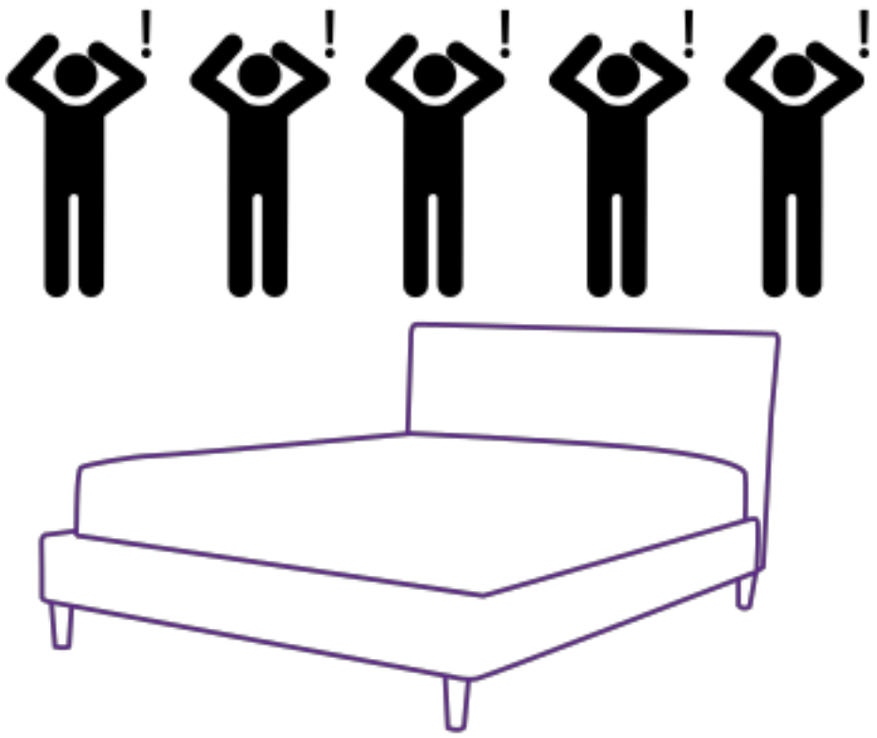


Tasks run simultaneously



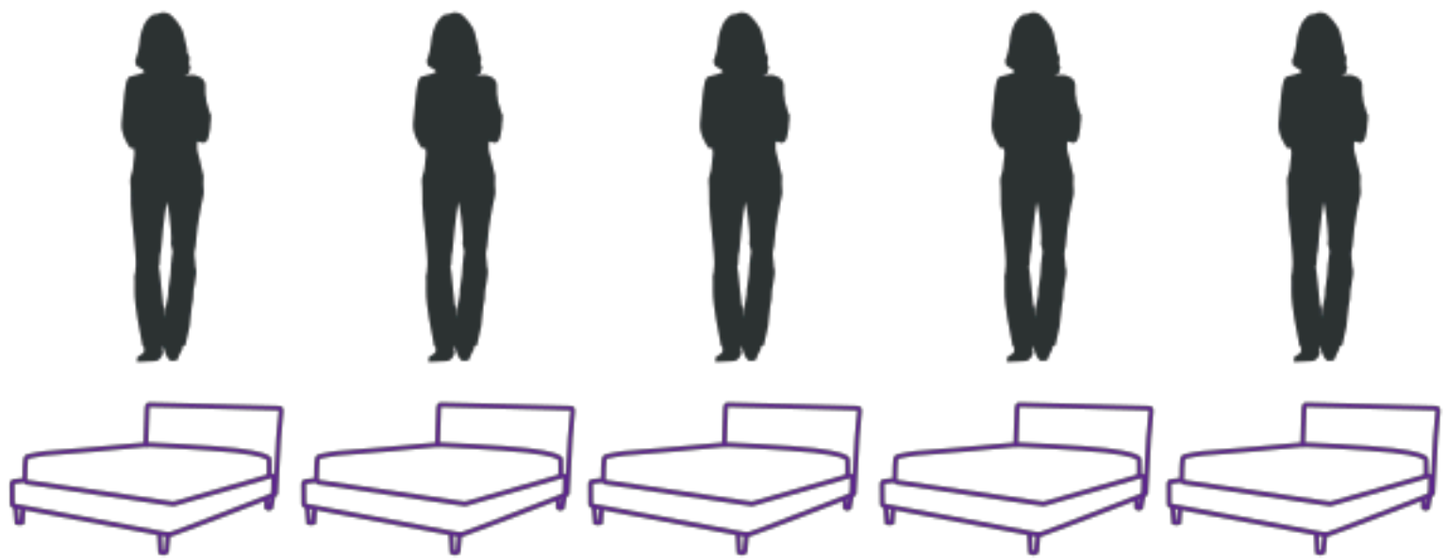
Concurrency

Complex co-ordination

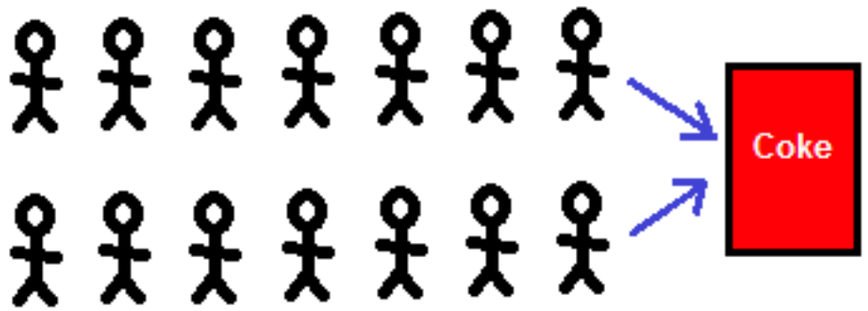


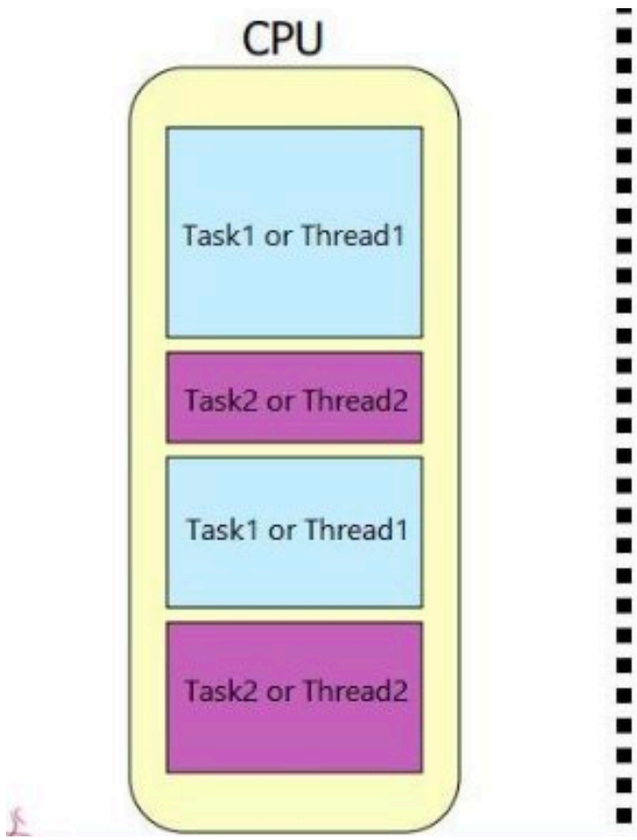
Parallelism

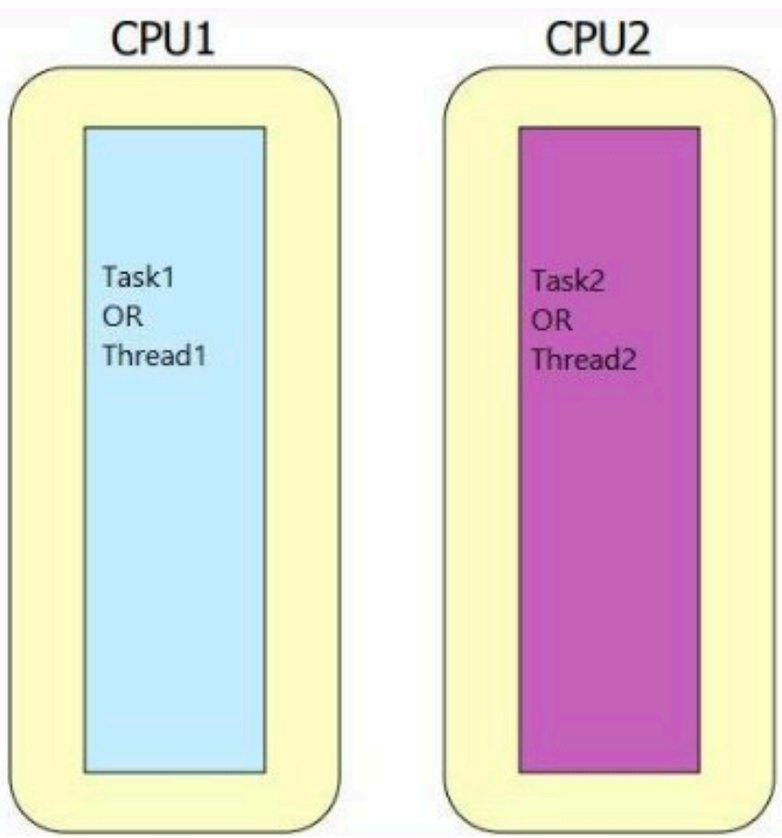
Simple co-ordination



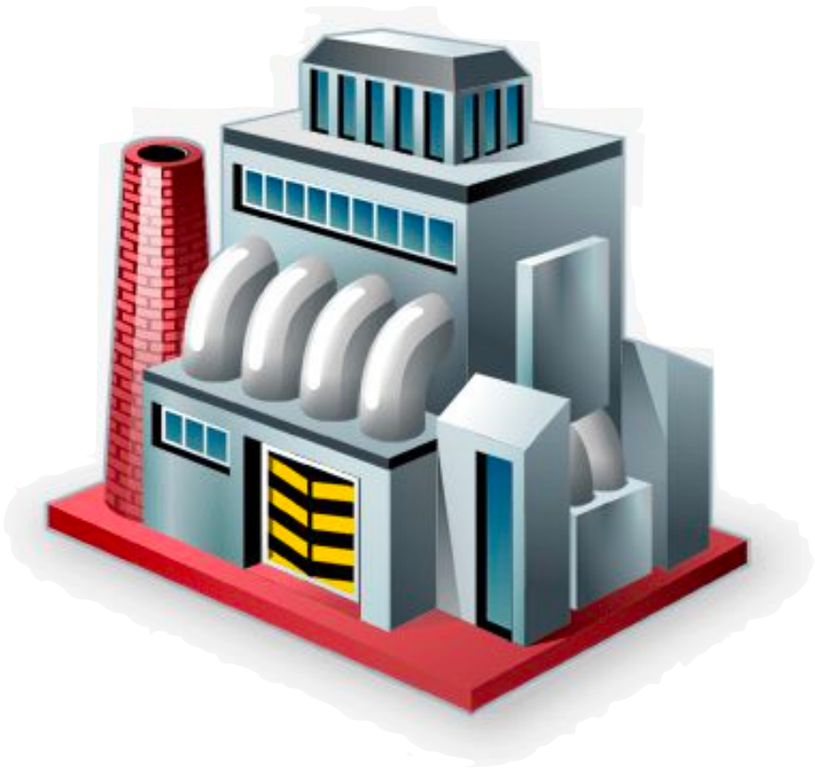
并发与并行


Coke
Concurrent: 2 queues, 1 vending machine
Coke
Coke
Parallel: 2 queues, 2 vending machines


CPU
Task1 or Thread1
Task2 or Thread2
Task1 or Thread1
Task2 or Thread2


CPU1
Task1
OR
Thread1
CPU2
Task2
OR
Thread2

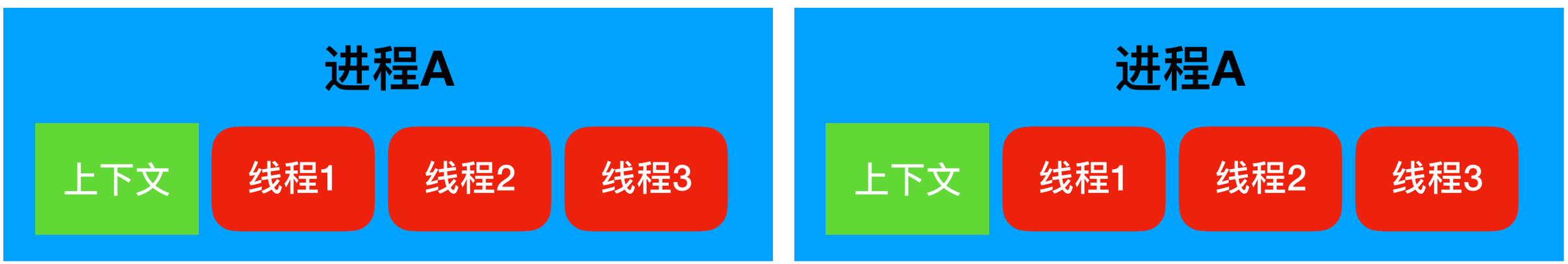
资源与调度



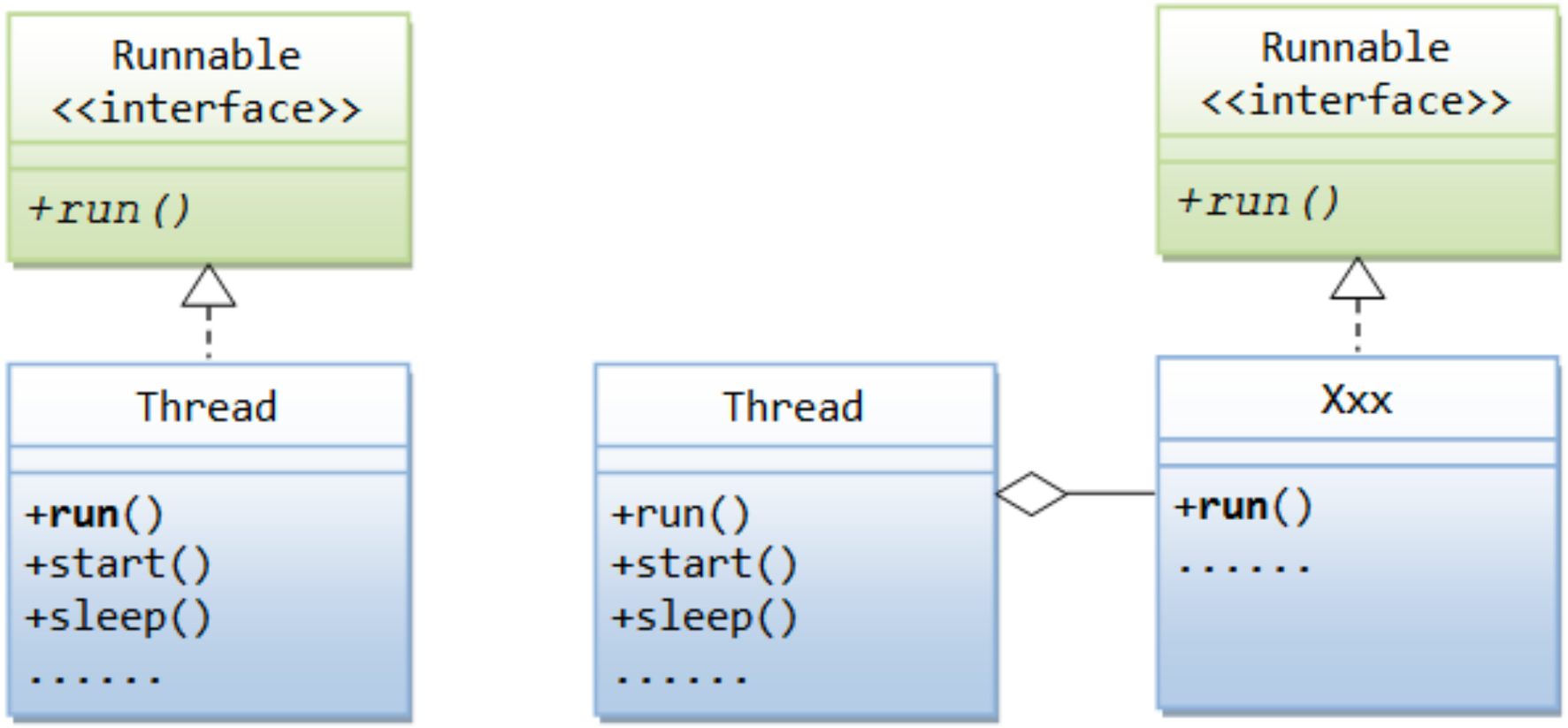
进程与线程

进程（Process）是进行资源分配和保护的基本单位

线程（Thread）处理器调度和分派的基本单位



Java中的多线程




```
public interface Runnable
{
    public abstract void run();
}
```

线程类 Thread

```
public class Thread extends Object implements Runnable
{
    public Thread()                          //构造方法
    public Thread(String name)               //name指定线程名
    public Thread(Runnable target)           //target指定线程的目标对象
    public Thread(Runnable target, String name)
    public void run()                        //描述线程操作的线程体
    public final String getName()            //返回线程名
    public final void setName(String name)   //设置线程名
    public static int activeCount()          //返回当前活动线程个数
    public static Thread currentThread()     //返回当前执行线程对象
    public Sting toString()   //返回线程的字符串信息
    public void start()                      //启动已创建的线程对象
}
```

```java
class HeavyWorkRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
            //Get database connection, delete unused data from DB
            doDBProcessing();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());
    }

    private void doDBProcessing() throws InterruptedException {
        Thread.sleep(5000);
    }
}
```

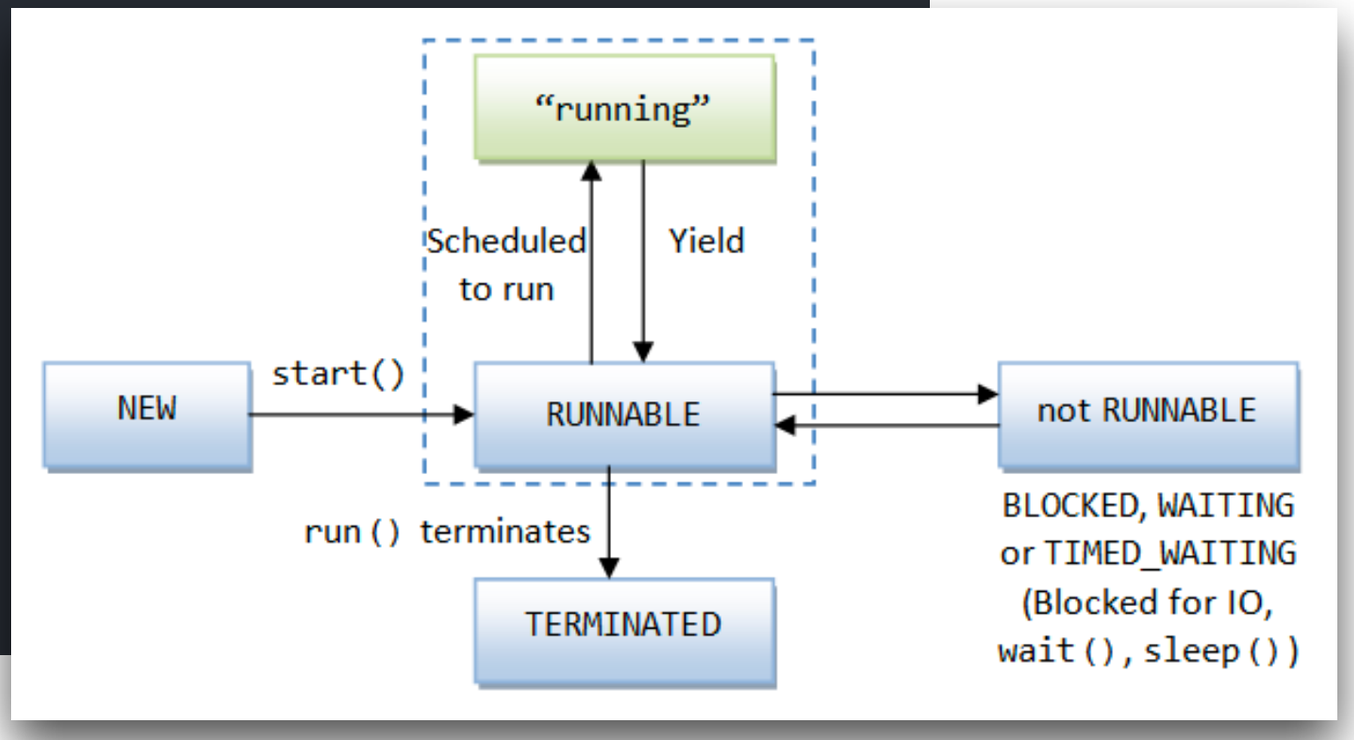
```java
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("MyThread - START "+Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
            //Get database connection, delete unused data from DB
            doDBProcessing();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("MyThread - END "+Thread.currentThread().getName());
    }

    private void doDBProcessing() throws InterruptedException {
        Thread.sleep(5000);
    }
}
```

创建进发程的方法

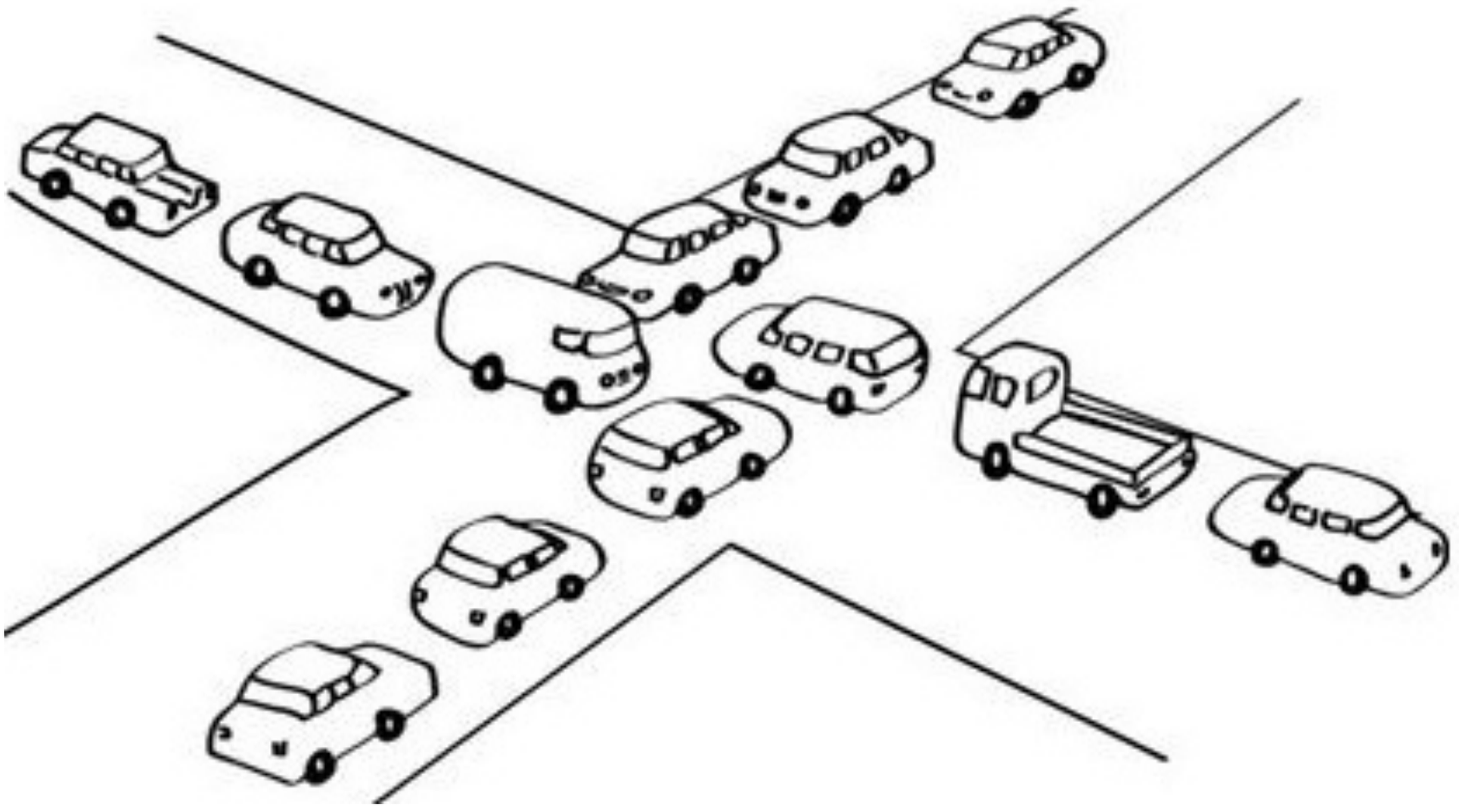
```
public class ThreadRunExample {

    public static void main(String[] args){
        Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");
        Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");
        System.out.println("Starting Runnable threads");
        t1.start();
        t2.start();
        System.out.println("Runnable Threads has been started");
        Thread t3 = new MyThread("t3");
        Thread t4 = new MyThread("t4");
        System.out.println("Starting MyThreads");
        t3.start();
        t4.start();
        System.out.println("MyThreads has been started");

    }
}
```

线程的状态 v1

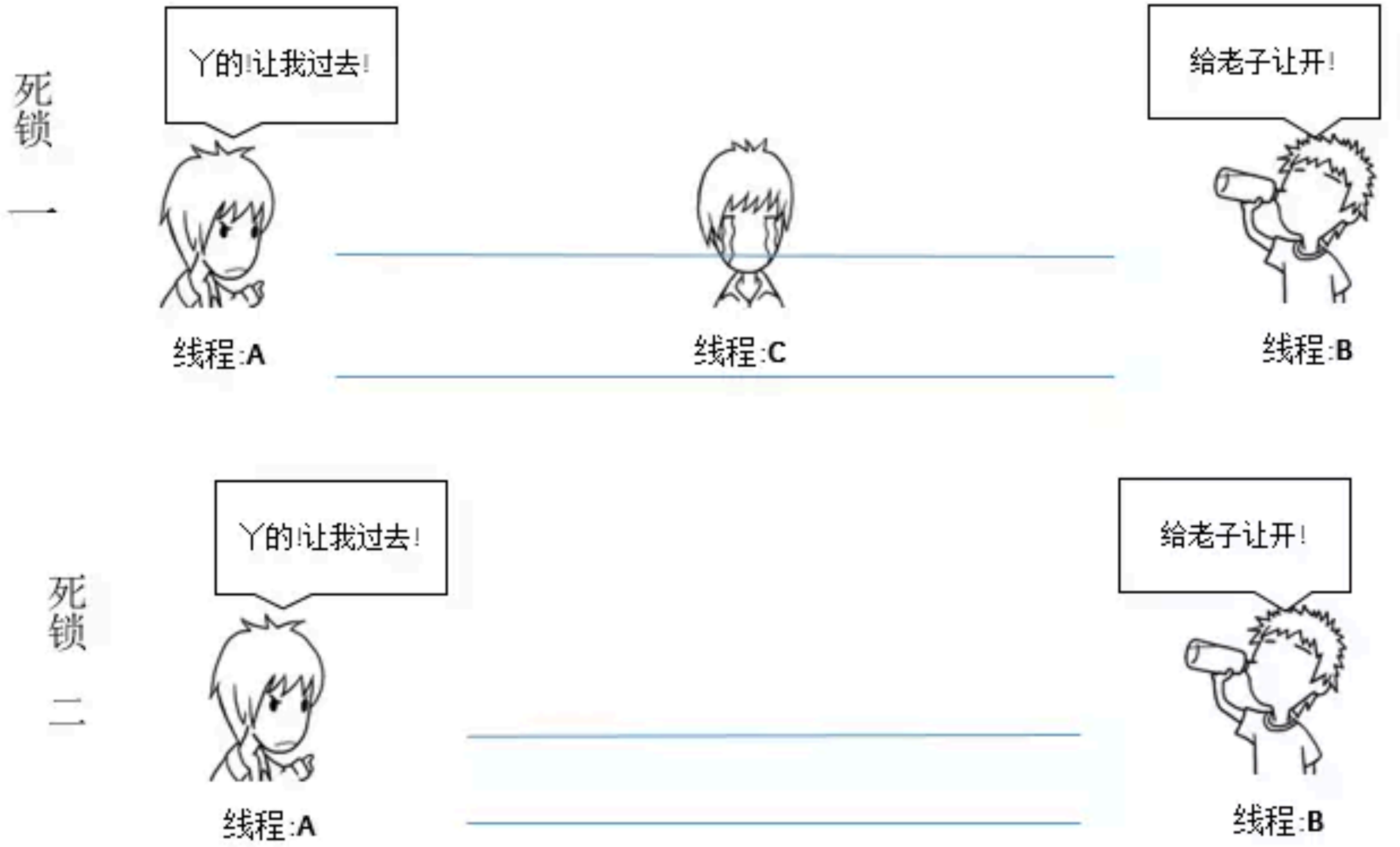
```java
public class YieldTest extends Thread {

	public YieldTest(String name) {
		super(name);
	}

	@Override
	public void run() {
		for (int i = 1; i <= 50; i++) {
			System.out.println("" + this.getName() + "-----" + i);
			// 当i为30时，该线程就会把CPU时间让掉，让其他或者自己的线程执行（也就是谁先抢到谁执行）
			if (i == 30) {
				this.yield();
			}
		}
	}

	public static void main(String[] args) {
		YieldTest yt1 = new YieldTest("张三");
		YieldTest yt2 = new YieldTest("李四");
		yt1.start();
		yt2.start();
	}
}
```



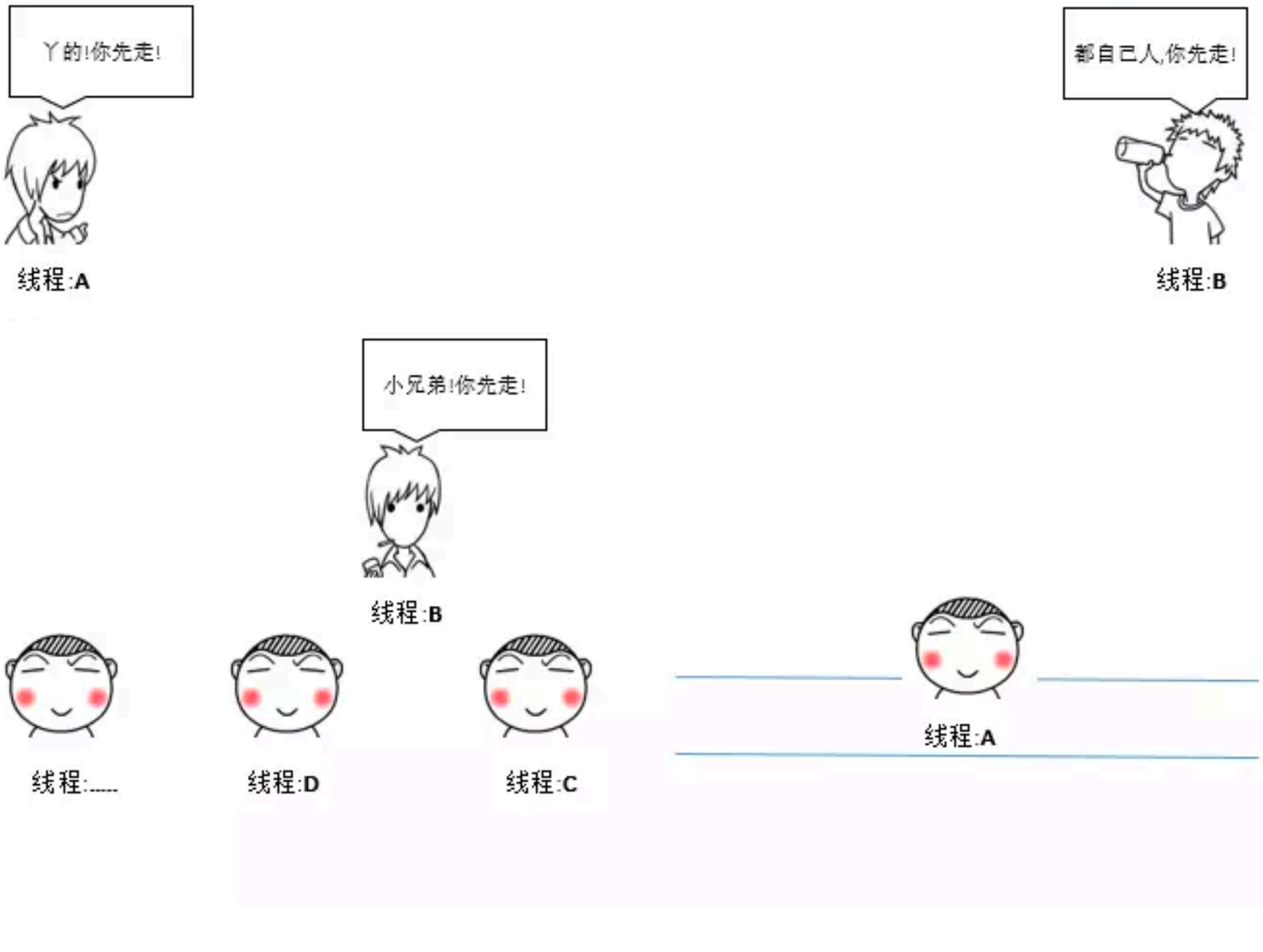
多线程其实很复杂



死锁

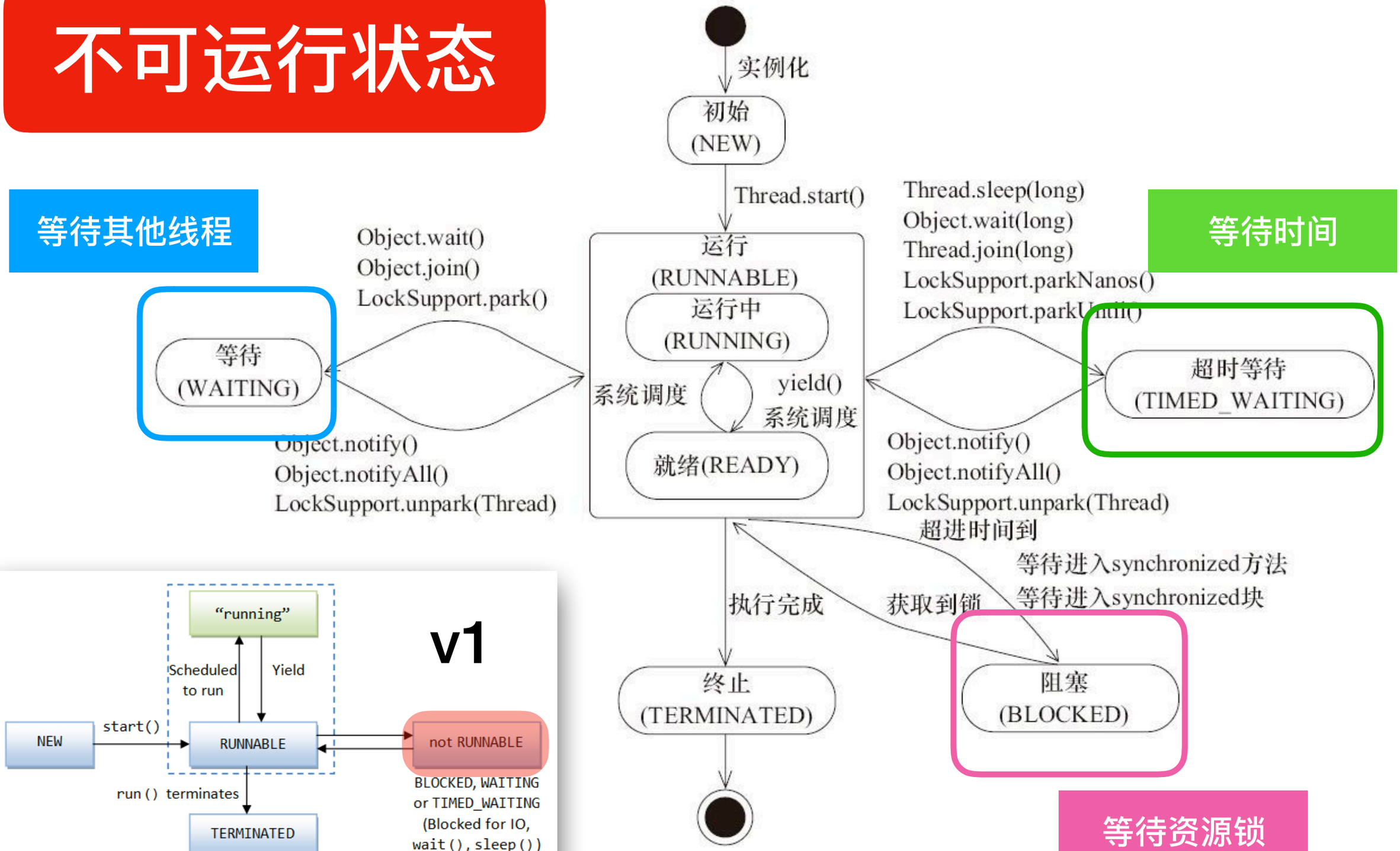


活锁与饿死



线程的状态 v2 - 基于状态

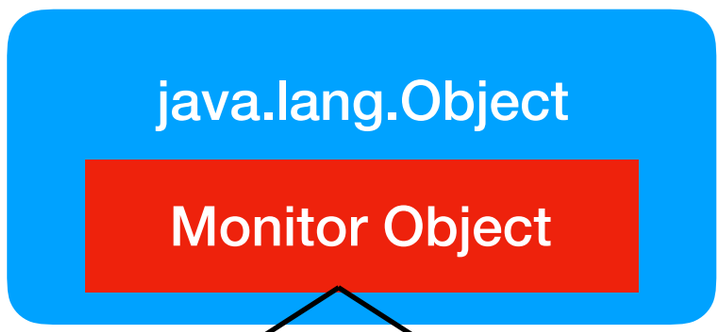
不可运行状态



EXCLUSIVE

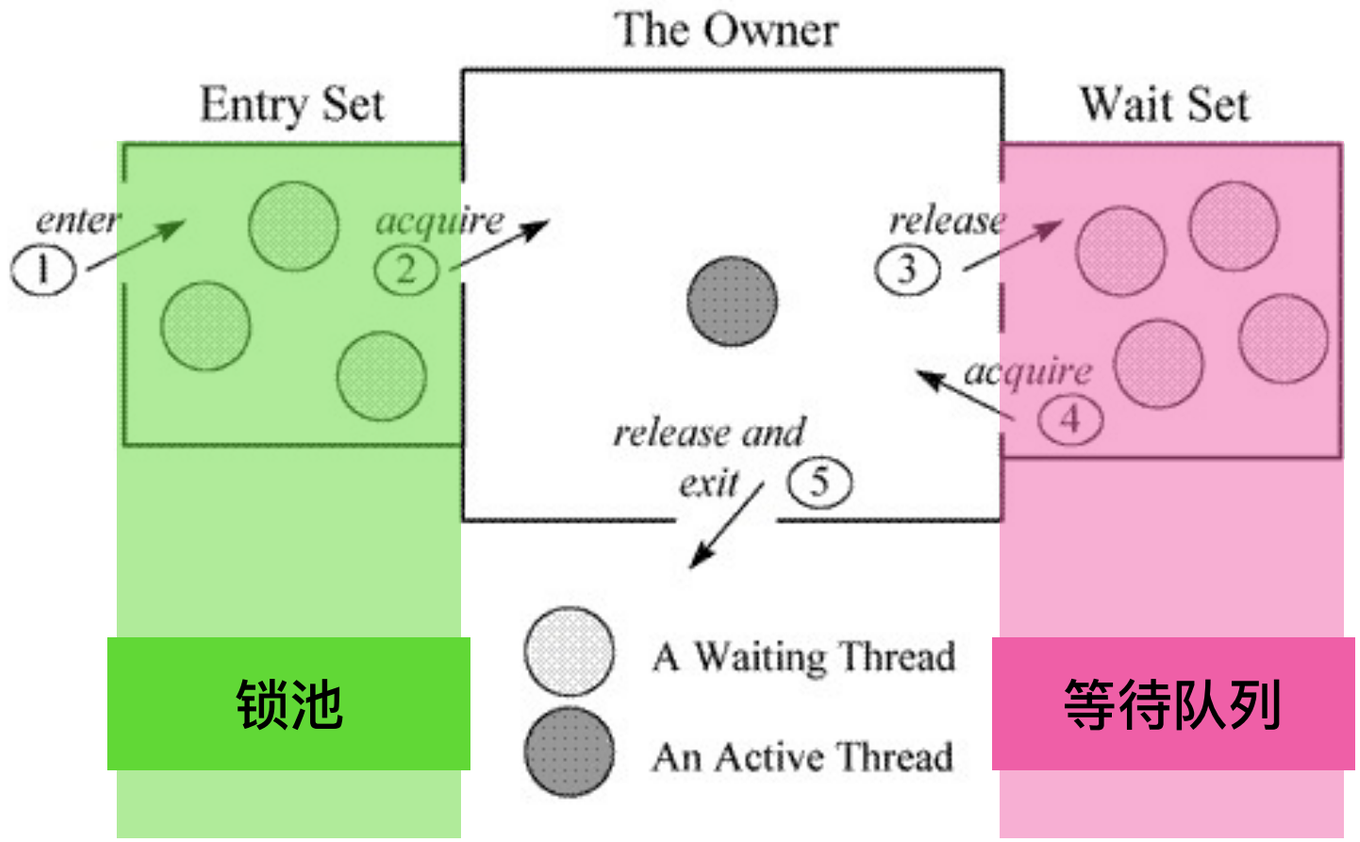
Monitor Object & Lock

线程安全：有些对象同时只能被一个线程所访问



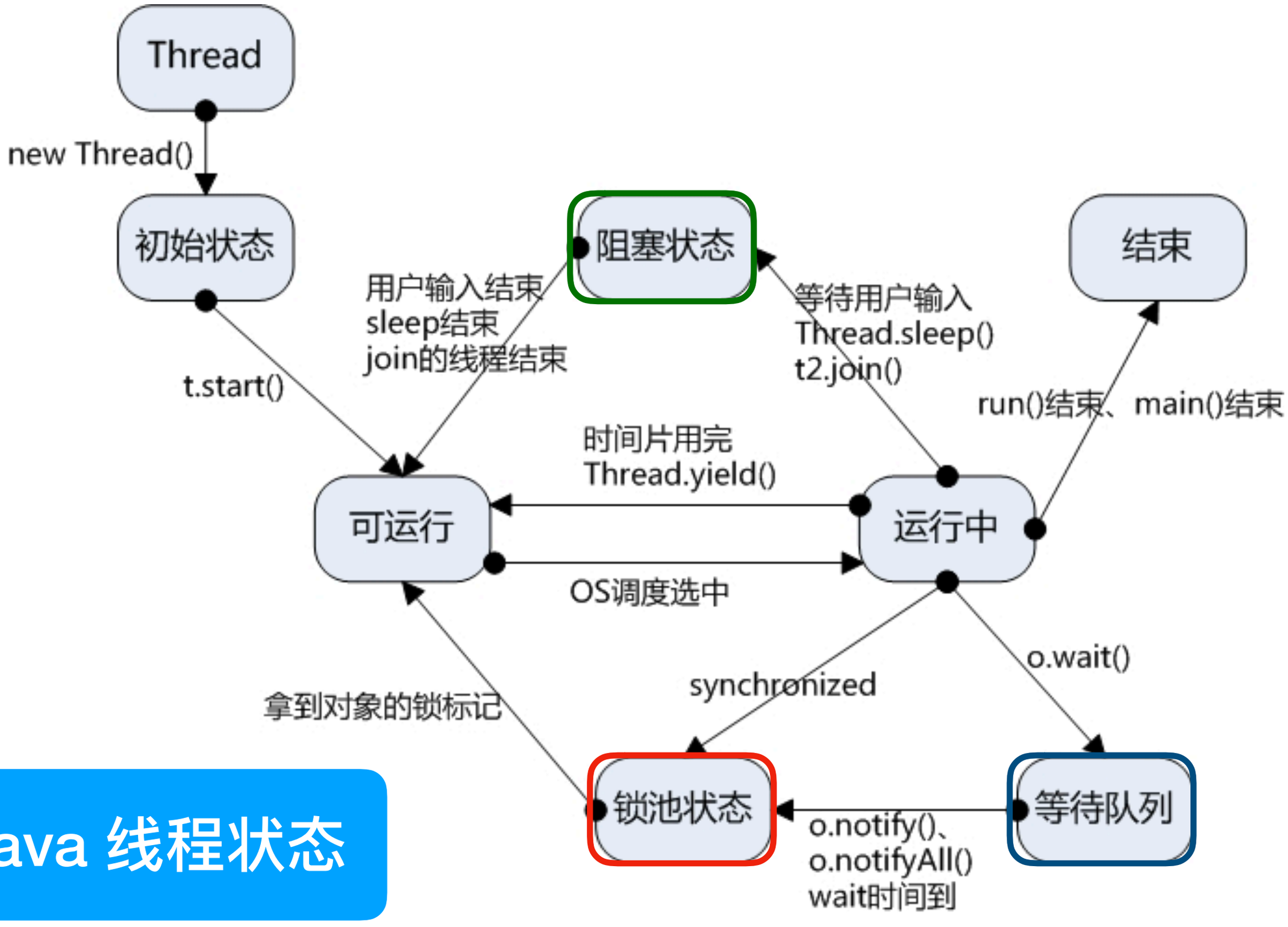
同时只有一个进程可以访问

主动放弃对资源的使用权



线程与对象之间的关系

线程的状态 v3 - 基于调度



Java 线程状态

三个重要状态（阻塞）

阻塞状态

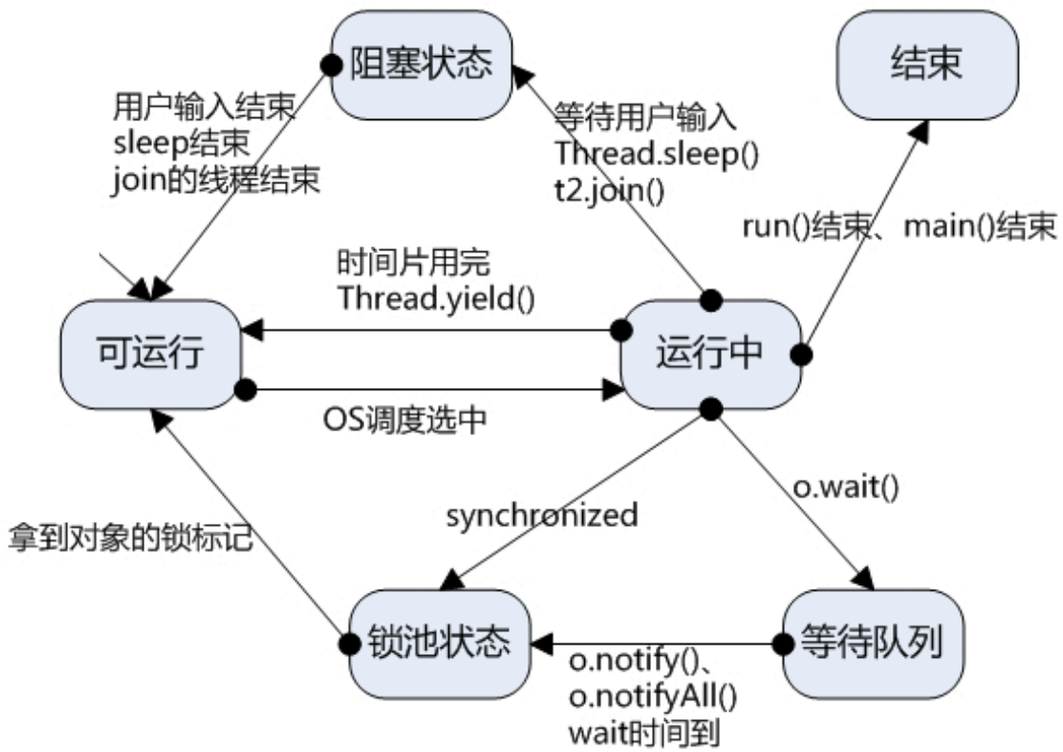
运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态

锁池状态

运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

等待队列

运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁



不同方法的主体

Thread Class

Thread Object

Java Object

Thread

new Thread()

初始状态

t.start()

可运行

用户输入结束
sleep结束
join的线程结束

阻塞状态

等待另一个线程结束

等待用户输入
Thread.sleep()
t2.join()

结束

run()结束、main()结束

时间片用完
Thread.yield()

运行中

OS调度选中

o.wait()

synchronized

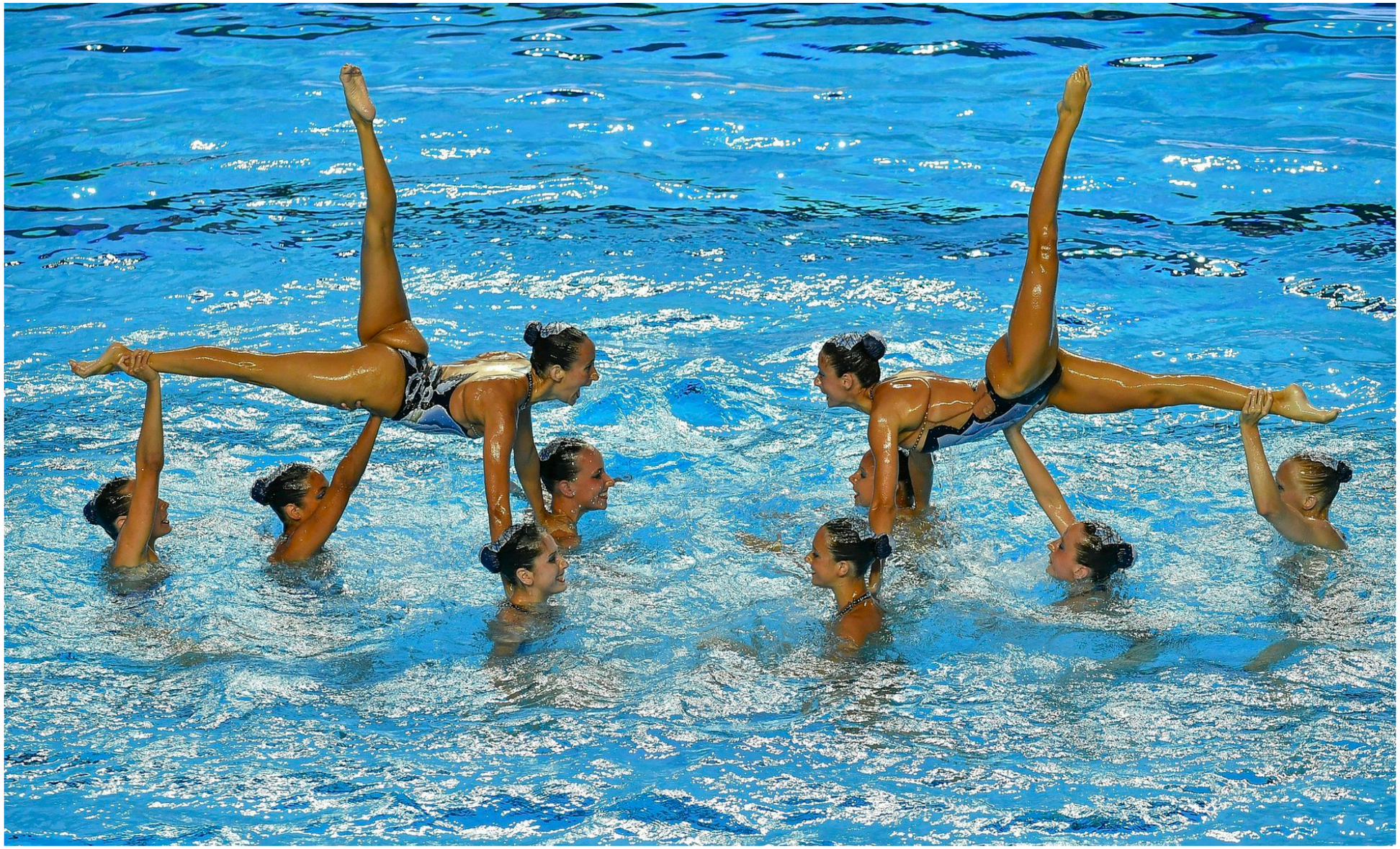
拿到对象的锁标记

锁池状态

等待队列

o.notify()、
o.notifyAll()
wait时间到

Why?



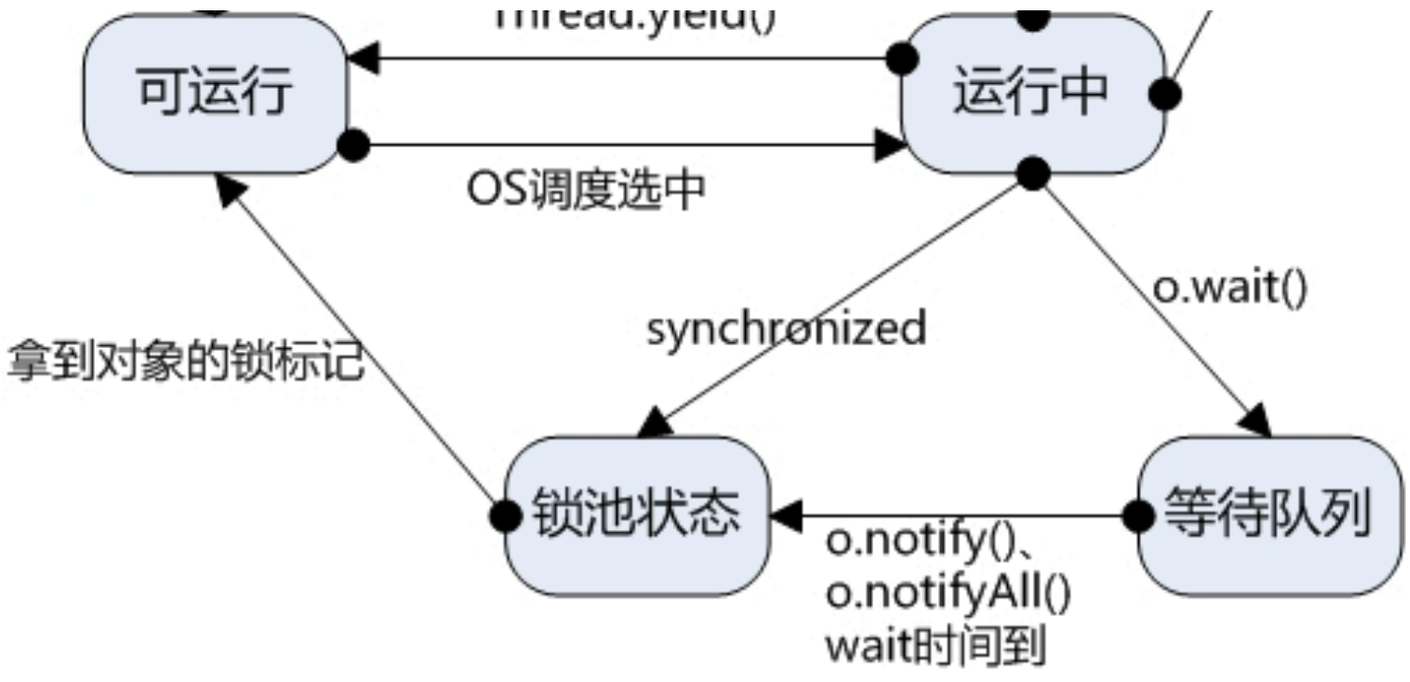
Synchronized Swimming

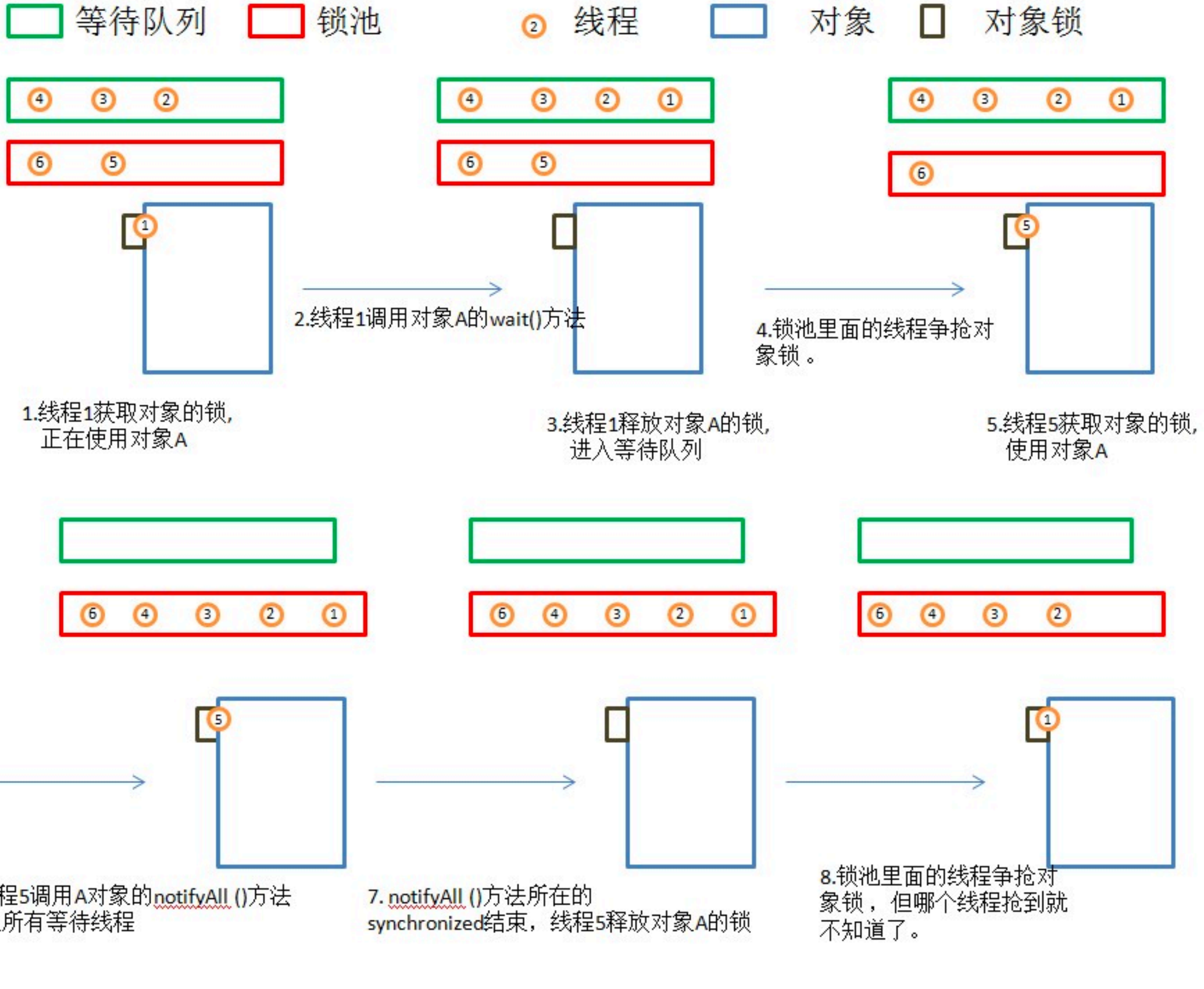
花样游泳

同步（synchronized）

```
public synchronized void methodA() { ..... }
// synchronized a method based on this object

public void methodB() {
    synchronized(this) {
    // synchronized a block of codes based on this object
        ......
    }
    synchronized(anObject) {
    // synchronized a block of codes based on another object
        ......
    }
}
```

与一个对象相关的所有synchronized代码在某一个时刻只能被一个线程执行




等待队列
锁池
线程
对象
对象锁
1.线程1获取对象的锁，
正在使用对象A
2.线程1调用对象A的wait()方法
3.线程1释放对象A的锁，
进入等待队列
4.锁池里面的线程争抢对
象锁 。
5.线程5获取对象的锁，
使用对象A
6.线程5调用A对象的notifyAll ()方法
唤醒所有等待线程
7. notifyAll ()方法所在的
synchronized结束，线程5释放对象A的锁
8.锁池里面的线程争抢对
象锁，但哪个线程抢到就
不知道了。

synchronized & wait/notify

These methods should only be called by a thread that is the owner of this object's monitor

IllegalMonitorStateException if the current thread is not the owner of the object's monitor.

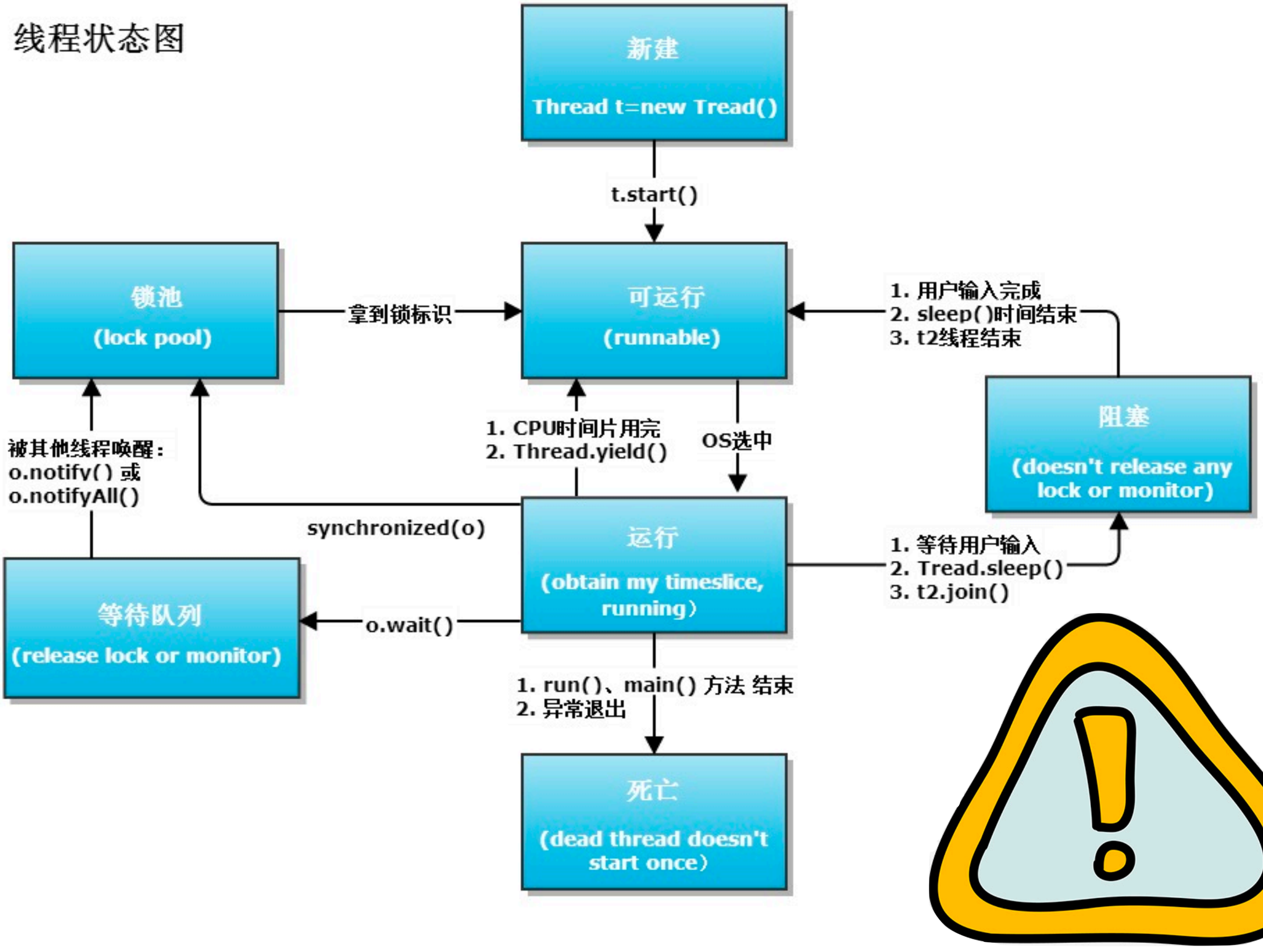
```
// 线程A 的代码
while(!condition){
    wait();
}
// do something
```

```
// 线程 B 的代码
if(!condition){
    // do something ...
    condition = true;
    notify();
}
```

```
// 线程 A 的代码
synchronized(obj_A)
{
    while(!condition){
        obj_A.wait();
    }
    // do something
}
```

```
// 线程 B 的代码
synchronized(obj_A)
{
    if(!condition){
        // do something ...
        condition = true;
        obj_A.notify();
    }
}
```

线程状态图


新建
Thread t=new Tread()
t.start()
锁池
(lock pool)
拿到锁标识
可运行
(runnable)
1. 用户输入完成
2. sleep()时间结束
3. t2线程结束
阻塞
(doesn't release any lock or monitor)
被其他线程唤醒：
o.notify() 或
o.notifyAll()
1. CPU时间片用完
2. Thread.yield()
OS选中
synchronized(o)
运行
(obtain my timeslice, running)
1. 等待用户输入
2. Tread.sleep()
3. t2.join()
等待队列
(release lock or monitor)
o.wait()
1. run()、main() 方法 结束
2. 异常退出
死亡
(dead thread doesn't start once)