一、填空题

1. 创建线程的方式有 实现Runnable接口 和 继承Thread类 。
2. 程序中可能出现一种情况：多个线种互相等待对方持有的锁，而在得到对方的锁之前都不会释放自己的锁，这就是 死锁 。
3. 若在线程的执行代码中调用yield方法后，则该线程将 从running变为ready，允许其他线程执行（自己也可能立即执行） 。
4. 线程程序可以调用 *sleep* ***方法，使线程进入睡眠状态，可以通过调用*** *setPriority* 方法设置线程的优先级。
5. 获得当前线程id的语句是 Thread.currentThread().getId() 。

二、单项选择题

1. 能够是线程进入死亡状态的是_C_。

A. 调用Thread类的yield方法

B. 调用Thread类的sleep方法

C. 线程任务的run方法结束

D. 线程死锁

2. 给定下列程序：

```
public class Holder {
    private int data = 0;
    public int getData () {return data;}
    public synchronized void inc (int amount) {
       int newValue = data + amount;
       try {Thread.sleep(5);
       } catch (InterruptedException e) {}
       data = newValue;
    }

    public void dec (int amount) {
       int newValue = data - amount;
       try {Thread.sleep(1);
       } catch (InterruptedException e) {}
       data = newValue;
    }
}

public static void main (String [] args) {
    ExecutorService es = Executors.newCachedThreadPool();
    Holder holder = new Holder ();
    int incAmount = 10, decAmount = 5, loops = 100;
    Runnable incTask = () -> holder.inc(incAmount);
    Runnable decTask = () -> holder.dec(decAmount);
```

```
    for (int i = 0; i < loops; i++) {
       es. execute(incTask);
       es. execute(decTask);
    }
    es. shutdown ();
    while (! es. isTerminated ()) {}
}
```

下列说法正确的是B。

A. 当一个线程进入holder对象的inc方法后，holder对象被锁住，因此其他线程不能进入inc方法和dec方法

B. 当一个线程进入holder对象的inc方法后，holder对象被锁住，因此其他线程不能进入inc方法，但可以进入dec方法

C. 当一个线程进入holder对象的dec方法后，holder对象被锁住，因此其他线程不能进入dec方法和inc方法

D. 当一个线程进入holder对象的dec方法后，holder对象被锁住，因此其他线程不能进入dec方法，但可以进入inc方法

3. 给定下列程序：

```
class Test2_3 {

    private static Object lockObject = new Object();
    /**
     * \* 计数器
     */
    public static class Counter {
        private int count = 0;
        public int getCount() {
            return count;
        }

        public void inc() {
            synchronized (lockObject) {
                int temp = count + 1;
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                }
                count = temp;
            }
        }

        public void dec() {
            synchronized (lockObject) {
                int temp = count - 1;
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
                }
                count = temp;
            }
```

```
        }
    }


    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        Counter counter1 = new Counter();
        Counter counter2 = new Counter();
        int loops1 = 10, loops2 = 5;
        Runnable incTask = () -> counter1.inc();
        Runnable decTask = () -> counter2.dec();
        for (int i = 0; i < loops1; i++) {
            es.execute(incTask);
        }
        for (int i = 0; i < loops2; i++) {
            es.execute(decTask);
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
    }
}
```

下面说法正确的是_C__。

A. incTask的执行线程进入counter1对象的inc方法后，counter1对象被上锁，会阻塞decTask的执行线程进入counter2对象的dec方法

B. incTask的执行线程进入counter1对象的inc方法后，counter1对象被上锁，不会阻塞decTask的执行线程进入counter2对象的dec方法

C. incTask的执行线程进入对象counter1的inc方法后，lockObject对象被上锁，会阻塞decTask执行线程进入counter2对象的方法dec

D. incTask的执行线程进入对象counter1的inc方法后，lockObject对象被上锁，不会阻塞decTask执行线程进入counter2对象的方法dec

4. 给定下列程序：

```
class Test2_4 {
    public static class Resource {
        private int value = 0;

        public int sum(int amount) {
            int newValue = value + amount;
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
            }
            return newValue;
        }

        public int sub(int amount) {
            int newValue = value - amount;
            try {
```

```
                Thread.sleep(5);
            } catch (InterruptedException e) {
            }
            return newValue;
        }
    }


    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        Resource r = new Resource();
        int loops1 = 10, loops2 = 5, amount = 5;
        Runnable sumTask = () -> r.sum(amount);
        Runnable subTask = () -> r.sub(amount);
        for (int i = 0; i < loops1; i++) {
            es.execute(sumTask);
        }
        for (int i = 0; i < loops2; i++) {
            es.execute(subTask);
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
    }
}
```

下面说法正确的是_C__。

A. 由于方法sum和sub都没有采取任何同步措施，所以sumTask和subTask的执行线程都可以同时进入共享资源对象r的sum方法或sub方法，造成对象r的实例成员value的值不一致；

B. 由于方法sum和sub都没有采取任何同步措施，所以sumTask和subTask的执行线程都可以同时进入共享资源对象r的sum方法或sub方法，造成方法内局部变量newValue和形参amount的值不一致；

C. 虽然方法sum和sub都没有采取任何同步措施，但Resource类的sum和sub里的局部变量newValue和形参amount位于每个线程各自的堆栈里互不干扰，同时多个线程进入共享资源对象r的sum方法或sub方法后，对实例数据成员value都只有读操作，因此Resource类是线程安全的

D. 以上说法都不正确

5. 给定下列程序：

```
class Test2_5 {
    public static class Resource {
        private static int value = 0;

        public static int getValue() {
            return value;
        }

        public static void inc(int amount) {
            synchronized (Resource.class) {
                int newValue = value + amount;
                try {
                    Thread.sleep(5);
                } catch (InterruptedException e) {
```

```
                }
                value = newValue;
            }
        }

        public synchronized static void dec(int amount) {
            int newValue = value - amount;
            try {
                Thread.sleep(2);
            } catch (InterruptedException e) {
            }
            value = newValue;
        }
    }

    public static void main(String[] args) {
        ExecutorService es = Executors.newCachedThreadPool();
        int incAmount = 10, decAmount = 5, loops = 100;
        Resource r1 = new Resource();
        Resource r2 = new Resource();
        Runnable incTask = () -> r1.inc(incAmount);
        Runnable decTask = () -> r2.dec(decAmount);
        for (int i = 0; i < loops; i++) {
            es.execute(incTask);
            es.execute(decTask);
        }
        es.shutdown();
        while (!es.isTerminated()) {
        }
    }
}
```

下面说法**错误的**的是_B__。

A. 同步的静态方法public synchronized static void dec (int amount) {} 等价于public static void dec (int amount) {synchronized (Resource. class) {}}

B. incTask的执行线程访问的对象r1，decTask访问的是对象r2，由于访问的是不同对象，因此incTask的执行线程和decTask的执行线程之间不会同步

C. 虽然incTask的执行线程和decTask的执行线程访问的是Resource类不同对象r1和r2，但由于调用的是Resource类的同步静态方法，因此incTask的执行线程和decTask的执行线程之间是被同步的

D. 一个线程进入Resource类的同步静态方法后，这个类的所有静态同步方法都被上锁，而且上的是对象锁，被锁的对象是Resource.class。但是这个锁的作用范围是Resource类的所有实例，即不管线程通过Resource类的哪个实例调用静态同步方法，都将被阻塞

6. 假设一个临界区通过Lock锁进行同步控制，当一个线程拿到一个临界区的Lock锁，进入该临界区后，该临界区被上锁。这时下面的说法正确的是D___。

A. 如果在临界区里线程执行Thread.sleep方法，将导致线程进入阻塞状态，同时临界区的锁会被释放；如果在临界区里线程执行Lock锁的条件对象的await方法，将导致线程进入阻塞状态，同时临界区的锁会被释放

B.如果在临界区里线程执行Thread.sleep方法，将导致线程进入阻塞状态，同时临界区的锁不会被释放；如果在临界区里线程执行Lock锁的条件对象的await方法，将导致线程进入阻塞状态，同时临界区的锁不会被释放

C. 如果在临界区里线程执行Thread.sleep方法，将导致线程进入阻塞状态，同时临界区的锁会被释放；如果在临界区里线程执行Lock锁的条件对象的await方法，将导致线程进入阻塞状态，同时临界区的锁不会被释放

D. 如果在临界区里线程执行Thread.sleep方法，将导致线程进入阻塞状态，同时临界区的锁不会被释放；如果在临界区里线程执行Lock锁的条件对象的await方法，将导致线程进入阻塞状态，同时临界区的锁会被释放

三、问答题

1：有三个线程T1，T2，T3，怎么确保它们按指定顺序执行：首先执行T1，T1结束后执行T2，T2结束后执行T3，T3结束后主线程才结束。请给出示意代码。

```
    public static void main(String[] args) {
        Thread t1 = new T1();
        Thread t2 = new T2();
        Thread t3 = new T3();

        try {
            t1.start();
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        try {
            t2.start();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        try {
            t3.start();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```