一、填空题

1. 创建线程的方式有___通过实现 Runnable 接口创建线程_____和___通过继承 Thread 类来创建线程_____。

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___主动让出 CPU 使用权，转到就绪态_____。

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(p:int)___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：

```
public 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：

```
public 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：

```
public 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 class TestThread {
	public static void main(String[] args) throws InterruptedException{
		Thread T1 = new Thread(new PrintTips("T1"));
		Thread T2 = new Thread(new PrintTips("T2"));
		Thread T3 = new Thread(new PrintTips("T3"));
		T1.start();
		T1.join();
		T2.start();
		T2.join();
		T3.start();
		T3.join();
		System.out.println(Thread.currentThread().getId()+" current thread,"+"Main thread has executed");
	}
}

class PrintTips implements Runnable{
	private String str;
	public PrintTips(String current){
		str = current;
	}

	@Override
```

```
    public void run() {
        System.out.println(Thread.currentThread().getId()+"  current  thread,"+str+"  has executed");
    }
}
```

编程题 2:

```
public class SyncQueue1<T> {
    private ArrayList<T> list = new ArrayList<>();
    private static Lock lock = new ReentrantLock();
    private static Condition producerWait = lock.newCondition();
    private static Condition consumerWait = lock.newCondition();
    public void produce(List<T> elements) {
        lock.lock();
        try{
            while (list.size()>0){
                producerWait.await();
            }
            System.out.print("Produce elements:");
            for (T cur:elements){
                list.add(cur);
                System.out.print(cur+"  ");
            }
            System.out.println("");
            consumerWait.signalAll();
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
    }
    public List<T> consume(){
        lock.lock();
        try {
            while (list.size()==0){
                consumerWait.await();
            }
            System.out.print("Consume elements:");
            for (T cur:list){
                System.out.print(cur+"  ");
            }
            List<T> temp = list;
            list.clear();
            System.out.println("");
            producerWait.signalAll();
            return temp;
```

```
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
        return null;
    }
}
public class SyncQueue2<T> {
    private ArrayList<T> list = new ArrayList<>();
    private static Lock lock = new ReentrantLock();
    private static Condition consumerWait = lock.newCondition();
    public void produce(List<T> elements) {
        lock.lock();
        System.out.print("Produce elements:");
        for (T cur:elements){
            list.add(cur);
            System.out.print(cur+"   ");
        }
        System.out.println("");
        consumerWait.signalAll();
        lock.unlock();
    }
    public List<T> consume(){
        lock.lock();
        try {
            while (list.size()==0){
                consumerWait.await();
            }
            System.out.print("Consume elements:");
            for (T cur:list){
                System.out.print(cur+"   ");
            }
            List<T> temp = list;
            list.clear();
            System.out.println("");
            return temp;
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
        return null;
    }}
```

编程题3:

```
public class ReusableThread extends Thread{
    private Runnable runTask = null;  //保存接受的线程任务
    Lock lock = new ReentrantLock();
    Condition runloopCondition = lock.newCondition();
    Condition taskCondition = lock.newCondition();
     //只定义不带参数的构造函数
    public ReusableThread(){
        super();
    }
    @Override
    public void run() {
        //这里必须是永远不结束的循环
        while (true){
            lock.lock();
            try {
                while (runTask==null){
                    runloopCondition.await();
                }
                runTask.run();
                runTask = null;
                taskCondition.signalAll();
            }catch (InterruptedException e){
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
        }
    }
    public void submit(Runnable task){
        lock.lock();
        try {
            while (runTask != null){
                taskCondition.await();
            }
            runTask = task;
            runloopCondition.signalAll();
        }catch (InterruptedException e){
            e.printStackTrace();
        }finally {
            lock.unlock();
        }
    }
}
```