Multithread Programming

1、 Distinct thread from process

The process is the minimum unit of resource.

The thread is the minimum unit of schedule.

2、 How to start a thread(3 method)

- Extends Thread

```
/**
 * account
 * @author ZhaoYao
 *
 */
class ThreadSample1 extends Thread{
    public ThreadSample1(String name){
       super(name);
    }
       public void run(){
              System.out.println(getName()+"running.....  ");
       }
}


 public class MTTest1 {
       public static void main(String[] args){
         System.out.println(Thread.currentThread().getName());
         for (int i = 0; i < 10; i++){
                    ThreadSample1 thread = new ThreadSample1("thread" + i);
```

```
                thread.start();
            }

        }
    }
```

- Implements Runnable

```
/**
 * account
 * @author ZhaoYao
 *
 */
class ThreadSample2 implements Runnable {
    String name;

    public ThreadSample2(String name) {
        this.name = name;
    }

    public void run() {
        System.out.println(name + "running.....  ");
    }
}

public class MTTest2 {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(new ThreadSample2("thread" + i));
            thread.start();
        }
```

```
        }
}
```

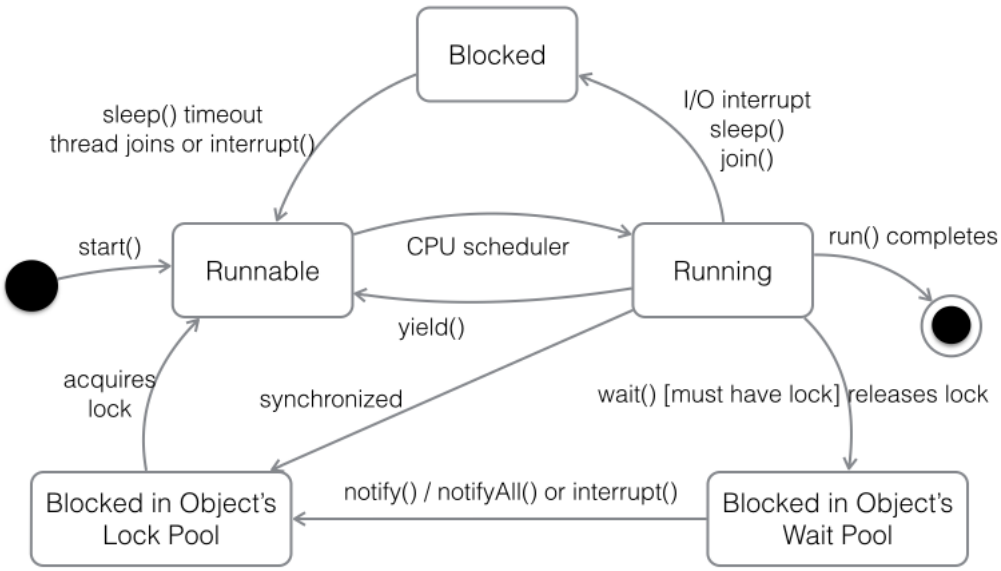
- Implements Callable

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/**
 * account
 * @author ZhaoYao
 *
 */
class ThreadSample3 implements Callable<String> {
        String name;

        public ThreadSample3(String name) {
                this.name = name;
        }

        public String call() {
                // System.out.println(name + "running..... ");
                return (name + "running.....    ");
        }
}

public class MTTest3 {
        public static void main(String[] args) {
                ExecutorService service = Executors.newFixedThreadPool(10);
                for (int i = 0; i < 10; i++) {
                        try {
                                System.out.println(service.submit(new ThreadSample3("thread" + i)).get());
                        } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }

        }
}
```

3、 The status of thread

A classic diagram:



4、 About synchronized,lock

a simple sample shows that why we need Synchronization.

```
/**
 * account
 * @author LuoHao
 *
 */
```

```
public class Account {
    private double balance;

    /**
     * deposit money
     * @param money
     */
    public void deposit(double money) {
        double newBalance = balance + money;
        try {
            Thread.sleep(10);    //pretend the operation will spent 10ms
        }
        catch(InterruptedException ex) {
            ex.printStackTrace();
        }
        balance = newBalance;
    }

    /**
     * get Balance
     */
    public double getBalance() {
        return balance;
    }
}
/**
 * AddMoneyThread
 * @author LuoHao
 *
 */
public class AddMoneyThread implements Runnable {
    private Account account;
    private double money;
```

```
    public AddMoneyThread(Account account, double money) {
        this.account = account;
        this.money = money;
    }

    @Override
    public void run() {
        account.deposit(money);
    }

}

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test01 {

    public static void main(String[] args) {
        Account account = new Account();
        ExecutorService service = Executors.newFixedThreadPool(100);

        for(int i = 1; i <= 100; i++) {
            service.execute(new AddMoneyThread(account, 1));
        }

        service.shutdown();

        while(!service.isTerminated()) {}

        System.out.println("The account have money:" + account.getBalance());
    }
}
```

If have no synchronization, the result maybe <10. In fact, it is 1.0 on my computer.

We have 3 method to get a correct result.

- Method 1:

Change the method of account from

éì ÄäáÅ=îçáÇ=ÇÉéçëáí EÇçì ÄäÉ=ãçåÉóF

to:

```
public synchronized void deposit(double money)
```

- Method 2:

Add synchronized for code block in AddMoneyThread like this:

```
  public void run() {
        synchronized (account) {
            account.deposit(money);
        }
    }
```

- Method 3:

Create a Lock object for Account

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 *
 *
 * @author LuoHao
 *
 */
public class Account {
    private Lock accountLock = new ReentrantLock();
    private double balance;
```

```
    /**
     *
     *
     *  @param money
     *
     */
    public void deposit(double money) {
        accountLock.lock();
        try {
            double newBalance = balance + money;
            try {
                Thread.sleep(10);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            balance = newBalance;
        }
        finally {
            accountLock.unlock();
        }
    }

    /**
     * get balance
     */
    public double getBalance() {
        return balance;
    }
}
```

- Method4:

Semaphore can be used to control that the number of one resource can be access meanwhile. The number is set by constructor like this :Semaphore semaphore = new Semaphore(5).

A method named acquire() is offered to attain a promise to access the resource

The method named release() is offered to free a promise to the resource

```
package CounterSemaphore;

import java.util.concurrent.Semaphore;

/**
 * account
 * @author ZhaoYao
 *
 */
public class Account {
//     private Lock accountLock = new ReentrantLock();
    private double balance;       // 账户余额
        private Semaphore semaphore = new Semaphore(1);

    /**
     *  存款
     * @param money  存入金额
     */
    public /*synchronized*/ void deposit(double money) {
//          accountLock.lock();


          try {
               semaphore.acquire();
                    double newBalance = balance + money;
                    try {
```

```
                    Thread.sleep(10);    // 模拟此业务需要一段处理时间
                }
                catch(InterruptedException ex) {
                    ex.printStackTrace();
                }

                balance = newBalance;
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        finally{
//            accountLock.unlock();
            semaphore.release();
        }
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

Defference between synchronized and lock:

- performance: lock is better than synchronized
- lock must to be freed in finally block. If you forget it and when the protected code block throw an exception, the lock will never not to be freed! For synchronized, JVM will assure it will free correctly.
- Lock offer versatile way to use, such as time limit, be interrupted and

so on.

5、 Executor and ThreadPoolExecutor

How to manage the threads more efficient and more convenient, java.util.concurrent offers some useful tools.

In the example of the 4 section, you have used ThreadPoolExecutor.

ThreadPoolExecutor can control the number of the thread, so that avoid creating too much threads and then exhaust the resource of the system.

It can reuse the existed thread, not need to create a new one, so it can save some spending, and improve response speed.

Java Library offers 4 method to create a thread pool:

1.newFixedThreadPool

2.newCachedThreadPool

3.newSingleThreadExecutor

4.newScheduledThreadPool

The most common is fixed thread pool.

This is a server that every time it accept a request, it will create a thread.

```
package Multisocket;


import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

/**

 * account

 *  @author ZhaoYao

 *

 */

class PerTaskSever {

	public static void main(String[] args) throws IOException {


		ServerSocket socket = new ServerSocket(2011);

		while (true) {

			final Socket connection = socket.accept();
```

```
                Runnable task = new Runnable() {
                    public void run() {
                        System.out.println("Handle connection....");
                        System.out.println(Thread.activeCount());
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                    }
                };
                new Thread(task).start();
            }
        }
}
```

It is a sample using fixed number threadpool. When a request arrived, it will use a thread in threadpool to handle the connection. The number we can monitor will not more than 3(including main thread).

```
package Multisocket;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
/**
 * account
 * @author ZhaoYao
 *
 */
public class TaskServerThreadPool {
    private static final int MAX_NUM_THREADS = 2;
```

```
    private static final Executor exec = Executors.newFixedThreadPool(MAX_NUM_THREADS);


    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        ServerSocket socket = new ServerSocket(2011);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    System.out.println("Handle connection....");
                    System.out.println(Thread.activeCount());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                }
            };
            exec.execute(task);
        }
    }
}
```