Chapter 8 Java Multi-thread

Xiang Zhang

javaseu@163.com

https://wdsseu.github.io/java/

Content

- Notion of Thread
- Creation of Thread
- Scheduling of Thread
- Priority of Thread
- Synchronization of Thread

Notion of Thread – Example

- How to send message to Mary , and meanwhile receive her message?

Notion of Thread – Example

- Way 1: Polling (轮询), 20s as interval, ask for response from Mary's port
  - Bad! How can Mary's port know what messages you have received and what have not?
  - Bad! Mary will be angry if I have no response in 20s 
- Way 2: When sending a message to Mary, ask her port
  - Bad! What if you have no interest of sending her message?
- Way 3: When Mary send me a message, show it immediately!
  - Good, but some code needed to query Mary's port repeatedly, but if this code is running, your code of sending message cannot be run…

Notion of Thread – Example

- Implementing way 3:
  - Create one thread for sending, and one for receiving
  - Assign each thread a stand-alone task

```
Runnable receiveMessageJob = new ReceiveMessageJob();
Runnable sendMessageJob = new SendMessageJob();
Thread receiveThread = new Thread(receiveMessageJob);
Thread sendThread = new Thread(sendMessageJob);
receiveThread.start();
sendThread.start();
```

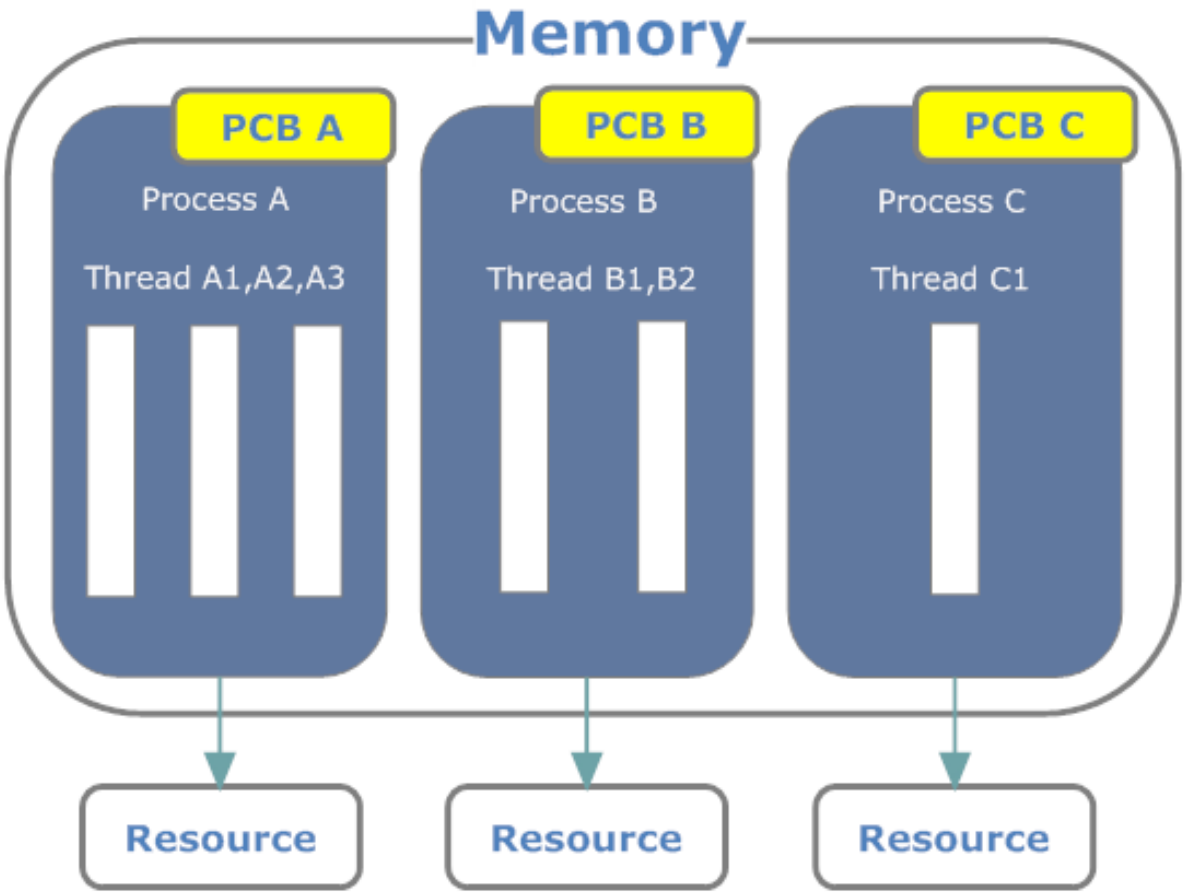
The Notion of Thread

- Thread is everywhere!
  - When we download multiple files…
  - When Outlook or Foxmail is sending and receiving mails ..
  - When we save text and meanwhile edit it…
  - When we can chat with many people at the same time…
  - When the ATM machine can give you money and meanwhile calculate the interest…

The Notion of Thread

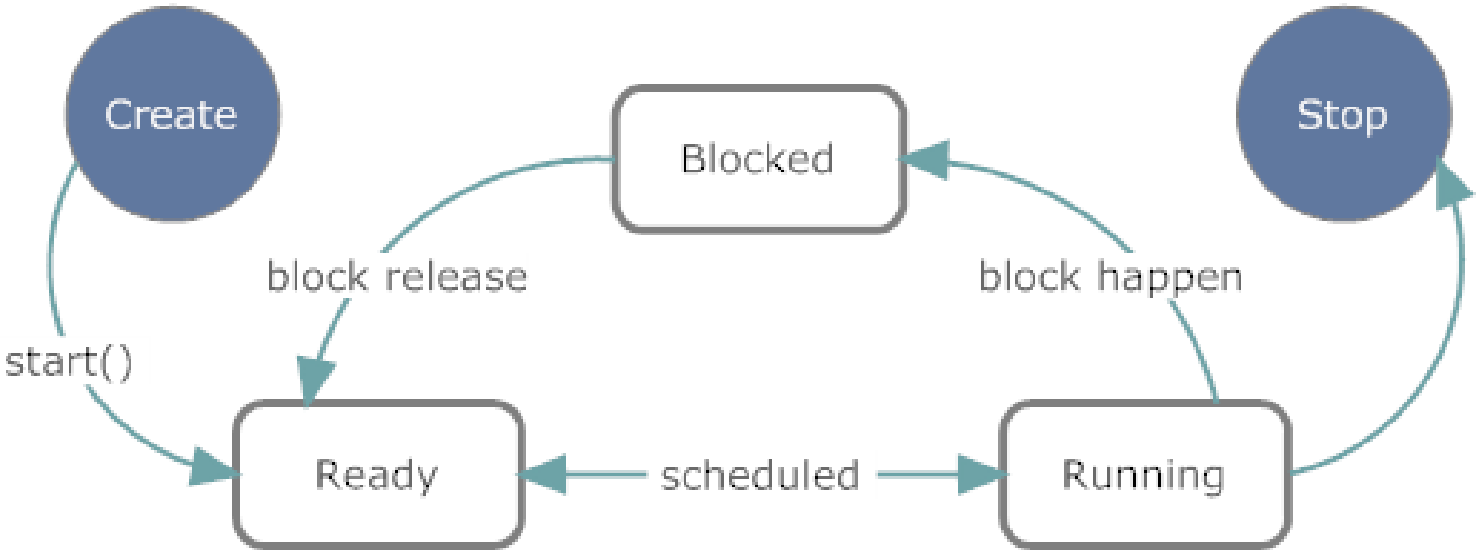
- Piece of code being able to run concurrently with others
- A part of Process (进程)
- A non-multithread process is a single-thread process
- Main method is a thread (called main thread)
- Process possesses system resource, while thread not
- All threads in a process shares the resource of the process

The Notion of Thread



The Notion of Thread

- Change of thread state



Creation of Thread

- Two ways to create a thread
  - Implements Runnable interface
    - Create a runnable task by implementing Runnable interface
    - Assign this task to an object of Thread class
    - Run the Thread object
  - Inherited from Thread class
    - Create a subclass of Thread class
    - Override the run() method
    - Run the object of this subclass
- Which one you prefer?

Implement Runnable Interface

```
public class MyTask implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread()
                    .getName() + " is doing MyTask");
            try{
                Thread.sleep(1000);
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        Thread thread = new Thread(new MyTask(), "My Thread");
        thread.start();
    }
}
```

Inherit Thread Class

```
public class MyThread extends Thread {
    public void run() {
        while (true) {
            System.out.println(this.getName()
                    + " is doing its own task");
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.setName("Thread1");
        myThread.start();
    }
}
```

Run Multiple Threads – Guess the Result

```
public class TwoThreadsDemo implements Runnable{

    public void run() {
        for(int i=0; i<10; i++){
            String threadName = Thread.currentThread().getName();
            System.out.println(threadName + " is running");
        }
    }

    public static void main(String[] args){
        System.out.println(Thread.currentThread().getName() + " is running");
        Runnable twoThreadDemo = new TwoThreadsDemo();
        Thread thread1 = new Thread(twoThreadDemo);
        thread1.setName("Thread1");
        Thread thread2 = new Thread(twoThreadDemo);
        thread2.setName("Thread2");
        thread1.start();thread2.start();
        System.out.println(Thread.currentThread().getName() + " is running");
    }
}
```

Scheduling of Thread

- Different threads need different running order
  - Downloading of important resource needs a high priority.
  - Downloading of minor resource needs a lower priority.
- Thread scheduler in JVM schedules threads according to their priority.
- Level 1 - 10
  - MAX_PRIORITY = 10
  - MIN_PRIORITY = 1
  - NORM_PRIORITY = 5

Thread Scheduling – Guess the Result?

```
public class TSchedulingDemo implements Runnable{
    public void run(){
        while(true){
            String name = Thread.currentThread().getName();
            System.out.println(name + "is running.");
            try{
                Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
    public static void main(String[] args){
        Runnable myTask = new TSchedulingDemo();
        Thread t1 = new Thread(myTask);
        t1.setName("t1");t1.setPriority(10);
        Thread t2 = new Thread(myTask);
        t2.setName("t2");t2.setPriority(1);
        t1.start();t2.start();
    }
}
```

Thread Scheduling – Guess the Result?

```
public class TSchedulingDemo2 implements Runnable{
    public void run(){
        for(int i=0;i<30; i++){
            String name = Thread.currentThread().getName();
            System.out.println(name + "is running.");
        }
    }
    public static void main(String[] args){
        Runnable myTask = new TSchedulingDemo2();
        Thread t1 = new Thread(myTask);
        t1.setName("t1");t1.setPriority(10);
        Thread t2 = new Thread(myTask);
        t2.setName("t2");t2.setPriority(1);
        t1.start();t2.start();
    }
}
```

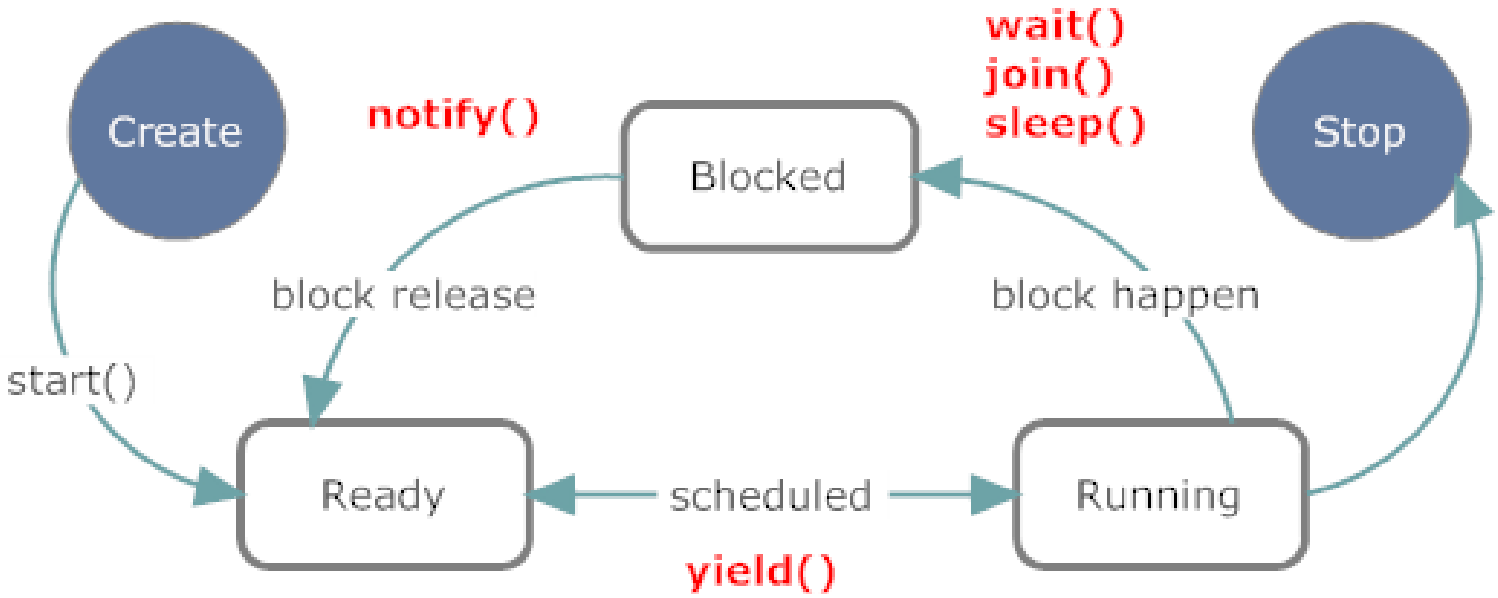
Notice

- The scheduling by priority only affects the running order, not the running frequence.
- The control of Scheduler has a basic pattern.
- But the pattern is not absolute!
- And the logic of your program should not rely on the pattern!

Control of Thread

- The state of current thread can be controlled
  - join() method: let one thread run after another’s finish
  - *sleep*() / wait() / join() let one thread change its state from running to blocked
  - interrupt() let a blocked thread to end
  - *yield*() let current thread yield its running to other threads

线程控制


Create
notify()
Blocked
wait()
join()
sleep()
Stop
block release
block happen
start()
Ready
scheduled
Running
yield()

Control of Thread – Join

```
public class JoinDemo extends Thread{

    public void run(){
        String name = Thread.currentThread().getName();
        System.out.println("Thread started.");
        for(int i=0; i<10; i++){
            System.out.println("Thread is running.");
        }
        System.out.println("Thread ended.");
    }

    public static void main(String[] args){
        System.out.println("Main started");
        Thread t = new JoinDemo();
        t.start();
        try{ t.join();}  catch(Exception e){}
        System.out.println("Main ended");
    }
}
```

Control of Thread – Yield

- Different with Join
  - yield意味着放手，放弃，投降。一个调用yield()方法的线程告诉虚拟机它乐意让其他线程占用自己的位置。这表明该线程没有在做一些紧急的事情。注意，这仅是一个暗示，并不能保证不会产生任何影响。
  - Yield告诉当前正在执行的线程把运行机会交给线程池中拥有相同优先级的线程。
  - Yield不能保证使得当前正在运行的线程迅速转换到可运行的状态
  - 它仅能使一个线程从运行状态转到可运行状态，而不是等待或阻塞状态

Control of Thread – Interrupt

```
public class InterruptDemo extends Thread{

    public void run(){
        System.out.println("Thread started");
        while(true){
            try{
                System.out.println("Thread is running");
                Thread.sleep(1000);
            }catch(InterruptedException e){
                System.out.println("Interupped, and ended");
                return;
            }
        }
    }
```

Control of Thread- Interrupt

```
    public static void main(String[] args){
        System.out.println("Main started");
        try{
            Thread t = new InterruptDemo();
            t.start();
            Thread.sleep(2000);
            t.interrupt();
        }catch(Exception e){}
        System.out.println("Main ended");
    }
}
```

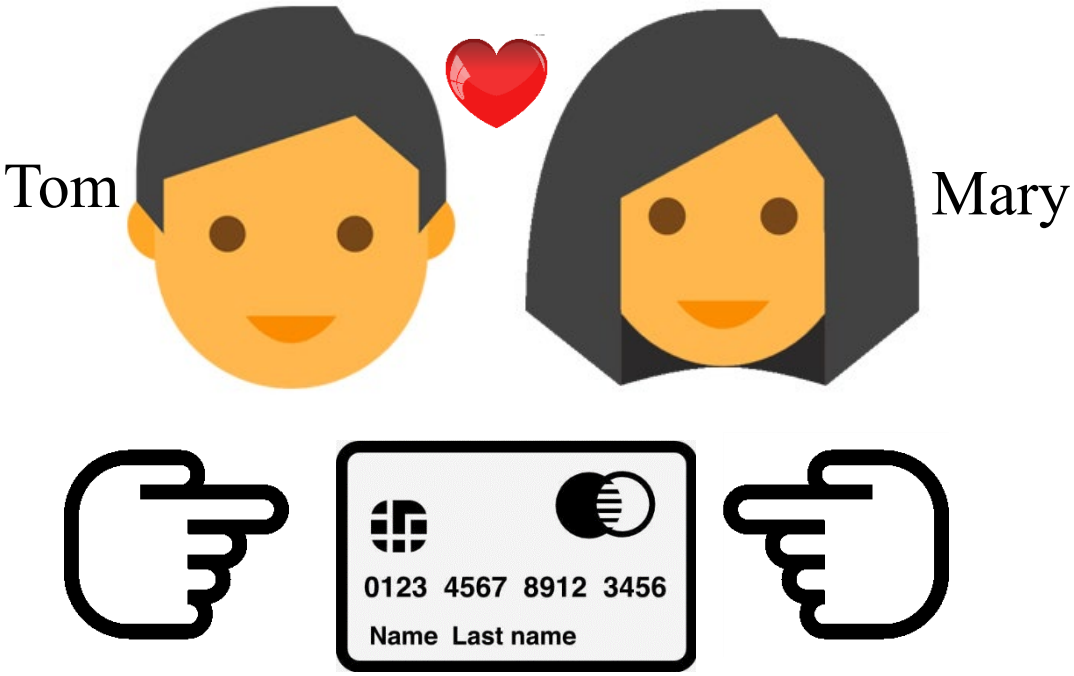
Control of Thread – Best Practice of Ending a Thread

```
public class EndThreadDemo extends Thread{
    public void run(){
        while(!interrupted()){
            System.out.println("Thread is running");
        }
        System.out.println("Thread interruped and ended");
    }

    public static void main(String[] args){
        Thread t = new EndThreadDemo();
        t.start();
        try{
            Thread.sleep(10);
        }catch(Exception e){}
        t.interrupt();
    }
}
```

Two People One Account

The Problem of **Two People One Account**



Two People One Account

Tom and Mary share an bank account

They both do not want a deficit

They both promise to query the balance before withdrawal

No enough money, no withdrawal!

But someday, they got a deficit!

How can it HAPPEN ??!!

The Reason …

Synchronization

- It happened this way
  - Tom wanted to withdraw $100 one day .
  - He found the balance was $150 , he was so happy!
  - Suddenly…… He fell asleep…….zzzzz
  - Mary did not know Tom checked the balance, she withdrew $100 when Tom was sleeping!
  - Tom woke up, withdrew $100, now they had -$50 in their account!!

Synchronization of Thread - describe the problem in Java

```
Runnable withdrawTask = new WithdrawTask();
Thread tom = new Thread (withdrawTask, "Tom");
Thread mary = new Thread(withdrawTask, "Mary");
tom.start();
Thread.sleep(9999);//在某处代码中Tom睡了一会儿
mary.start();

//WithdrawTask中的部分代码：
public void withdraw(double amount){
    if(account.getBalance()>=amount){
        account.withdraw(amount);
    }else{
        //放弃取款
    }
}
```

Synchronization

- Problem is: Mary can withdraw before Tom is done
  - The withdraw action should be ATOMIC
  - Mary should not be allowed to withdraw before Tom finishes withdraw(double amount)
  - We need a LOCK to control the invocation of withdraw
    - When Tom withdraw, he locks the Object of withdraw method, and take the key
    - When Mary want to withdraw, she finds the key is taken away
    - Tom finish his sleep and withdraw, and then return the key
    - Mary get the key, withdraw, and return the key

Thread Synchronization

> *“ Thread synchronization ensures that objects are modified by **only one** thread **at a time** and that threads are **prevented** from accessing **partially updated** objects during modification by another thread. ”*

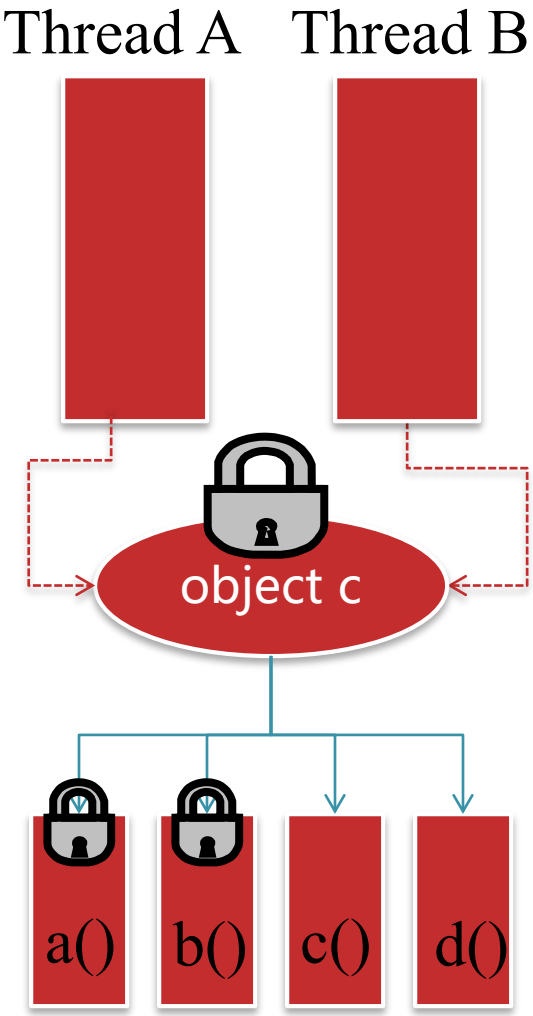
Synchronization of Thread - describe the problem in Java

```
Runnable withdrawTask = new WithdrawTask();
Thread tom = new Thread (withdrawTask, "Tom");
Thread mary = new Thread(withdrawTask, "Mary");
tom.start();
Thread.sleep(9999);//在某处代码中Tom睡了一会儿
mary.start();

//WithdrawTask中的部分代码：
public synchronized void withdraw(double amount){
    if(account.getBalance()>=amount){
        account.withdraw(amount);
    }else{
        //放弃取款
    }
}
```

Synchronization

- Each object has a lock and corresponding key
  - When the object has no synchronized methods, the lock does not work.
  - When the object has a synchronized methods, the lock begins to work.

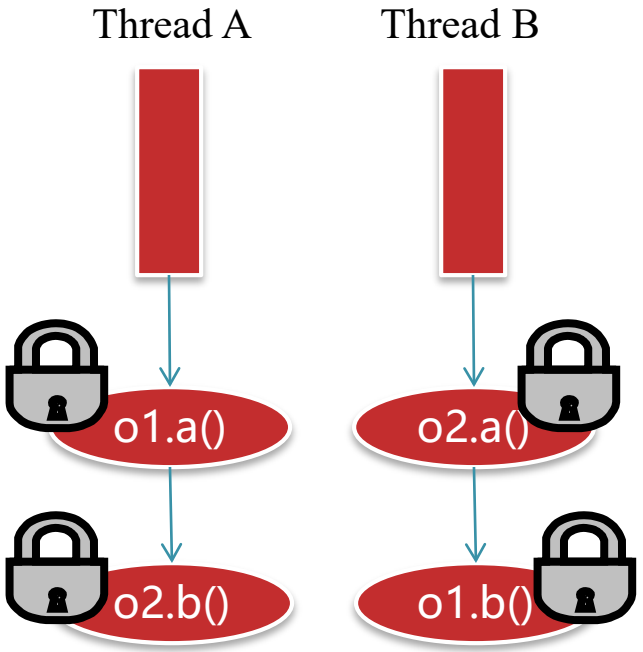


Synchronization

- Synchronization by locking the objects
- If an object has multiple synchronized method:
  - When locked (visited by current thread), other threads cannot visit any synchronized method
- Synchronized methods bring Thread-safe
- Think: Why we do not make all methods synchronized?
  - Performance…
  - Logic…

Deadlock

- Synchronized methods may bring deadlock
- Suppose:
  - Thread A need to visit
    - o1.a()
    - o2.b()
  - Thread B need to visit
    - o2.a()
    - o1.b()



Synchronization


```
public class ATM {

    String accountName = "Tom and Mary";
    double balance = 0;

    public synchronized void withdraw(double amount) {
        while (balance < amount) {
            System.out.println("No enough balance, "
                    + Thread.currentThread().getName() + " has to wait.");
            try {
                wait(1000);
            } catch (Exception e) {
            }
        }
        System.out.println("Enough balance now, "
                + Thread.currentThread().getName() + " can withdraw now.");
        balance -= amount;
    }

    public void deposit(double amount) {
        System.out.println(Thread.currentThread().getName() + " is depositing.");
        balance += amount;
    }
}
```

```java
public class TomTask implements Runnable{

    ATM o;

    public TomTask(ATM account){
        o = account;
    }

    public void run(){
        o.withdraw(100);
    }
}
```

```java
public class MaryTask implements Runnable{

    ATM o;

    public MaryTask(ATM account){
        o = account;
    }

    public void run(){
        o.deposit(1500);
        o.withdraw(100);
    }
}
```

```
public class ATMThreads {

    ATM o;
    Thread tom;
    Thread mary;

    public ATMThreads(){
        o = new ATM();
        Thread tom = new Thread(new TomTask(o), "Tom");
        Thread mary = new Thread(new MaryTask(o), "Mary");
        tom.start();
        mary.start();
    }

    public static void main(String[] args){
        ATMThreads threads = new ATMThreads();
    }
}
```

Can we use two different “o” here?

```
No enough balance, Tom has to wait.
Mary is depositing.
Enough balance now, Mary can withdraw now.
Enough balance now, Tom can withdraw now.
```

Synchronization of Thread

- synchronized statement
  - Synchronize a block of code, not the method

```
// make all elements in the array non-negative
public static void abs(int[] values) {
    synchronized (values) {
        for (int i = 0; i < values.length; i++) {
            if (values[i] < 0)
                values[i] = -values[i];
        }
    }
}
```

Think

- Suppose:
  - 10 threads (t0 – t9), visit an synchronized method of an object competitively
  - But there is a promise, for example, the balance of account MUST higher than $1000 before withdraw (but deposit is always welcome)
  - If t0 take the key of account, but the balance is $100, t0 will hold the key forever, and no one can visit the account. (assuming that deposit() is also synchronized method)

wait() and notification

- A communication between object and threads
  - When blocked, current thread is supposed to give up key.
  - When unblocked, threads in waiting area are notified.
- In Java, wait() and notification is used to communicate

Wait and Notification

```
public class Account(){
    double balance;
    ...
    synchronized void withdraw(double amount){
        while(balance<1000){
            wait(); //条件不成立，不得取款，放弃对Account对象的控制权
        }
        balance -= amount;  //如果条件成立，则取款
    }

    synchronized void deposit(double amount){
        balance += amount;
        if(balance>1000)
            notifyAll(); //如果条件成立，则通知处于等待状态的线程
    }
}
```

Synchronization of Thread

- wait()
  - Each object has a wait method, inherited from java.lang.Object
  - wait() method ask current thread to give up exclusive control
  - wait() method give other thread a chance to visit the object
  - wait() / wait(long timeout)
- notifyAll() / notify()
  - notifyAll() wakes all waiting thread, thus, all waiting thread turn to Ready
  - notify() only wakes one of waiting thread, others remain blocked

Synchronization of Thread

- wait() / notifyAll() / notify()
  - The object must be locked before visit these methods
    - They can be used in synchronized method of an object
    - Or obj.wait() / obj.notifyAll() / obj.notify() in synchorized(obj){…}
  - Otherwise: java.lang.IllegalMonitorStateException

Difference Between wait() and sleep()

- wait()
  - From java.lang.Object
  - Must lock the object first
  - Let other thread visit the shared object
- sleep()
  - From java.lang.Thread
  - Other threads cannot visit the shared object

```
synchronized(obj){
    while(condition){
        obj.wait(1000);
    }
}

synchronized(obj){
    while(condition){
        Thread.sleep(1000);
    }
}
```

Sleep

```
public class MyTask implements Runnable{
    public synchronized void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + " started.");
        try{
            Thread.sleep(1000);
        }catch(Exception e){
            e.printStackTrace();
        }
        System.out.println(name + " ended.");
    }
}
```

Wait

```
public class MyTask implements Runnable{
    public synchronized void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + " started.");
        try{
            wait(1000);
        }catch(Exception e){
            e.printStackTrace();
        }
        System.out.println(name + " ended.");
    }
}
```

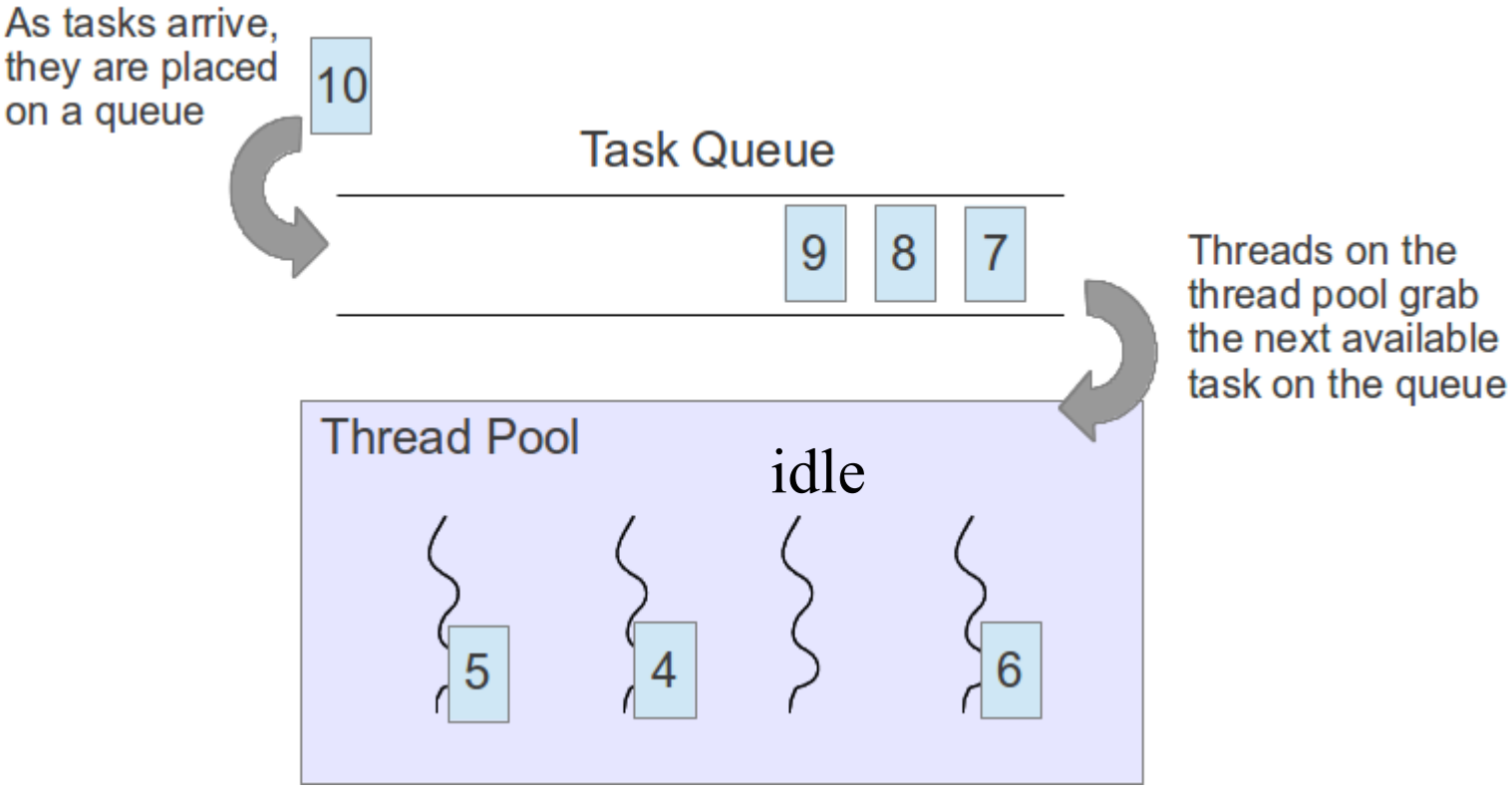
Main

```
public static void main(String[] args){
    long begin = System.currentTimeMillis(); //计时开始
    WaitAndSleepDemo test = new WaitAndSleepDemo();
    MyTask task = test.new MyTask(); //生成一个runnable task
    ArrayList<Thread> threadGroup = new ArrayList<Thread>();//建立一个Thread组
    for(int i=0; i<10; i++){
        Thread t = new Thread(task, "Thread"+i);
        threadGroup.add(t);
        t.start();
    }
    for(int i=0; i<threadGroup.size(); i++){ //使得每一个thread在main退出前运行完
        Thread t = threadGroup.get(i);
        try{
            t.join();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    System.out.println("time: " + (System.currentTimeMillis() - begin));
}
```

Think

- The same main method，different MyTask, what is the result?

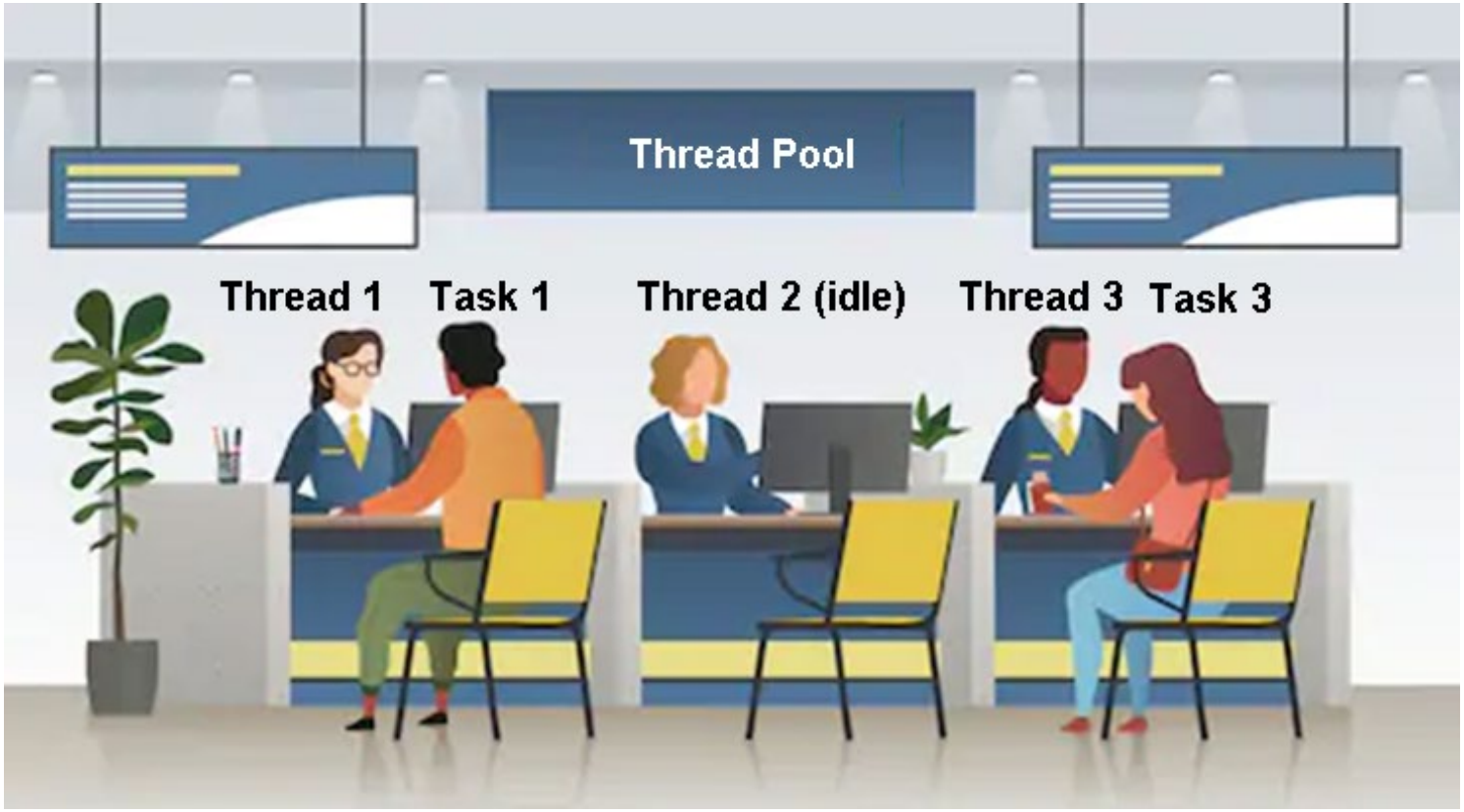
Thread Pool


As tasks arrive, they are placed on a queue
10
Task Queue
9
8
7
Threads on the thread pool grab the next available task on the queue
Thread Pool
idle
5
4
6

Thread Pool

- The Thread Pool generates some threads before receiving tasks;
- The Thread Pool takes a task received in a task queue, and passes it to a idle thread;
- After finishing the task, the thread does not go die, but become idle again;

Thread Pool



ExecutorService

Create a pool that creates threads as needed

```
public static void main(String[] args) {
    // create a pool that creates threads as needed;
    ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
    for (int i = 0; i < 100; i++) { // create 100 tasks;
        cachedThreadPool.execute(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName()
                        + " is executing.");
            }
        });
    }
```

Guess how many threads are created

```
...
pool-1-thread-28 is executing.
pool-1-thread-38 is executing.
pool-1-thread-27 is executing.
pool-1-thread-9 is executing.
pool-1-thread-34 is executing.
pool-1-thread-26 is executing.
pool-1-thread-29 is executing.
pool-1-thread-33 is executing.
pool-1-thread-11 is executing.
pool-1-thread-5 is executing.
pool-1-thread-30 is executing.
pool-1-thread-3 is executing.
pool-1-thread-32 is executing.
pool-1-thread-6 is executing.
pool-1-thread-24 is executing.
pool-1-thread-23 is executing.
pool-1-thread-31 is executing.
```

ExecutorService


```
public static void main(String[] args) {
    // create a pool that creates threads as needed;
    ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
    for (int i = 0; i < 100; i++) { // create 100 tasks;
        try{
            Thread.sleep(1000); // let the main thread take a break;
        }catch(Exception e) {
            e.printStackTrace();
        }
        cachedThreadPool.execute(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName()
                        + " is executing.");
            }
        });
    }
}
```

Guess how many threads are created

```
pool-1-thread-1 is executing.
pool-1-thread-1 is executing.
pool-1-thread-1 is executing.
pool-1-thread-1 is executing.
pool-1-thread-1 is executing.
pool-1-thread-1 is executing.
```

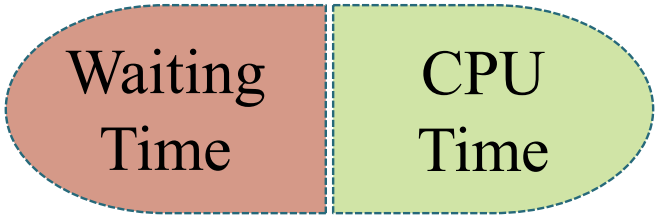
Why?

ExecutorService

Create a pool that creates fixed-number of threads

```
public static void main(String[] args) {
    // creates a pool with fix-number of threads;
    ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 10; i++) {
        fixedThreadPool.execute(new Runnable() {
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                            + " is executing");
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

Calculating the Pool Size



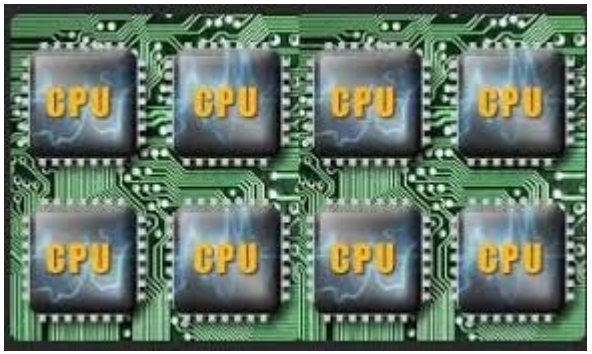
more threads  less threads

Calculating the Pool Size

#core

For CPU-intensive: $N + 1$

For IO-intensive: $2N + 1$



```java
// creates a pool with fix-number of threads;
int cores = Runtime.getRuntime().availableProcessors();
System.out.println("#core: " + cores);
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(2*cores+1);
```

ExecutorService

Creates a pool with timed execution (5 secs delay)

```
// creates a pool with size 1 for timed execution;
ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(1);
// delays for 5 secs before executing;
System.out.println("delay for 5 secs;");
scheduledThreadPool.schedule(new Runnable() {
    public void run() {

        System.out.println(Thread.currentThread().getName() + " is executing;");
    }
}, 5, TimeUnit.SECONDS);
```

ExecutorService

Creates a pool with periodic execution;
initially delays for 2 secs, and then executes with a period of 3 secs;

```
// creates a pool with size 5 for periodic execution;
ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);
// initially delays for 2 secs, and then executes with a period of 3 secs;
scheduledThreadPool.scheduleAtFixedRate(new Runnable() {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is executing;");
    }
}, 2, 3, TimeUnit.SECONDS);
```

ExecutorService

Creates a pool with single thread; executes the task in FIFO order

```
public static void main(String[] args) {
    /*
     * creates a pool with single thread;
     * executes the task in FIFO order
     */
    ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
    for (int i = 0; i < 10; i++) {
        final int index = i;
        singleThreadExecutor.execute(new Runnable() {
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName()
                            + " is executing task " + index);
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

Lab Work – Producer and Consumer

- Create two threads
- One of them prints numbers from 1 to 52
- The other prints letter A to Z
- Output: 12A34B56C...5152Z
- Please use synchorized and wait()

Lab Work - Producer and Consumer

生产者进入，消费者无法进入

```
public synchronized void printNum() {
    for (int i = 0; i < 26; i++) {
        while (flag) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        flag = !flag;
        System.out.print(++count);
        System.out.print(++count);
        notify();
    }
}
```

不符合特定条件，生产者退出等待

翻转控制标记

生产者完成生产任务退出，并通知消费者

Lab Work – Document Retrieval

- Given a keyword – for example: “stream”
- Find all your .java files that contain this keyword
- Requirements:
  - Use a thread to list all .java files in your path;
  - Create a ThreadPool with 10 threads
  - When you find a .java file in your path, throw it as a task into the ThreadPool
  - Benchmark the time efficiency of ThreadPool against a single thread

Lab Work – Document Retrieval

use Scanner to read a text file and judge if the file contains the key

```
String key = "stream";
Scanner sc = new Scanner(new File("e:/mytest.java")) ;
String line = "";
while(sc.hasNextLine()) {
    line = sc.nextLine();
    if(line.indexOf(key)>=0) {
        System.out.println("yes!");
        return;
    }
}
System.out.println("no!");
```

```
try{
    Thread.sleep(10000);
    threadPool.shutdown();
}catch(Exception e) {
    e.printStackTrace();
}
```

Lab Work – ATM 3.0

- ATM with synchronization
- A runnable task for adding interest