



## Chapter 7: Concurrent and Distributed Programming

### 7.1 Concurrency

### 并发

Wang Zhongjie  
[rainy@hit.edu.cn](mailto:rainy@hit.edu.cn)

May 16, 2020

# Outline

- **What is Concurrent Programming?**
- **Processes, threads, and time-slicing**
  - (1) Process
  - (2) Thread
  - (3) Starting a thread in Java
- **3 Interleaving and Race Condition**
  - (1) Time slicing
  - (2) Shared Memory among Threads
  - (3) Race Condition
  - (4) Message Passing Example
  - (5) Concurrency is hard to test and debug
  - (6) Some operations for interfering automatic interleaving of threads
- **Summary**

本章关注复杂软件系统的构造。  
这里的“复杂”包括两方面：

- (1) 多线程程序
- (2) 分布式程序

本节关注第一个方面：  
多线程程序的基本概念

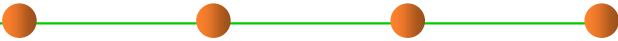
# Reading

- MIT 6.031: 19
- CMU 17-214: Oct 29、Nov 5、Nov 7
- Java编程思想: 第21章
- Java Concurrency in Practice: 第1-5章
- Effective Java: 第10章
- 代码整洁之道: 第13章



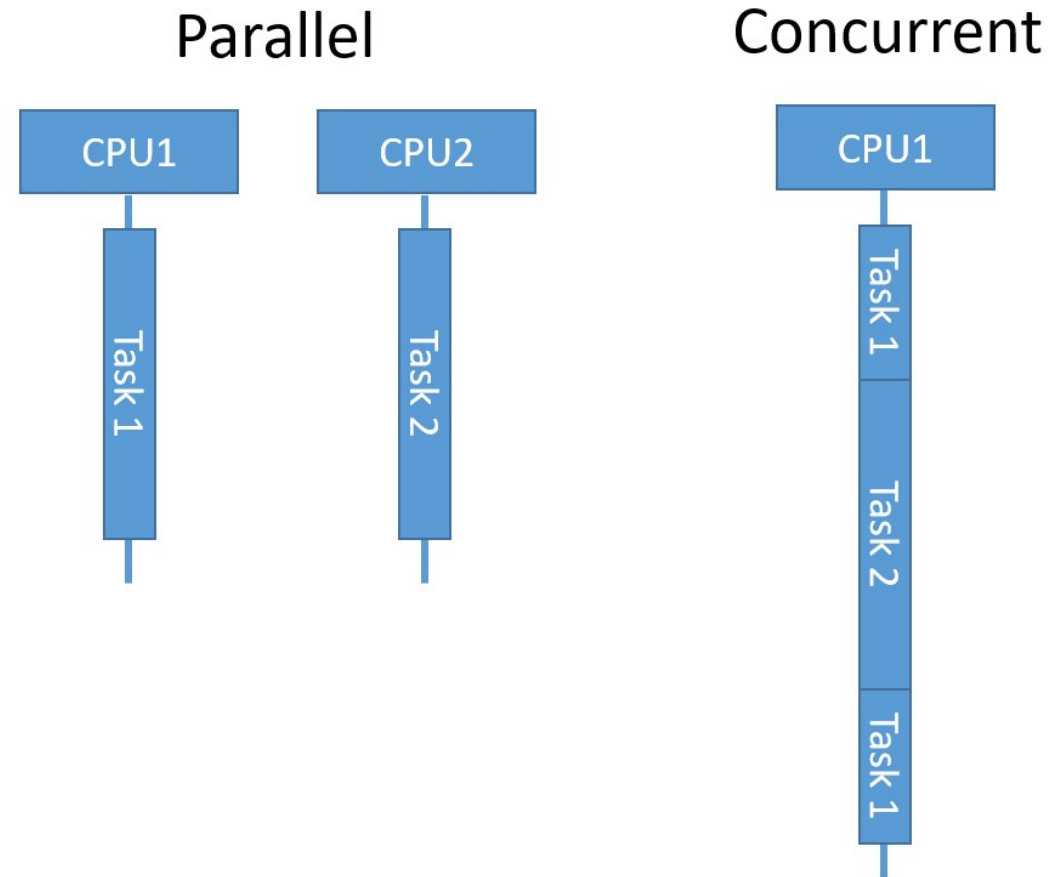


# 1 What is Concurrent Programming?



# Concurrency

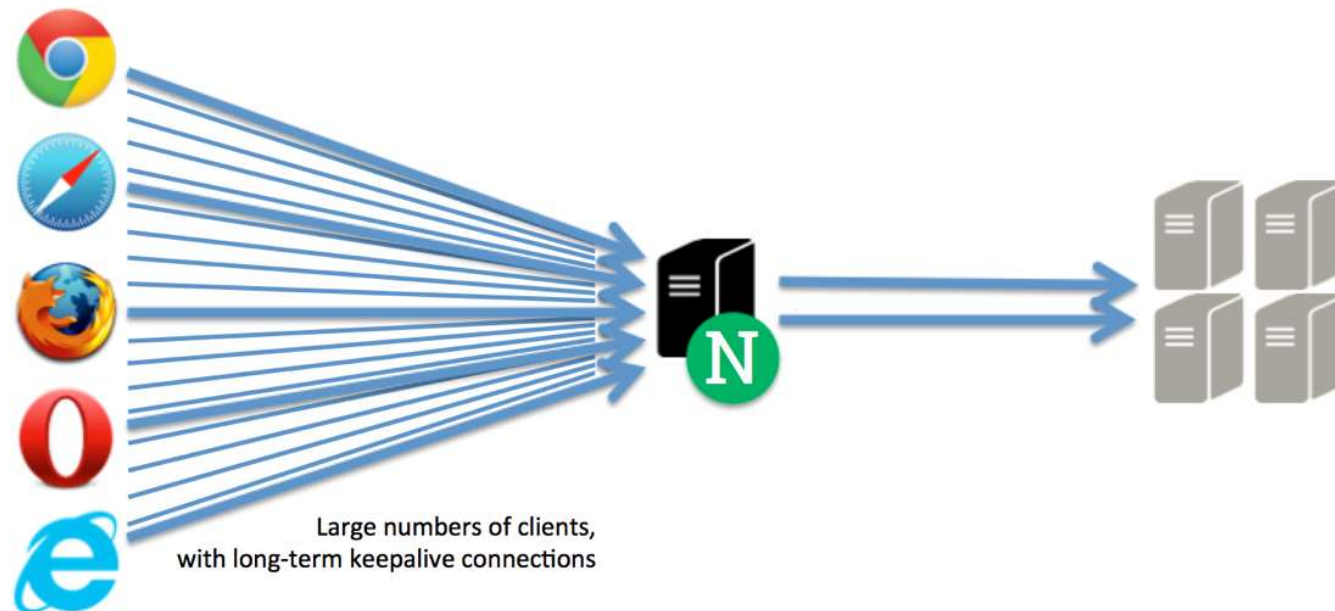
- **Concurrency**: multiple computations happen at the same time.
- Concurrency is everywhere in modern programming:
  - Multiple computers in a network 网络上的多台计算机
  - Multiple applications running on one computer 一台计算机上的多个应用
  - Multiple processors in a computer (today, often multiple processor cores on a single chip) 一个CPU上的多核处理器



# Concurrency

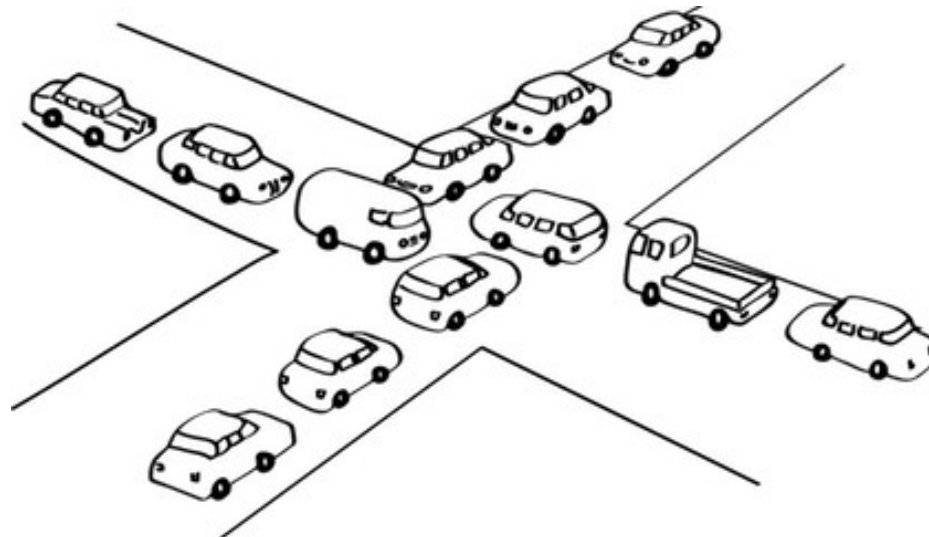
## ■ Concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users. 多用户并发请求服务器的计算资源
- Mobile apps need to do some of their processing in the cloud. App在手机端和在云端都有计算
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it. GUI的前端用户操作和后台的计算



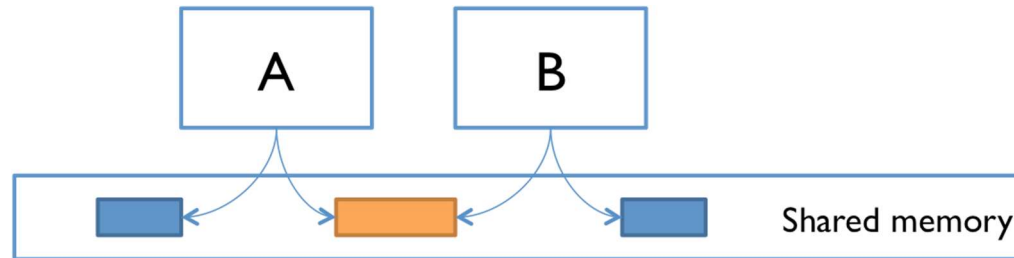
# Why “Concurrency”?

- Processor clock speeds are no longer increasing. 摩尔定律失效了
- Instead, we're getting more cores with each new generation of chips. “核” 变得越来越多
- In order to get a computation to run faster, we'll have to split up a computation into concurrent pieces. 为了充分利用多核和多处理器，需要将程序转化为并行执行

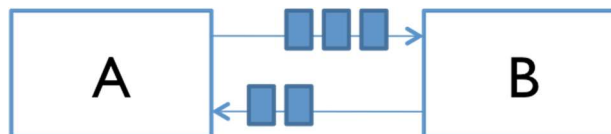


# Two Models for Concurrent Programming

- **Shared memory:** Concurrent modules interact by reading and writing shared objects in memory. 共享内存：在内存中读写共享数据



- **Message passing:** Concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. 消息传递：通过channel交换消息

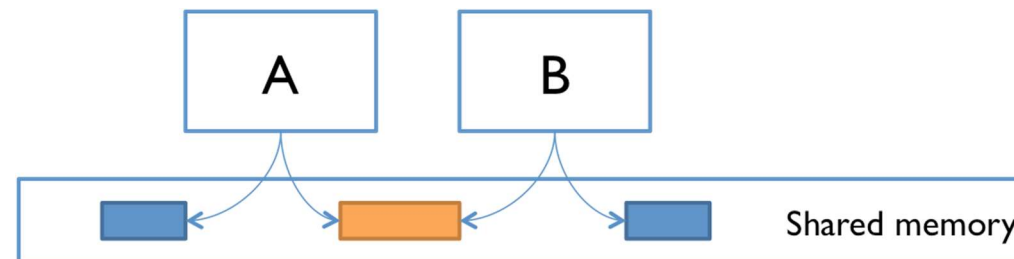




# Shared memory

## ■ Examples of the shared-memory model:

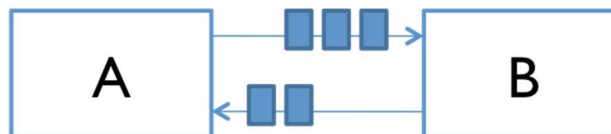
- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory. 两个处理器，共享内存
- A and B might be two programs running on the same computer, sharing a common file system with files they can read and write. 同一台机器上的两个程序，共享文件系统
- A and B might be two threads in the same Java program, sharing the same Java objects. 同一个Java程序内的两个线程，共享Java对象



# Message passing

## ■ Examples of message passing:

- A and B might be two computers in a network, communicating by network connections. 网络上的两台计算机，通过网络连接通讯
- A and B might be a web browser and a web server – A opens a connection to B and asks for a web page, and B sends the web page data back to A. 浏览器和Web服务器，A请求页面，B发送页面数据给A
- A and B might be an instant messaging client and server. 即时通讯软件的客户端和服务端
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like `ls | grep` typed into a command prompt. 同一台计算机上的两个程序，通过管道连接进行通讯





## 2 Processes, Threads, Time-slicing



# Process and Threads

- The message-passing and shared-memory models are about how concurrent **modules** communicate.
- The concurrent modules themselves come in two different kinds: **processes and threads**, two basic units of execution. 并发模块的类型：进程和线程
  - A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine's memory. 进程：私有空间，彼此隔离
  - A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches **return** statements). 线程：程序内部的控制机制



# (1) Process



# Process

- **The process abstraction is a virtual computer** (a self-contained execution environment with a complete, private set of basic run-time resources, in particular, memory). **进程：拥有整台计算机的资源**
  - It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.
- **Just like computers connected across a network, processes normally share no memory between them.** **多进程之间不共享内存**
  - A process can't access another process's memory or objects at all.
  - By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you've used in Java. **进程之间通过消息传递进行协作**

# Process

- Processes are often seen as synonymous with programs or applications. 一般来说，进程==程序==应用
  - However, what the user sees as a single application may in fact be a set of cooperating processes. 但一个应用中可能包含多个进程
- To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. OS支持的IPC机制(pipe/socket)支持进程间通信
  - IPC is used not just for communication between processes on the same system, but processes on different systems. 不仅是本机的多个进程之间，也可以是不同机器的多个进程之间。
- Most implementations of the Java virtual machine run as a single process. But a Java application can create additional processes using a `ProcessBuilder` object. JVM通常运行单一进程，但也可以创建新的进程。



## (2) Thread





# Thread and Multi-threaded programming

- Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*, and threads are sometimes called *lightweight process* 进程=虚拟机; 线程=虚拟CPU
  - Making a new thread simulates making a fresh processor inside the virtual computer represented by the process.
  - This new virtual processor runs the same program and shares the same resources (memory, open files, etc) as other threads in the process, i.e., “threads exist within a process”. 程序共享、资源共享, 都隶属于进程
- Threads are automatically ready for shared memory, because threads share all the memory in the process. 共享内存
  - It takes special effort to get “thread-local” memory that’s private to a single thread. 很难获得线程私有的内存空间 (how about thread stack?)
  - It’s also necessary to set up message-passing explicitly, by creating and using queue data structures. 通过创建消息队列在线程之间进行消息传递

# Threads vs. processes

Threads are **lightweight**

Threads **share** memory space

Threads require **synchronization**

Threads hold locks while mutating objects

It's **unsafe** to kill threads

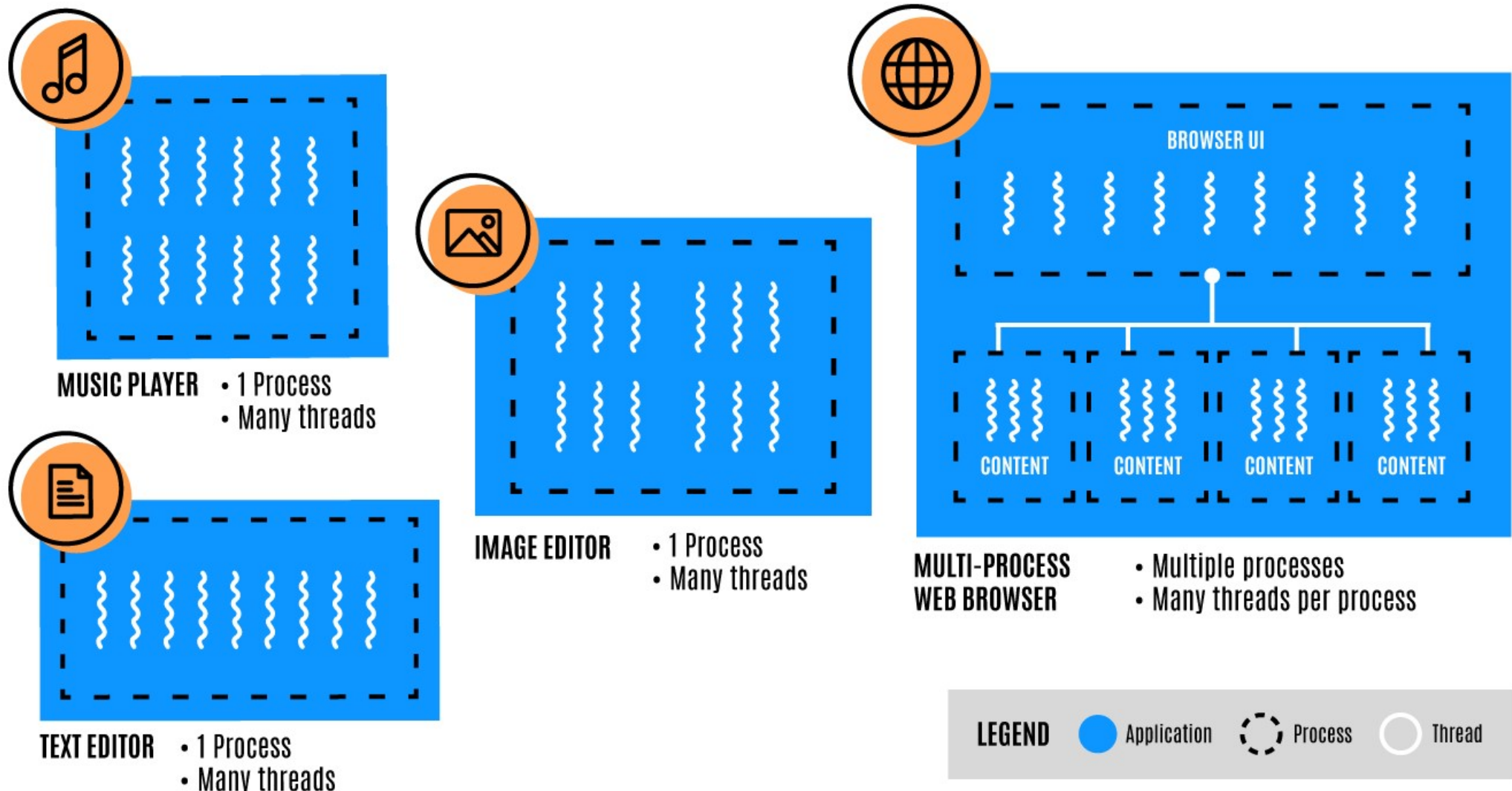
Processes **heavyweight**

Processes have **own**

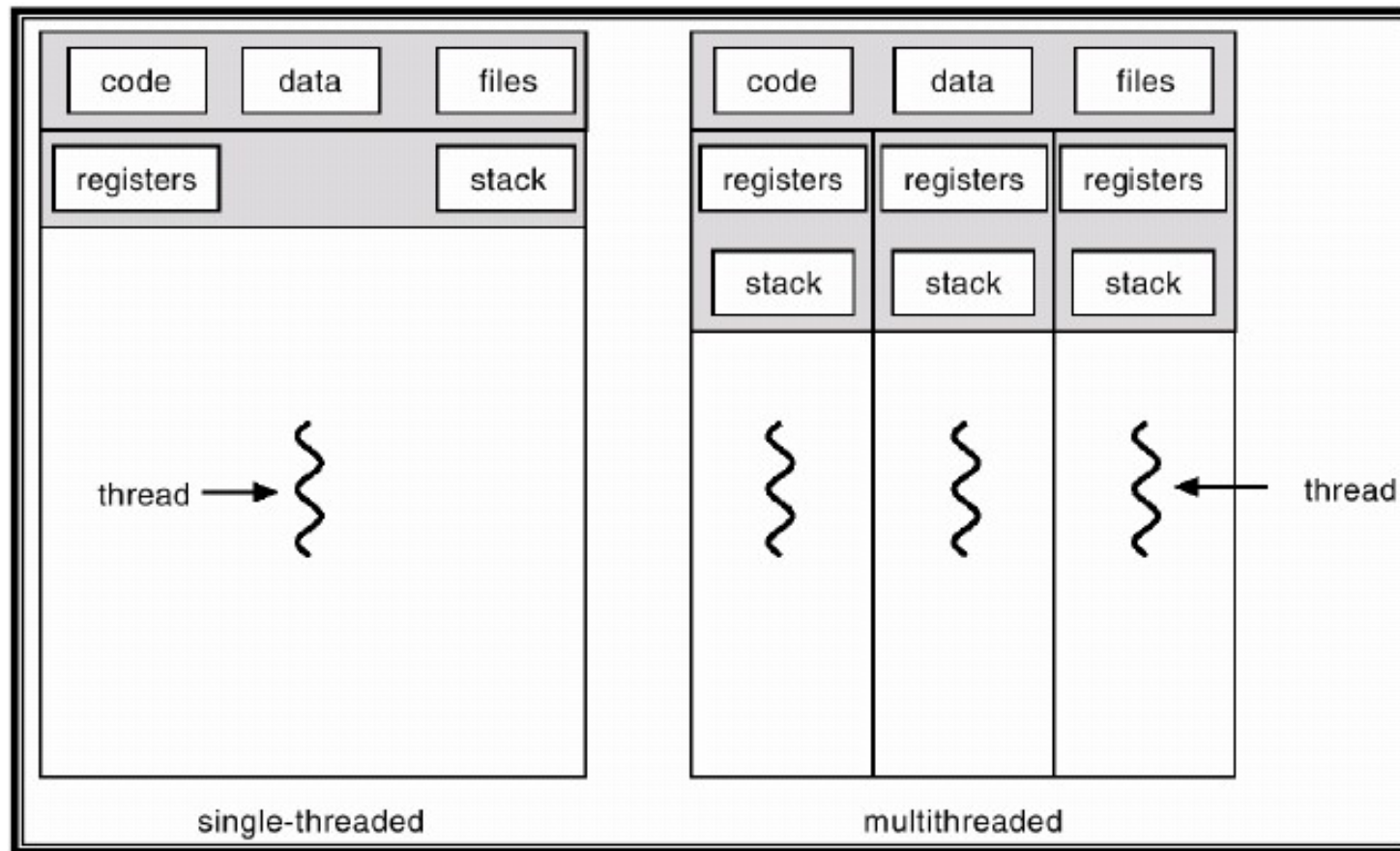
Processes **don't**

**Safe** to kill processes

# Threads vs. processes



# Single vs. Multiple Threads



# Why use threads?

- **Performance in the face of blocking activities**
  - Consider a web server
- **Performance on multiprocessors**
- **Cleanly dealing with natural concurrency**
- **In Java threads are a fact of life**
  - Example: garbage collector runs in its own thread ([recall: section 8-1](#))

Number of Threads	Seconds to run
1	22.0
2	13.5
3	11.7
4	10.8

# We are all concurrent programmers ...

- In order to utilize our multicore processors, we must write **multithreaded** code
- Good news: a lot of it is written for you
  - Excellent libraries exist (`java.util.concurrent`)
- Bad news: you still must understand fundamentals
  - To use libraries effectively
  - To debug programs that make use of them



# java.util.concurrent

- java.lang.**Object**
  - java.util.**AbstractCollection**<E> (implements java.util.Collection<E>)
  - java.util.**AbstractQueue**<E> (implements java.util.Queue<E>)
    - java.util.concurrent.**ArrayBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
    - java.util.concurrent.**ConcurrentLinkedQueue**<E> (implements java.util.Queue<E>, java.io.Serializable)
    - java.util.concurrent.**DelayQueue**<E> (implements java.util.concurrent.BlockingQueue<E>)
    - java.util.concurrent.**LinkedBlockingDeque**<E> (implements java.util.concurrent.BlockingDeque<E>, java.io.Serializable)
    - java.util.concurrent.**LinkedBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
    - java.util.concurrent.**LinkedTransferQueue**<E> (implements java.io.Serializable, java.util.concurrent.TransferQueue<E>)
    - java.util.concurrent.**PriorityBlockingQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
    - java.util.concurrent.**SynchronousQueue**<E> (implements java.util.concurrent.BlockingQueue<E>, java.io.Serializable)
  - java.util.**AbstractSet**<E> (implements java.util.Set<E>)
    - java.util.concurrent.**ConcurrentSkipListSet**<E> (implements java.lang.Cloneable, java.util.NavigableSet<E>, java.io.Serializable)
    - java.util.concurrent.**CopyOnWriteArraySet**<E> (implements java.io.Serializable)
  - java.util.concurrent.**ConcurrentLinkedDeque**<E> (implements java.util.Deque<E>, java.io.Serializable)
- java.util.concurrent.**AbstractExecutorService** (implements java.util.concurrent.ExecutorService)
  - java.util.concurrent.**ForkJoinPool**
  - java.util.concurrent.**ThreadPoolExecutor**
    - java.util.concurrent.**ScheduledThreadPoolExecutor** (implements java.util.concurrent.ScheduledExecutorService)
- java.util.**AbstractMap**<K,V> (implements java.util.Map<K,V>)
  - java.util.concurrent.**ConcurrentHashMap**<K,V> (implements java.util.concurrent.ConcurrentMap<K,V>, java.io.Serializable)
  - java.util.concurrent.**ConcurrentSkipListMap**<K,V> (implements java.lang.Cloneable, java.util.concurrent.ConcurrentNavigableMap<K,V>, java.io.Serializable)
- java.util.concurrent.**CompletableFuture**<T> (implements java.util.concurrent.CompletionStage<T>, java.util.concurrent.Future<V>)
- java.util.concurrent.**ConcurrentHashMap.KeySetView**<K,V> (implements java.io.Serializable, java.util.Set<E>)
- java.util.concurrent.**CopyOnWriteArrayList**<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
- java.util.concurrent.**CountDownLatch**
- java.util.concurrent.**CyclicBarrier**
- java.util.concurrent.**Exchanger**<V>
- java.util.concurrent.**ExecutorCompletionService**<V> (implements java.util.concurrent.CompletionService<V>)
- java.util.concurrent.**Executors**
- java.util.concurrent.**ForkJoinTask**<V> (implements java.util.concurrent.Future<V>, java.io.Serializable)
  - java.util.concurrent.**CountedCompleter**<T>
  - java.util.concurrent.**RecursiveAction**
  - java.util.concurrent.**RecursiveTask**<V>
- java.util.concurrent.**FutureTask**<V> (implements java.util.concurrent.RunnableFuture<V>)
- java.util.concurrent.**Phaser**
- java.util.**Random** (implements java.io.Serializable)
  - java.util.concurrent.**ThreadLocalRandom**
- java.util.concurrent.**Semaphore** (implements java.io.Serializable)
- java.lang.**Thread** (implements java.lang.Runnable)
  - java.util.concurrent.**ForkJoinWorkerThread**
- java.util.concurrent.**ThreadPoolExecutor.AbortPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
- java.util.concurrent.**ThreadPoolExecutor.CallerRunsPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
- java.util.concurrent.**ThreadPoolExecutor.DiscardOldestPolicy** (implements java.util.concurrent.RejectedExecutionHandler)
- java.util.concurrent.**ThreadPoolExecutor.DiscardPolicy** (implements java.util.concurrent.RejectedExecutionHandler)



## (3) Starting a thread in Java





# Thread

- **Multithreaded execution** is an essential feature of the Java platform.
  - Every application has at least one thread. 每个应用至少有一个线程
  - From the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads. 主线程, 可以创建其他的线程
- **Two ways to create a thread:**
  - (Seldom used) Subclassing **Thread**.  
从**Thread**类派生子类
  - (More generally used) Implement the **Runnable** interface and use the **new Thread(..)** constructor.  
从**Runnable**接口构造**Thread**对象

迄今为止所学过的-able:

- Comparable (and Comparator)
- Iterable (and Iterator)
- Observable (and Observer)
- Throwable
- Cloneable
- Runnable

# Ways to create a thread: Subclass **Thread**

```
public class Thread  
extends Object  
implements Runnable
```

## ■ Subclass **Thread**

- The **Thread** class itself implements **Runnable**, though its run method does nothing. An application can subclass **Thread**, providing its own implementation of **run()**.
- To invoke **Thread.start()** in order to start the new thread.

```
public class HelloThread extends Thread {  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
    public static void main(String args[]) {  
        HelloThread p = new HelloThread();  
        p.start();  
    }  
    //-----启动该线程的两个方式  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

除了必须要实现run()之后,和其他类一样,也可具有属性和方法

# Ways to create a thread: Provide a **Runnable** object

- **Provide a **Runnable** object**

- The **Runnable** interface defines a single method, **run()**, meant to contain the code executed in the thread.
- The **Runnable** object is passed to the **Thread** constructor.

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
}
```

# Ways to create a thread

- A very common idiom is starting a thread with an anonymous **Runnable**, which eliminates the named class:

```
new Thread(new Runnable() {  
    public void run() {  
        System.out.println("Hello");  
    }  
}).start();
```

1. A **Runnable** object is created
2. A **Thread** object is created
3. **Thread.start()** is called
4. The thread starts running
5. "Hello" is printed
6. The thread finishes running

- The **Runnable** interface represents the work to be done by a thread.

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

# Example 1

```
public class Parcae {  
    public static void main(String[] args) {  
        Thread nona = new Thread(new Runnable() {  
            public void run() { System.out.println("spinning"); }  
        });  
        nona.run(); // bug! called run instead of start  
  
        Runnable decima = new Runnable() {  
            public void run() { System.out.println("measuring"); }  
        };  
        decima.run(); // bug? maybe meant to create a Thread?  
        // ...  
    }  
}
```

The diagram highlights two bugs in the code. The first bug is that the `nona` thread is started using `run()` instead of `start()`. The second bug is that the `decima` Runnable is started using `run()` instead of creating a new thread and calling `start()`.

# Example 1

```
public class Parcae {  
  
    public static void main(String[] args) {  
        Thread nona = new Thread(new Runnable() {  
            public void run() { System.out.println("spinning"); }  
        });  
        nona.run(); ➔ nona.start();  
  
        Runnable decima = new Runnable() {  
            public void run() { System.out.println("measuring"); }  
        };  
        decima.run(); ➔ new Thread(decima).start();  
  
        // ...  
    }  
}
```

## Example 2

What is the maximum number of threads that might be running at the same time?

```
public class Moirai {  
    public static void main(String[] args) {  
        Thread clotho = new Thread(new Runnable() {  
            public void run() { System.out.println("spinning"); }  
        });  
        clotho.start();  
  
        new Thread(new Runnable() {  
            public void run() { System.out.println("measuring"); }  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() { System.out.println("cutting"); }  
        });  
    }  
}
```



# 3 Interleaving and Race Condition

交错和竞争





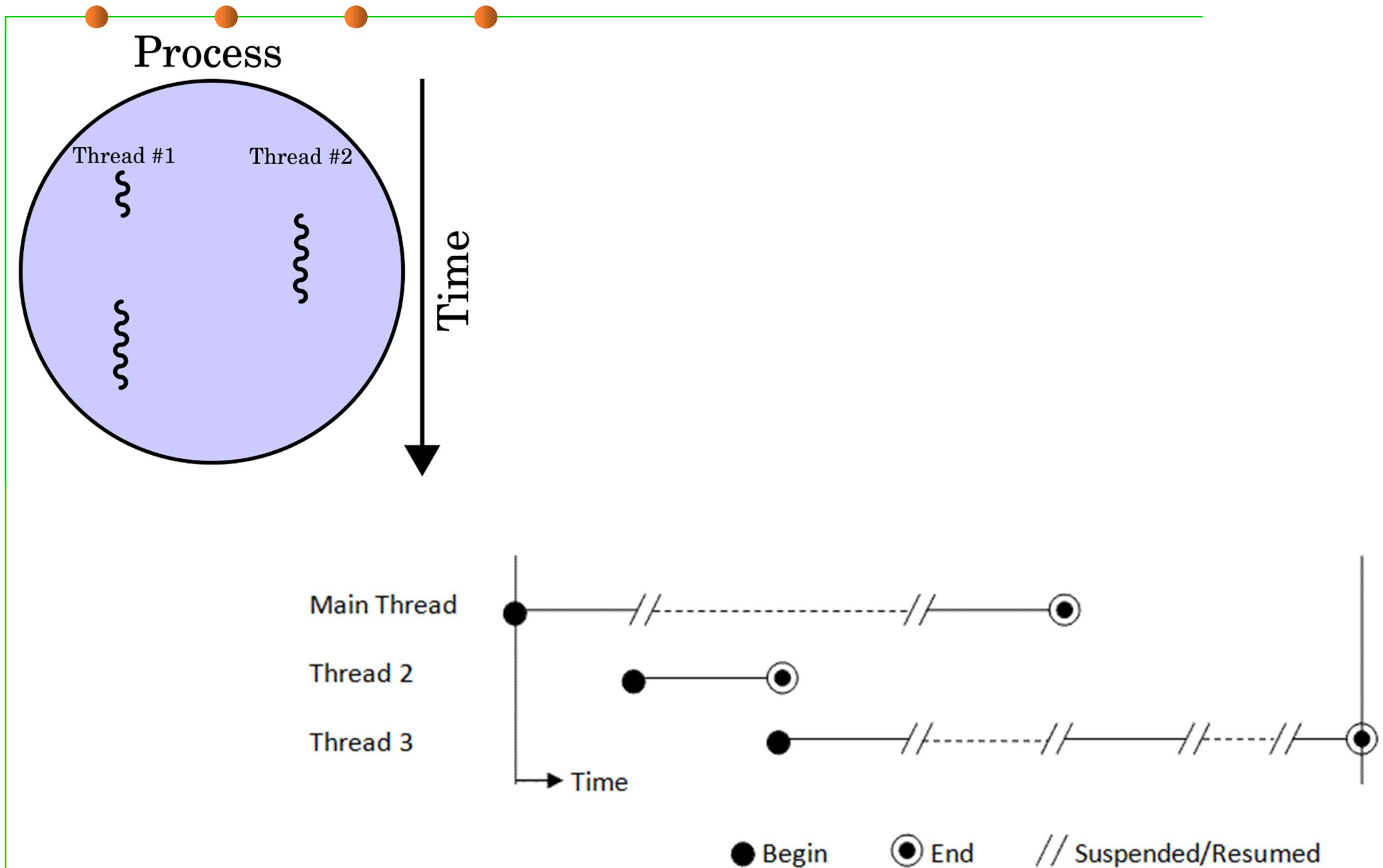
# (1) Time slicing



# Time slicing 时间分片

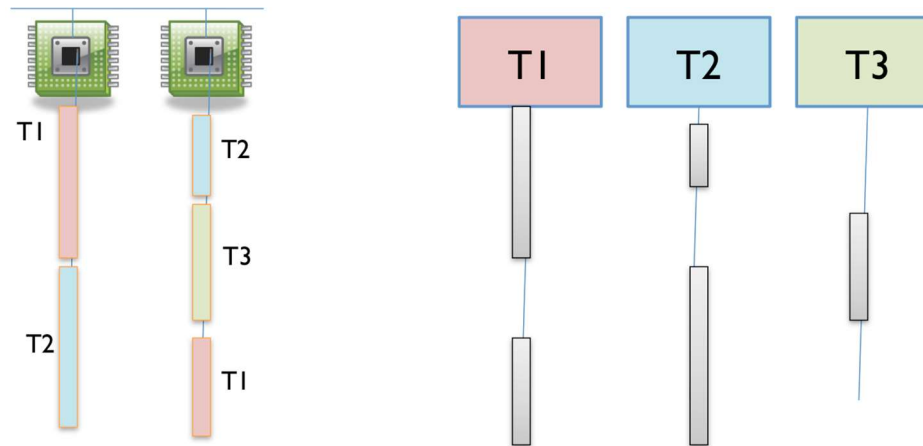
- In computer systems that have a single execution core, only one thread is actually executing at any given moment. 虽然有多线程，但只有一个核，每个时刻只能执行一个线程
  - Processing time for a single core is shared among processes and threads through an OS feature called **time slicing**. 通过时间分片，在多个进程/线程之间共享处理器
- Today's computer systems to have multiple processors or processors with multiple execution cores. So, how can I have many concurrent threads with only one or two processors in my computer? 即使是多核CPU，进程/线程的数目也往往大于核的数目
  - When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. 时间分片

# Time slicing 时间分片



# An example of time slicing

- Three threads T1, T2, and T3 might be time-sliced on a machine that has two actual processors.
  - At first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3.
  - Thread T2 simply pauses, until its next time slice on the same processor or another processor.



- On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time. 时间分片是由OS自动调度的

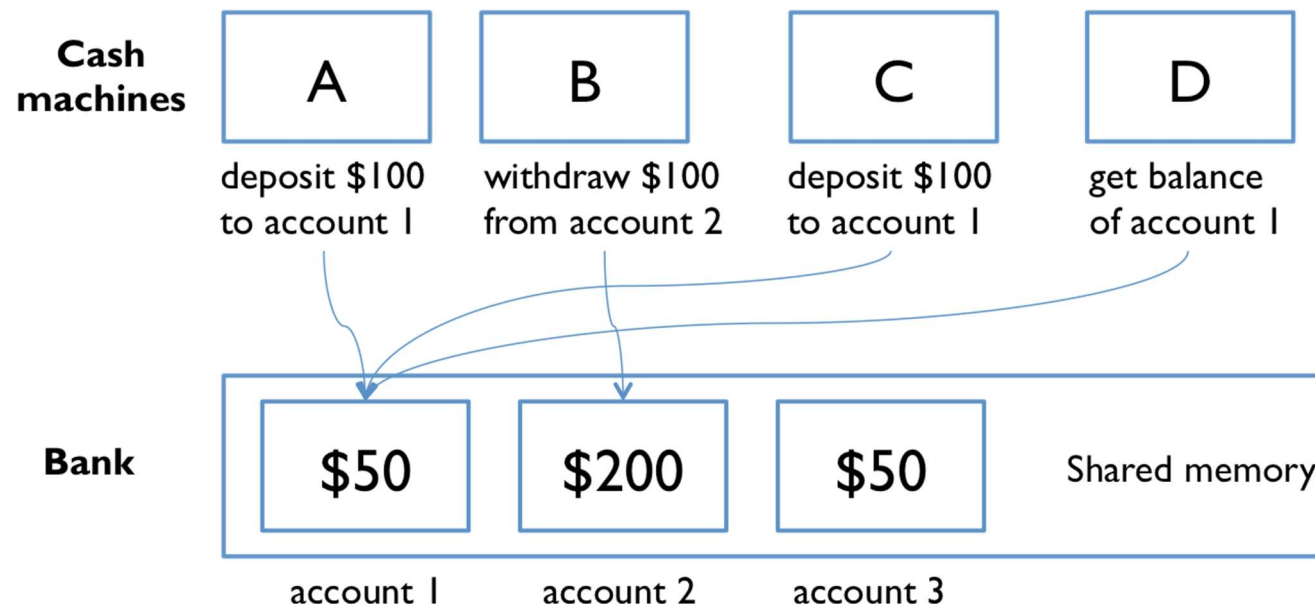


## (2) Shared Memory among Threads



# Shared Memory Example

- **Shared memory among threads could induce subtle bugs !!!**
- **Example:** a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.



# Shared Memory Example

- Simplify the bank down to a single account, with a dollar balance stored in the **balance** variable, and two operations **deposit** and **withdraw** that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

- Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

# Shared Memory Example

- Every transaction is just a one dollar deposit followed by a one-dollar withdrawal, **so it should leave the balance in the account unchanged.**

- Each machine is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
public static void cashMachine() {
    new Thread(new Runnable() {
        public void run() {
            for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
                deposit(); // put a dollar in
                withdraw(); // take it back out
            }
        }
    }).start();
}
```

- At the end of the day, regardless of how many cash machines were running, or how many transactions we processed, **we should expect the account balance to still be 0.** 按理说，余额应该始终为0
  - But if we run this code, we discover frequently that the balance at the end of the day is not 0. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then balance may not be zero at the end of the day. Why not?



# Interleaving

- Suppose two cash machines, A and B, are both working on a deposit at the same time.
- Here's how the `deposit()` step typically breaks down into low-level processor instructions:
- When A and B are running concurrently, these low-level instructions interleave with each other...

---

get balance (balance=0)

---

add 1

---

write back the result (balance=1)

---

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)



## (3) Race Condition



# Race Condition 竞争条件

## ■ The balance is now 1 – A's dollar was lost!

- A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

- **Race condition:** the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say **"A is in a race with B."**
  - Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings produce wrong answers – **violating postconditions or invariants.**

# Tweaking the code won't help

这三个版本是否都存在  
race condition?

- All these versions of the bank-account code exhibit the same race condition!
- You can't tell just from looking at Java code how the processor is going to execute it.
- You can't tell what the **atomic operations** will be.
  - It isn't atomic just because it's one line of Java.
  - It doesn't touch balance only once just because the balance identifier occurs only once in the line. 单行、单条语句都未必是原子的
- The Java compiler makes no commitments about what low-level operations it will generate from your code. 是否原子，由JVM确定
  - A typical modern Java compiler produces exactly the same code for all three of these versions!

```
// version 1
private static void deposit() { balance = balance + 1; }
private static void withdraw() { balance = balance - 1; }

// version 2
private static void deposit() { balance += 1; }
private static void withdraw() { balance -= 1; }

// version 3
private static void deposit() { ++balance; }
private static void withdraw() { --balance; }
```

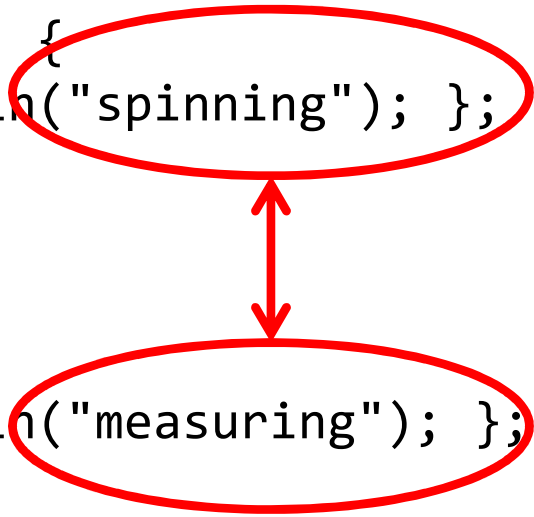
# Race Condition

- The key lesson is that you can't tell by looking at an expression whether it will be safe from race conditions.
- Race condition is also called **“Thread Interference”** 线程干扰  
<https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>

# Example 1

What are possible outputs from this program?

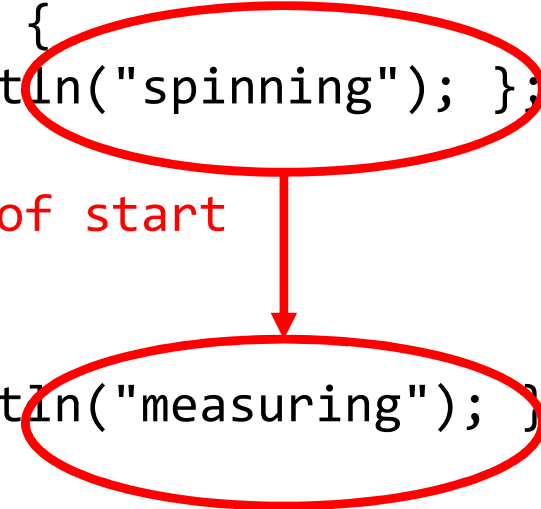
```
public class Moirai {  
    public static void main(String[] args) {  
        Thread clotho = new Thread(new Runnable() {  
            public void run() { System.out.println("spinning"); };  
        });  
        clotho.start();  
  
        new Thread(new Runnable() {  
            public void run() { System.out.println("measuring"); };  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() { System.out.println("cutting"); };  
        });  
        // bug! never started  
    }  
}
```



## Example 2

What are possible outputs from this program?

```
public class Parcae {  
  
    public static void main(String[] args) {  
        Thread nona = new Thread(new Runnable() {  
            public void run() { System.out.println("spinning"); }  
        });  
        nona.run(); // bug! called run instead of start  
  
        Runnable decima = new Runnable() {  
            public void run() { System.out.println("measuring"); }  
        };  
        decima.run(); // bug? maybe meant to create a Thread?  
  
        // ...  
    }  
}
```



## Example 3

```
private static int x = 1;

public static void methodA() {
    x *= 2;
    x *= 3;
}

public static void methodB() {
    x *= 5;
}
```

```
x *= 5;
x = x * 5;
```

- (1) Read x from the static variable x;
- (2) Multiple it by 5;
- (3) Store the result into the variable;

Suppose methodA and methodB run **sequentially**, i.e. first one and then the other. What is the final value of x?

Now suppose methodA and methodB run **concurrently**, so that their instructions might interleave arbitrarily. Which of the following are possible final values of x?

- 1
- 2
- 5
- 6
- 10
- 30
- 150



# Example 3

```
private static int x = 1;
```

```
public static void methodA() {
```

```
1.  read x;           x=1
```

```
2.  p = x*2;
```

```
3.  x = p;           x=2
```

```
4.  q = x*3;         q=6
```

```
5.  x = q;           x=6
```

```
}
```

```
public static void methodB() {
```

```
6.  read x;           x=2
```

```
7.  r = x*5;         r=10
```

```
8.  x = r;           x=10
```

```
}
```

1-2-3-4-5 - 6-7-8

30

6-7-8 - 1-2-3-4-5

30

1-2 - 6-7-8 - 3-4-5

6

6-7 - 1-2-3-4-5 - 8

5



1-2-3 - 6-7 - 4-5 - 8

10

1-2 - 6-7 - 3-4 - 8 - 5

6

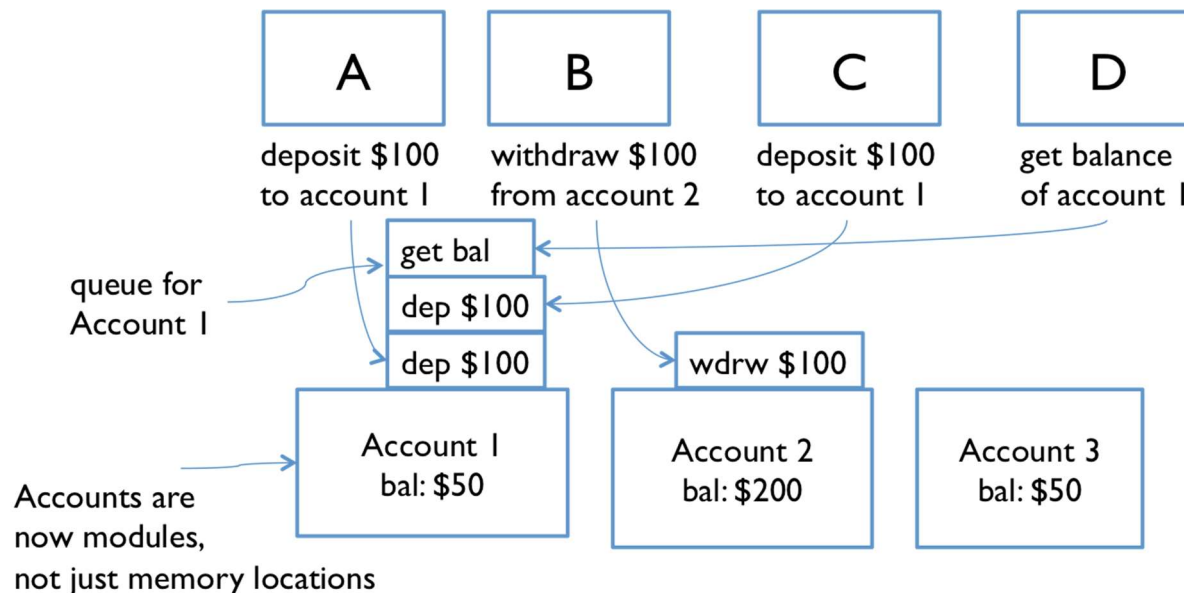


## (4) Message Passing Example



# Message Passing Example

- Now not only are the cash machine modules, **but the accounts are modules, too.**
- **Modules interact by sending messages to each other.**
  - Incoming requests are placed in a queue to be handled one at a time.
  - The sender doesn't stop working while waiting for an answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.



# Can message-passing solve race condition?

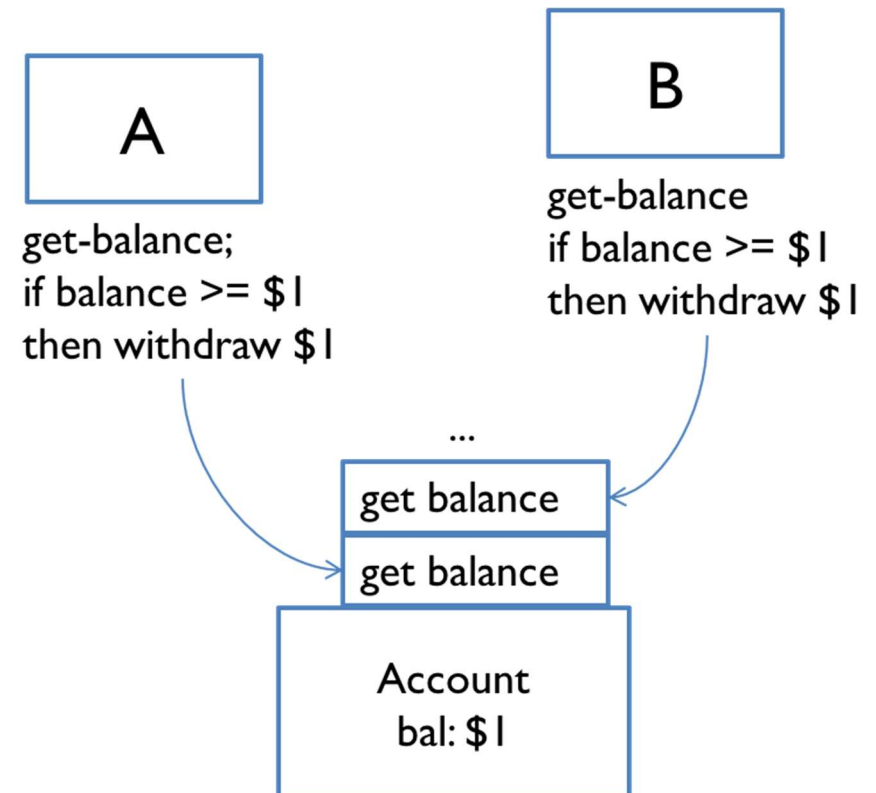
- Unfortunately, message passing doesn't eliminate the possibility of race conditions. 消息传递机制也无法解决竞争条件问题
  - Suppose each account supports get-balance and withdraw operations, with corresponding messages.
  - Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account.
  - They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties.

get-balance

if balance  $\geq$  1 then withdraw 1

# Can message-passing solve race condition?

- The problem is again interleaving, but this time interleaving of the *messages* sent to the bank account, rather than the *instructions* executed by A and B. 仍然存在消息传递时间上的交错
- If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawing the account?





(5) Concurrency is hard to test and debug



# Concurrency is hard to test and debug !

- It's very hard to discover race conditions using testing. 很难测试和调试因为竞争条件导致的bug
  - Even once a test has found a bug, it may be very hard to localize it to the part of the program causing it. ----WHY?
- Concurrency bugs exhibit very poor reproducibility. 因为interleaving的存在，导致很难复现bug
  - It's hard to make them happen the same way twice.
  - Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment.
  - Delays are caused by other running programs, other network traffic, OS scheduling decisions, variations in processor clock speed, etc.
  - Each time you run a program containing a race condition, you may get different behavior.

# Heisenbugs and Bohrbugs

Almost all bugs in sequential programming are bohrbugs.

A heisenbug is a software bug that seems to disappear or alter its behavior when one attempts to study it.

## *Heisenbugs*

Nondeterministic and hard to reproduce



Werner Karl Heisenberg (1901-1976)  
Nobel Prize in Physics in 1932

## *Bohrbugs*

Showing up repeatedly whenever you look at it.



Niels Bohr (1885-1962)  
Nobel Prize in Physics in 1922



# Concurrency is hard to test and debug !

- A heisenbug may even disappear when you try to look at it with `println` or debugger! 增加`print`语句甚至导致这种bug消失! ~
  - The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving. 神奇的原因
- So inserting a simple `print` statement into the `cashMachine()`:

```
private static void cashMachine() {  
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {  
        deposit(); // put a dollar in  
        withdraw(); // take it back out  
        System.out.println(balance); // makes the bug disappear!  
    }  
}
```

...and suddenly **the balance is always 0**, as desired, and the bug appears to disappear. But it's only masked, not truly fixed.



## (6) Some operations for interfering automatic interleaving of threads

---

利用某些方法调用来主动影响线程之间的interleaving关系

# Thread.sleep()

- **Pausing Execution with Thread.sleep(time):** causes the current thread to suspend execution for a specified period. 线程的休眠
  - This is an efficient means of making processor time available to the other threads or other applications that might be running on the same computer. 将某个线程休眠，意味着其他线程得到更多的执行机会
  - Thread sleep doesn't lose any monitors or locks current thread has acquired. 进入休眠的线程不会失去对现有monitor或锁的所有权

```
for (int i = 0; i < n; i++) {  
    //Pause for 4 seconds  
    Thread.sleep(4000);  
    //Print a message  
    System.out.println(msg[i]);  
}
```

# Thread.interrupt()

- A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted using **interrupt()** method  
向线程发出中断信号
  - **t.interrupt()**      在其他线程里向t发出中断信号
- To check if a thread is interrupted, using **isInterrupted()** method. 检查线程是否被中断
  - **t.isInterrupted()**    检查t是否已在中断状态中
- An interrupt is an indication to a thread that it should stop what it is doing and do something else. 当某个线程被中断后，一般来说应停止其**run()**中的执行，取决于程序员在**run()**中处理
  - It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. 一般来说，线程在收到中断信号时应该中断，直接终止

但是，线程收到其他线程发来的中断信号，并不意味着一定要“停止” ...

# Thread.interrupt()

```
class Task implements Runnable{
    private double d = 0.0;

    public void run() {
        try{
            while (true) {
                for (int i = 0; i < 900000; i++)
                    d = d + (Math.PI + Math.E) / d;
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {return;}
    }
}
```

在sleep()的时候检测是否收到别人的中断信号。若是，抛出异常

正常运行期间，即使接收到中断信号，也不理会

进入异常处理后，线程才真正终止

```
Thread t = new Thread(new Task());
t.start();
Thread.sleep(100); //当前线程休眠
t.interrupt(); //试图中断t线程
```

# Thread.interrupt()

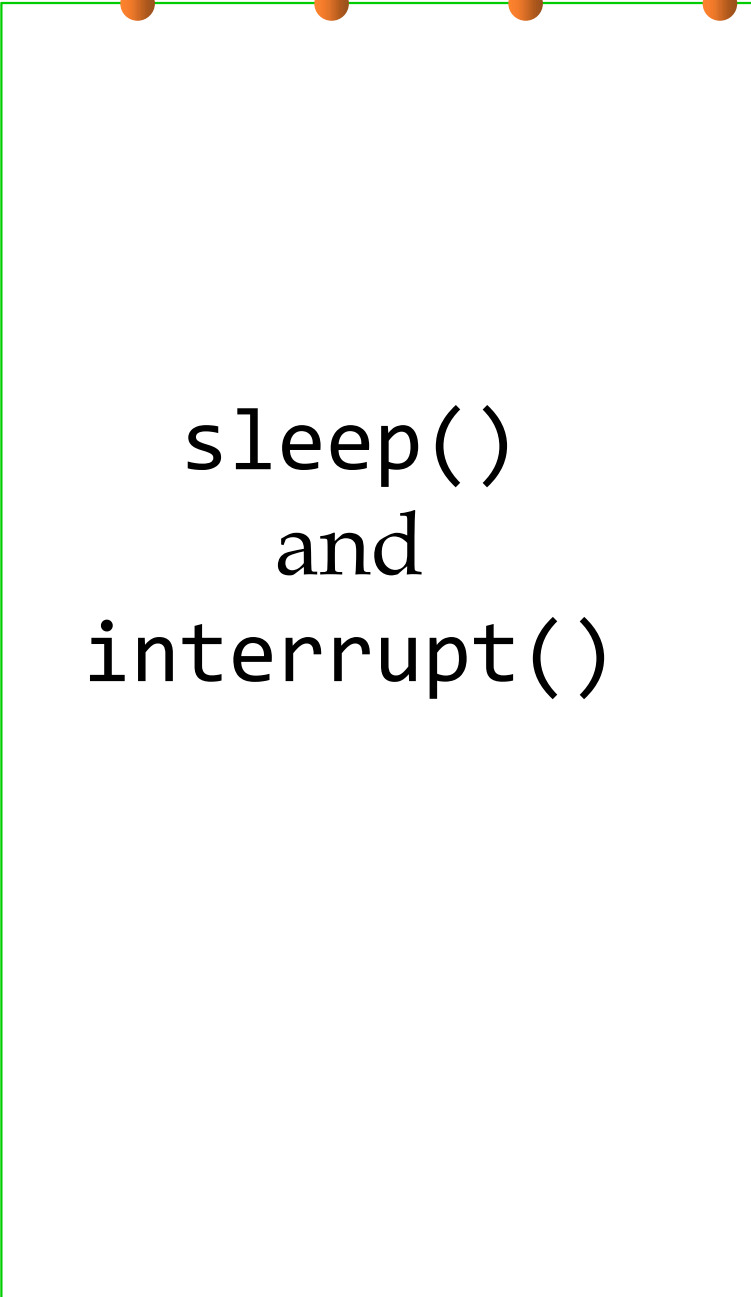
由此可见，如果不sleep()/其他几个特定操作，无法检测到中断信号，相当于对别人发来的中断信号“置之不理”——这“不友好”！

```
class Task implements Runnable{
    private double d = 0.0;
    public void run() {
        try {
            //检查线程是否收到中断信号
            while (!Thread.interrupted()) {
                Thread.sleep(20);
                for (int i = 0; i < 900000; i++)
                    d = d + (Math.PI + Math.E) / d;
            }
        } catch (InterruptedException e) {
            System.out.println("Exiting by Exception");
        }
    }
}
```

若进入while之后发生中断，通过sleep()检测中断信号，抛出异常，停止线程

另一种方案：不用  
sleep()

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

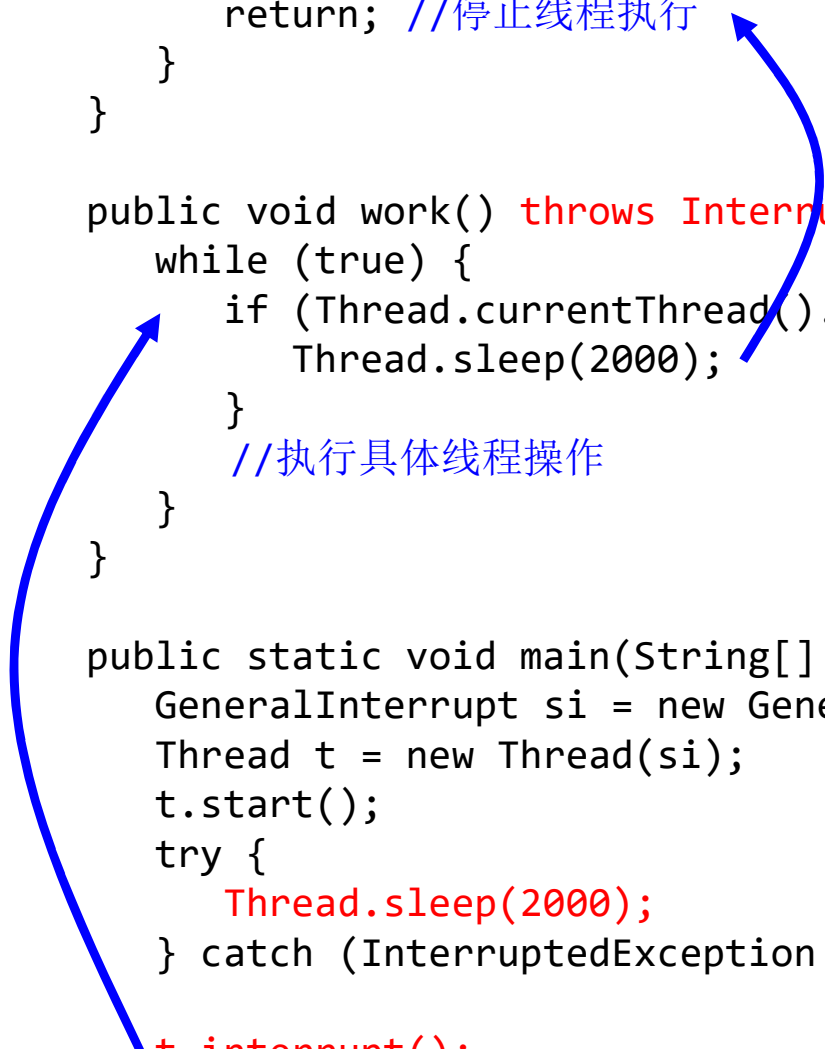


sleep()  
and  
interrupt()

```
public class GeneralInterrupt implements Runnable {
    public void run() {
        try {
            work();
        } catch (InterruptedException x) {
            return; //停止线程执行
        }
    }

    public void work() throws InterruptedException {
        while (true) {
            if (Thread.currentThread().isInterrupted()) {
                Thread.sleep(2000);
            }
            //执行具体线程操作
        }
    }

    public static void main(String[] args) {
        GeneralInterrupt si = new GeneralInterrupt();
        Thread t = new Thread(si);
        t.start();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException x) { }
        t.interrupt();
    }
}
```



# sleep() and interrupt()

```
class TestInterruptingThread3 extends Thread{
```

```
    public void run(){  
        for(int i=1;i<=1000;i++)  
            System.out.println(i);  
    }
```

This thread did not receive the interrupt signal, so it does nothing but continue running until finished.

```
    public static void main(String args[]){  
        TestInterruptingThread3 t1=new TestInterruptingThread3();  
        t1.start();  
        t1.interrupt();  
    }
```

What will happen?

```
}
```



# sleep() and interrupt()

```
class TestInterruptingThread2 extends Thread{
    public void run(){
        try{
            Thread.sleep(1000);
            System.out.println("task");
        }catch(InterruptedException e){
            System.out.println("Exception handled "+e);
        }
        System.out.println("thread is running...");
    }

    public static void main(String args[]){
        TestInterruptingThread2 t1=new TestInterruptingThread2();
        t1.start();
        t1.interrupt();
    }
}
```

It does really receive the interrupt signal, but does not end the thread but continue..

What will happen?

# sleep() and interrupt()

```
public class GeneralInterrupt extends Thread {
    public void run() {
        for (int i = 1; i <= 2; i++) {
            if (Thread.interrupted())
                System.out.println("code for interrupted thread");
            else
                System.out.println("code for normal thread");
        }
    }

    public static void main(String args[]) {
        GeneralInterrupt t1 = new GeneralInterrupt();
        GeneralInterrupt t2 = new GeneralInterrupt();

        t1.start();
        t1.interrupt();
        t2.start();
    }
}
```

What will happen?

# Thread.yield()

- This static method is essentially used to notify the system that the current thread is willing to “give up the CPU” for a while. 使用该方法，线程告知调度器：我可以放弃CPU的占用权，从而可能引起调度器唤醒其他线程。
  - The general idea is that: the thread scheduler will select a different thread to run instead of the current one.
- This is a seldom used method in thread programming because the scheduling should be up to the scheduler’s responsibility. 尽量避免在代码中使用

```
public void run() {  
    ...  
    for (int i = 0; i < 5; i++) {  
        if ((i % 5) == 0)  
            Thread.yield();  
    }  
}
```

# Thread.join()

- The **join()** method is used to hold the execution of currently running thread until the specified thread is dead (finished execution). 让当前线程保持执行，直到其执行结束
  - In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads. 一般不需要这种显式指定线程执行次序
  - by using **join()** method, we can make one thread to wait for another.

```
public class JoinExample2 {  
    public static void main(String[] args) {  
        Thread th1 = new Thread(new MyClass2(), "th1");  
        Thread th2 = new Thread(new MyClass2(), "th2");  
        Thread th3 = new Thread(new MyClass2(), "th3");  
  
        th1.start();  
        th2.start();  
        th3.start();  
    }  
}
```

有多少种可能的执行次序？

# join()

```
public class JoinExample {
    public static void main(String[] args) {
        Thread th1 = new Thread(new MyClass(), "th1");
        Thread th2 = new Thread(new MyClass(), "th2");
        Thread th3 = new Thread(new MyClass(), "th3");

        th1.start();

        try {
            th1.join();
        } catch (InterruptedException ie) {}

        th2.start();

        try {
            th2.join();
        } catch (InterruptedException ie) {}

        th3.start();

        try {
            th3.join();
        } catch (InterruptedException ie) {}
    }
}
```

To cause current thread (main) to pause execution until th1 terminates.

现在有多少种可能的执行次序？

join() will also throw InterruptedException if it is interrupted by other threads  
执行该操作时，也会检测其他线程发来的中断信号



# 4 Summary



# A short summary

- **Concurrency:** multiple computations running simultaneously
- **Shared-memory & message-passing** paradigms
- **Processes & threads**
  - Process is like a virtual computer; thread is like a virtual processor
- **Race conditions**
  - When correctness of result (postconditions and invariants) depends on the relative timing of events
  - Multiple threads **sharing the same mutable variable** without coordinating what they're doing.
  - This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

# A short summary

- These ideas connect to our key properties of good software mostly in bad ways.
- **Concurrency is necessary but it causes serious problems for correctness:**
  - **Safe from bugs.** Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
  - **Easy to understand.** Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design your code in such a way that programmers don't have to think about interleaving at all.





The end

May 16, 2020