哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 7: Concurrent and Distributed Programming

7.3 Locks and Synchronization

锁与同步

May 17, 2020

Objective of this lecture

- **Understand how a lock is used to protect shared mutable data**
- **Be able to recognize deadlock and know strategies to prevent it**
- **Know the monitor pattern and be able to apply it to a data type**

Outline

- **Synchronization**
- **Locking**
- **Atomic operations**
- **Liveness: deadlock, starvation and livelock**
- `wait()`, `notify()`, **and** `notifyAll()`
- **Summary**

本章关注复杂软件系统的构造。
这里的“复杂”包括三方面：
（1）多线程程序
（2）分布式程序
（3） GUI 程序

本节关注第一个方面：
如何设计threadsafe的ADT

Reading

- MIT 6.031：21
- CMU 17-214：Nov 12
- 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

HARBIN INSTITUTE OF TECHNOLOGY

1 Synchronization

Recall

- **Thread safety for a data type or a function:** behaving correctly when used from multiple threads, regardless of how those threads are executed, without additional coordination. 线程安全不应依赖于偶然

Principle: the correctness of a concurrent program should not depend on accidents of timing.

- **The correctness of a concurrent program should not depend on accidents of timing.**

Recall

- **There are four strategies for making code safe for concurrency:**
  - Confinement : don’t share data between threads.
  - Immutability : make the shared data immutable.
  - Use existing threadsafe data types : use a data type that does the coordination for you.
  - **Synchronization**: prevent threads from accessing the shared data at the same time. This is what we use to implement a threadsafe type, but we didn’t discuss it at the time.
- **前三种策略的核心思想：**
  - 避免共享 ➔ 即使共享，也只能读/不可写(immutable) ➔ 即使可写(mutable)，共享的可写数据应自己具备在多线程之间协调的能力，即“使用线程安全的mutable ADT”
  - 缺陷：不能用全局rep共享数据➔只能“读”共享数据，不能写➔可以共享“读写”，但只有单一方法是安全的，多个方法调用就不安全了

Synchronization and Locks

- Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, reproduce, and debug — we need a way for concurrent modules that share memory to **synchronize** with each other.
- **很多时候，无法满足上述三个条件…➔要读写共享数据，且线程中的读写操作复杂**

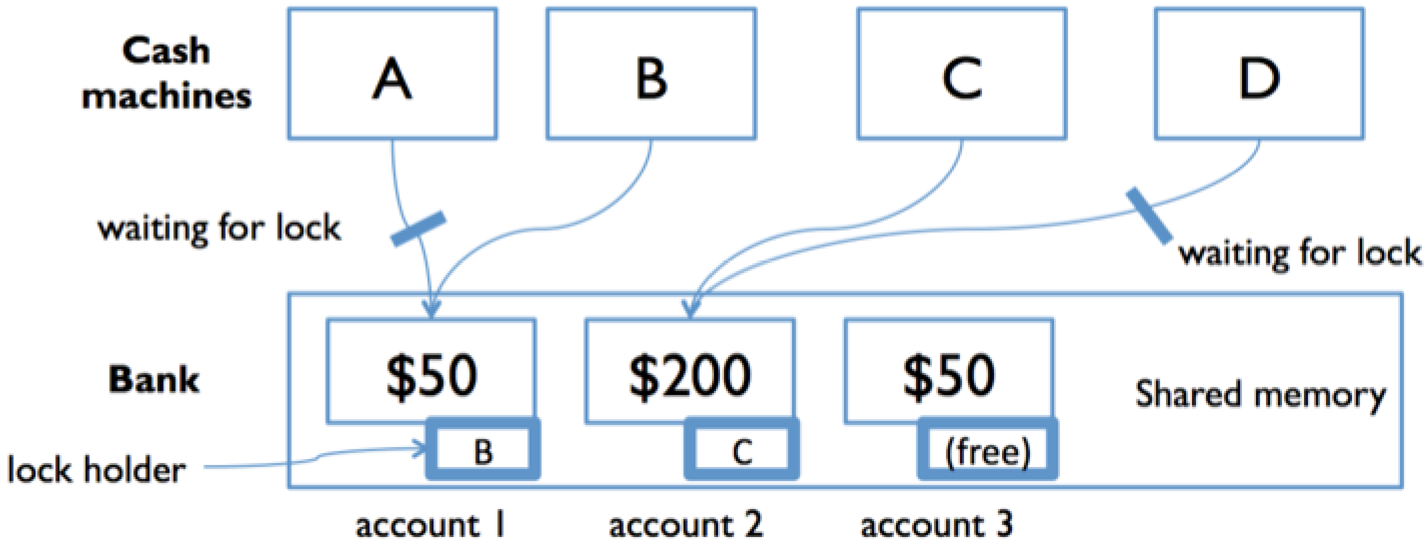
- **The fourth strategy for making code safe for concurrency:**
  - Synchronization 同步和锁: prevent threads from accessing the shared data at the same time.

- **程序员来负责多线程之间对mutable数据的共享操作，通过“同步”策略，避免多线程同时访问数据**

Synchronization and Locks

- **Locks** are one synchronization technique.
  - A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread tells other threads: "I'm changing this thing, don't touch it right now."
  - 使用锁机制，获得对数据的独家mutation权，其他线程被阻塞，不得访问
- Using a lock tells the compiler and processor that you're using shared memory concurrently, so that registers and caches will be flushed out to shared storage, ensuring that the owner of a lock is always looking at up-to-date data.
- **Blocking** means, in general, that a thread waits (without doing further work) until an event occurs.

Bank account example



- Both A and B are trying to access account 1.
- Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock.
- This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 Locking

Locking

- **Locks are so commonly-used that Java provides them as a built-in language feature. Lock是Java语言提供的内嵌机制**
  - Every object has a lock implicitly associated with it — a `String`, an array, an `ArrayList`, and every class and all of their instances have a lock. 每个object都有相关联的lock
  - Even a humble `Object` has a lock, so bare `Object` are often used for explicit locking:

```
Object lock = new Object();
```

- **You can't call `acquire` and `release` on Java's intrinsic locks, however. Instead you use the `synchronized` statement to acquire the lock for the duration of a statement block:**

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

Locking

- Synchronized regions like this provide **mutual exclusion 互斥**: only one thread at a time can be in a synchronized region guarded by a given object's lock. 拥有lock的线程可独占式的执行该部分代码
- In other words, you are back in **sequential programming** world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object.

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

Locks guard access to data

- **Locks are used to guard a shared data variable. Lock保护共享数据**
  - If all accesses to a data variable are guarded (surrounded by a `synchronized` block) by the same `lock` object, then those accesses will be guaranteed to be atomic — uninterrupted by other threads.
- **Acquiring the lock associated with object `obj` using**

```
synchronized (obj) { ... }
```

  - It prevents other threads from entering a `synchronized(obj)` block, until thread `t` finishes its synchronized block.
- **Locks only provide mutual exclusion with other threads that acquire the same lock. All accesses to a data variable must be guarded by the same lock. 注意：要互斥，必须使用同一个lock进行保护**
  - You might guard an entire collection of variables behind a single lock, but all modules must agree on which lock they will all acquire and release.

Monitor pattern

- **When you are writing methods of a class, the most convenient lock is the object instance itself, i.e.** `this`**. 用ADT自己做lock**
- **As a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside** `synchronized(this)`**.**
- **Monitor pattern:** a monitor is a class whose methods are mutually exclusive, so that only one thread can be inside an instance of the class at a time.

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        synchronized (this) {
            text = "";
            checkRep();
        }
    }
    public void insert(int pos, String ins) {
        synchronized (this) {
            text = text.substring(0, pos) + ins + text.substring(pos);
            checkRep();
        }
    }
    public void delete(int pos, int len) {
        synchronized (this) {
            text = text.substring(0, pos) + text.substring(pos+len);
            checkRep();
        }
    }
    public int length() {
        synchronized (this) {
            return text.length();
        }
    }
    public String toString() {
        synchronized (this) {
            return text;
```

所有对ADT的rep的访问都加锁

哪怕这些trivial的方法也要如此

Monitor模式：ADT所有方法都是互斥访问

为何如此简单的observer也需要lock？

Monitor pattern

- If add the keyword `synchronized` to a method signature, Java will act as if you wrote `synchronized(this)` around the method body.

Java actually forbids it, syntactically, because an object under construction is expected to be confined to a single thread until it returned from constructor. Synchronizing constructors should be unnecessary.

```
/**
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        text = "";
        checkRep();
    }
    public synchronized void insert(int pos, String ins) {
        text = text.substring(0, pos) + ins + text.substring(pos);
        checkRep();
    }
    public synchronized void delete(int pos, int len) {
        text = text.substring(0, pos) + text.substring(pos+len);
        checkRep();
    }
    public synchronized int length() {
        return text.length();
    }
    public synchronized String toString() {
        return text;
    }
}
```

Synchronized Statements/Block

- **What's the difference between a `synchronized` method and a `synchronized(this)` block?**
  - Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock.
  - Synchronized statements are useful for improving concurrency with fine-grained synchronization.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

同一时刻只能有一个线程访问该段代码

- **二者有何区别?**
  - 后者需要显式的给出lock，且不一定非要是this
  - 后者可提供更细粒度的并发控制

Sprinkling `synchronized` everywhere?

- **So is thread safety simply a matter of putting the synchronized keyword on every method in your program?**
- **Unfortunately not.**

- **First, you actually don't want to synchronize methods willy-nilly.**
  - Synchronization imposes a large cost on your program. 同步机制给性能带来极大影响
  - Making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors).
  - Java leaves many of its mutable datatypes unsynchronized by default exactly for these performance reasons. When you don't need synchronization, don't use it. 除非必要，否则不要用。Java中很多mutable的类型都不是threadsafe就是这个原因

Sprinkling synchronized everywhere?

- **Another argument for using synchronized in a more deliberate way is that it minimizes the scope of access to your lock. 尽可能减小lock的范围**
  - Adding synchronized to every method means that your lock is the object itself, and every client with a reference to your object automatically has a reference to your lock, that it can acquire and release at will.
  - Your thread safety mechanism is therefore public and can be interfered with by clients.

- **Contrast that with using a lock that is an object internal to your rep, and acquired appropriately and sparingly using `synchronized()` blocks.** 避免在方法spec中加synchronized，而是在方法代码内部更加精细的区分哪些代码行可能有threadsafe风险，为其加锁。

Sprinkling `synchronized` everywhere?

- **Finally, it's not actually sufficient to sprinkle `synchronized` everywhere.**
  - Dropping `synchronized` onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the right lock for guarding the shared data access you're about to do. 要先去思考清楚到底lock谁，然后再synchronized(…)
- **Suppose we had tried to solve `findReplace`'s synchronization problem simply by dropping synchronized onto its declaration:**

```
public static synchronized boolean findReplace(EditBuffer buf, ...)
```

  - It would indeed acquire a lock — because `findReplace` is a static method, it would acquire a static lock for the whole class that `findReplace` happens to be in, rather than an instance object lock. 意味着在class层面上锁！
  - As a result, only one thread could call `findReplace` at a time — even if other threads want to operate on different buffers, which should be safe, they'd still be blocked until the single lock was free. So we'd suffer a significant loss in performance. 对性能带来极大损耗！

Sprinkling `synchronized` everywhere?

- **The `synchronized` keyword is not a panacea.**
- **Thread safety requires a discipline — using confinement, immutability, or locks to protect shared data.**
- **And that discipline needs to be written down, or maintainers won't know what it is.**

- Synchronized不是灵丹妙药，你的程序需要严格遵守设计原则，先尝试其他办法，实在做不到再考虑lock。
- 所有关于threadsafe的设计决策也都要在ADT中记录下来。

Problem 1

- **If thread B tries to acquire a lock currently held by thread A**
- **What happens to thread A?**
  - blocks until B acquires the lock
  - blocks until B releases the lock
  - Nothing
- **What happens to thread B?**
  - blocks until A acquires the lock
  - blocks until A releases the lock
  - nothing

```
Thread A
public void run() {
    …
    synchronized(lock) {
    …
    }
}
```

```
Thread B
public void run() {
    …
    synchronized(lock) {
    …
    }
}
```

Problem 2

- **Suppose `list` is an instance of `ArrayList<String>`.**
- **What is true while `A` is in a `synchronized (list) { ... }` block?**
  - It owns the lock on `list`
  - It does not own the lock on `list`
  - No other thread can use observers of `list`
  - No other thread can use mutators of `list`
  - No other thread can acquire the lock on `list`
  - No other thread can acquire locks on elements in `list`

Thread A
```
public void run() {
    …
    synchronized(list) {
    …
    }
}
```

Thread B
```
public void run() {
    …
    lst.add(…);
    lst.size();
}
```

对同一个mutable对象的操作，必须要在各线程里用synchronized全部保护起来

Problem 3

- **Suppose `sharedList` is a `List` returned by `Collections.synchronizedList`.**
- **It is now safe to use `sharedList` from multiple threads without acquiring any locks… except!**
- **Which of the following would require a `synchronized(sharedList) { ... }` block?**
  - call `isEmpty`
  - call `add`
  - iterate over the `list`
  - call `isEmpty`, if it returns false, call `remove(0)`

Individual operations are safe to call without additional synchronization

You must synchronize on the list before iterating. This prevents other clients from mutating the list during the iteration.

In between call to `isEmpty` and `remove`, someone else could have emptied the list!

Thread safety argument with synchronization

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   true
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Safety from rep exposure:
    //   text is private and immutable
    // Thread safety argument:
    //   all accesses to text happen within SimpleBuffer methods,
    //   which are all guarded by SimpleBuffer's lock
```

Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument.

If `text` were `public`, then clients outside `SimpleBuffer` would be able to read and write it without knowing that they should first acquire the lock, and `SimpleBuffer` would no longer be threadsafe.

Locking discipline

- **A locking discipline is a strategy for ensuring that synchronized code is threadsafe.**
- **We must satisfy two conditions:**
  - Every shared mutable variable must be guarded by some lock. The data may not be read or written except inside a synchronized block that acquires that lock. **任何共享的mutable变量/对象必须被lock所保护**
  - If an invariant involves multiple shared mutable variables (which might even be in different objects), then all the variables involved must be guarded by the same lock. Once a thread acquires the lock, the invariant must be reestablished before releasing the lock. **涉及到多个mutable变量的时候，它们必须被同一个lock所保护**
- The monitor pattern as used here satisfies both rules. All the shared mutable data in the rep — which the rep invariant depends on — are guarded by the same lock. **monitor pattern中，ADT所有方法都被同一个synchronized(this)所保护**

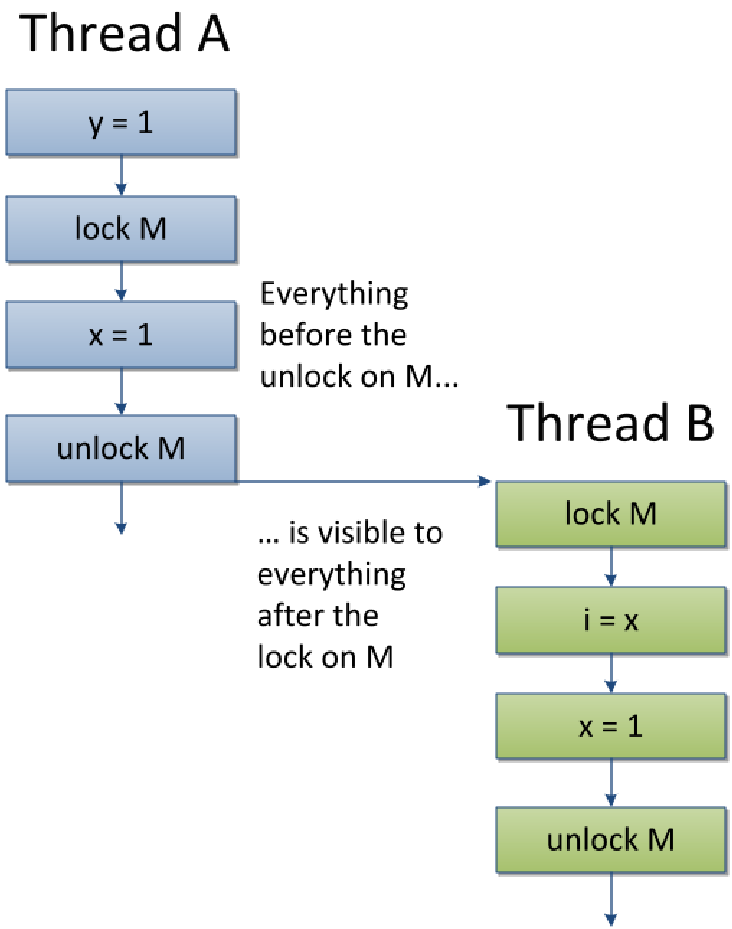
Locking discipline

```
class Poem {
      private String title;
      private List<String> lines = Collections.synchronizedList(…);

      public synchronized void operation(String filter) {
        Iterator<String> iter = lines.iterator();
        while (iter.hasNext()) {
           String line = iter.next();
           if (line.contains(filter))
                iter.remove();
        }
        title = title.toUpperCase();
      }
}
```

happens-before relationship

- **This happens-before relationship is simply a guarantee that the objects shared by multiple threads writes by one specific statement are visible to another specific statement which read the same object.**



- **This is to ensure Memory Consistency.**

- **https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

3 Atomic operations

Use synchronization to develop a threadsafe ADT

- Suppose we're building a **multi-user editor** that allows multiple people to connect to it and edit it at the same time.
- We'll need a **mutable datatype** to represent the text in document.
- Here's the interface ➔ basically it represents a **string** with **insert** and **delete** operations.

```java
/** An EditBuffer represents a threadsafe mutable
 *  string of characters in a text editor. */
public interface EditBuffer {
    /**
     * Modifies this by inserting a string.
     * @param pos position to insert at
                     (requires 0 <= pos <= current buffer length)
     * @param ins string to insert
     */
    public void insert(int pos, String ins);

    /**
     * Modifies this by deleting a substring
     * @param pos starting position of substring to delete
     *               (requires 0 <= pos <= current buffer length)
     * @param len length of substring to delete
     *               (requires 0 <= len <= current buffer length - pos)
     */
    public void delete(int pos, int len);

    /**
     * @return length of text sequence in this edit buffer
     */
    public int length();

    /**
     * @return content of this edit buffer
     */
    public String toString();
}
```

Three Reps for `EditBuffer`

- **A String**
  - `private String text;`

Every time we do an insert or delete, we have to copy the entire string into a new string. That gets expensive.

- **A character array,** with space at the end.
  - `private char[] text;`

If the user is typing at the beginning of the document, then we're copying the entire document with every keystroke.

- **A gap buffer**
  - A character array with extra space in it, but instead of having all space at the end, the extra space is a *gap* that can appear anywhere in the buffer.
  - Whenever an insert or delete operation is needed, the datatype first moves the gap to the location of the operation, and then does the insert or delete.
  - If the gap is already there, then nothing needs to be copied — an insert just consumes part of the gap, and a delete just enlarges the gap.
  - Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves.

Three Reps for `EditBuffer`

- **Gap buffer**

```
/** GapBuffer is a non-threadsafe EditBuffer that is optimized
 *  for editing with a cursor, which tends to make a sequence of
 *  inserts and deletes at the same place in the buffer. */
public class GapBuffer implements EditBuffer {
    private char[] a;
    private int gapStart;
    private int gapLength;
    // Rep invariant:
    //   a != null
    //   0 <= gapStart <= a.length
    //   0 <= gapLength <= a.length - gapStart
    // Abstraction function:
    //   represents the sequence a[0],...,a[gapStart-1],
    //                                a[gapStart+gapLength],...,a[length-1]
```

In a multiuser scenario, we'd want multiple gaps, one for each user's cursor

Atomic operations

- **Consider a find-and-replace operation on the `EditBuffer` datatype:**

```
/** Modifies buf by replacing the first occurrence of s with t.
 *  If s not found in buf, then has no effect.
 *  @returns true if and only if a replacement was made
 */
public static boolean findReplace(EditBuffer buf, String s, String t) {
    int i = buf.toString().indexOf(s);
    if (i == -1) {
        return false;
    }
    buf.delete(i, s.length());
    buf.insert(i, t);
    return true;
}
```

- **This method makes three different calls to `buf`.**
  - Even though each of these calls individually is atomic, the `findReplace` method as a whole is not threadsafe, because other threads might mutate the buffer while `findReplace` is working, causing it to delete the wrong region or put the replacement back in the wrong place.
- **To prevent this, `findReplace` needs to synchronize with all other clients of `buf`.**

Giving clients access to a lock

- **It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype.**
- **So one approach to the problem with `findReplace` is to document that clients can use the `EditBuffer`'s lock to synchronize with each other:**

```
/** An EditBuffer represents a threadsafe mutable string of characters
 *  in a text editor. Clients may synchronize with each other using the
 *  EditBuffer object itself. */
public interface EditBuffer {
   ...
}
```

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
    synchronized (buf) {
        int i = buf.toString().indexOf(s);
        if (i == -1) {
            return false;
        }
        buf.delete(i, s.length());
        buf.insert(i, t);
        return true;
    }
}
```

To ensure that all three methods are executed without interference from other threads.

To implement higher-level atomic operations

- **The effect of this is to enlarge the synchronization region that the monitor pattern already put around the individual `toString`, `delete`, and `insert` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads.**
- **这种方式让atomic的范围更大了，将多个atomic的操作组合为更大的atomic操作**

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
    synchronized (buf) {
        int i = buf.toString().indexOf(s);
        if (i == -1) {
            return false;
        }
        buf.delete(i, s.length());
        buf.insert(i, t);
        return true;
    }
}
```

4 Deadlock

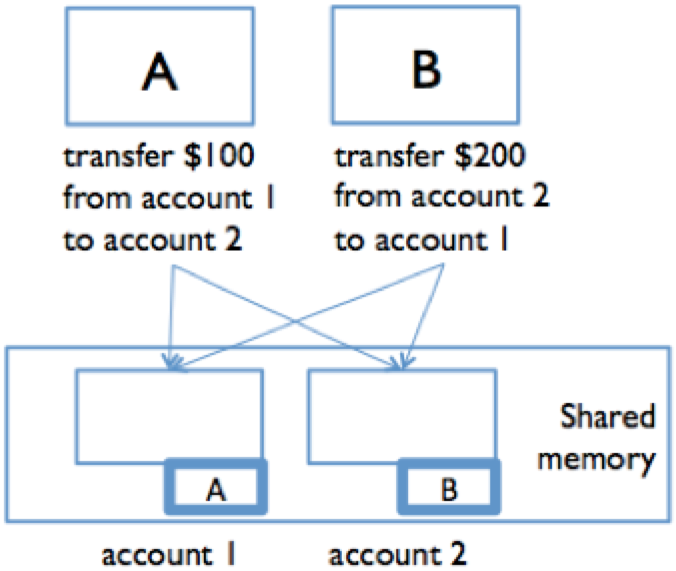
Deadlock

- **When used properly and carefully, locks can prevent race conditions.**
- **But then another problem rears its ugly head.**
- **Because the use of locks requires threads to wait (acquire blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting for each other — and hence neither can make progress.**

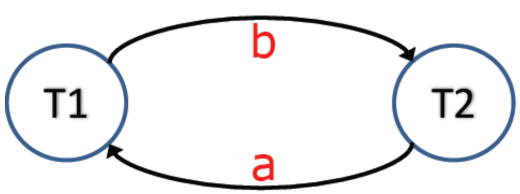
- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.
- **死锁：多个线程竞争lock，相互等待对方释放lock**

Deadlock

- **Deadlock** occurs when concurrent modules are stuck waiting for each other to do something.
- A deadlock may involve more than two modules: the signal feature of deadlock is a **cycle of dependencies** , e.g. A is waiting for B which is waiting for C which is waiting for A. None of them can make progress.




```
T1: synchronized(a){ synchronized(b){ … } }
T2: synchronized(b){ synchronized(a){ … } }
```



```
public class Wizard {
    private final String name;
    private final Set<Wizard> friends;
    // Rep invariant:
    //    name, friends != null
    //    friend links are bidirectional:
    //        for all f in friends, f.friends contains this
    // Concurrency argument:
    //    threadsafe by monitor pattern: all accesses to rep
    //    are guarded by this object's lock

    public Wizard(String name) {
        this.name = name;
        this.friends = new HashSet<Wizard>();
    }

    public synchronized boolean isFriendsWith(Wi
        return this.friends.contains(that);
    }

    public synchronized void friend(Wizard that) {
        if (friends.add(that)) {
            that.friend(this);
        }
    }

    public synchronized void defriend(Wizard that) {
        if (friends.remove(that)) {
            that.defriend(this);
        }
    }
}
```

```
Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");
```

```
// thread A                    // thread B
harry.friend(snape);           snape.friend(harry);
harry.defriend(snape);         snape.defriend(harry);
```

?

It modifies the reps of both objects, because they use the monitor pattern means acquiring the locks to both objects.

Deadlock rears its ugly head

- **Deadlock:**
  - Thread A acquires the lock on `harry` (because the friend method is synchronized).
  - Then thread B acquires the lock on `snape` (for the same reason).
  - They both update their individual reps independently, and then try to call `friend()` on the other object — which requires them to acquire the lock on the other object.
- **So A is holding `harry` and waiting for `snape`, and B is holding `snape` and waiting for `harry`.**
  - Both threads are stuck in `friend()`, so neither one will ever manage to exit the synchronized region and release the lock to the other.
  - This is a classic deadly embrace. The program simply stops.

The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free.

Deadlock solution 1: lock ordering

- **To put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order**.
  - In the example, we might always acquire the locks on the `Wizard` objects in alphabetical order by the wizard's name.

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

Deadlock solution 1: lock ordering

- Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice.
- First, it's **not modular** — the code has to know about all the locks in the system, or at least in its subsystem.
- Second, it may be difficult or **impossible** for the code to know exactly **which of those locks** it will need before it even acquires the first one. It may need to do some computation to figure it out.
  - Think about doing a depth-first search on the social network graph, for example — how would you know which nodes need to be locked, before you've even started looking for them?

Deadlock solution 2: coarse-grained locking

- **To use coarser locking — use a single lock to guard many object instances, or even a whole subsystem of a program.**
  - E.g., we might have a single lock for an entire social network, and have all the operations on any of its constituent parts synchronize on that lock.
  - In the code, all `Wizards` belong to a `Castle`, and we just use that `Castle` object's lock to synchronize.
- **However, it has a significant performance penalty.**
  - If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently.
  - In the worst case, having a single lock protecting everything, your program might be essentially sequential.

```
public class Wizard {
    private final Castle castle;
    private final String name;
    private final Set<Wizard> friends;
    ...
    public void friend(Wizard that) {
        synchronized (castle) {
            if (this.friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

Deadlock

```
Thread 1:

synchronized (alpha) {
  // using alpha
  // ...
}

synchronized (gamma) {
  synchronized (beta) {
   // using beta & gamma
   // ...
  }
}
// finished
```

```
Thread 2:

synchronized (gamma) {
  synchronized (alpha) {
    synchronized (beta) {
      // using alpha, beta, & gamma
      // ...
    }
  }
}
// finished
```

```
Thread 3:

synchronized (gamma) {
  synchronized (alpha) {
      // using alpha & gamma
      // ...
  }
}

synchronized (beta) {
  synchronized (gamma) {
      // using beta & gamma
      // ...
  }
}
// finished
```

Scenario A

Thread 1 inside using alpha
Thread 2 blocked on synchronized (alpha)
Thread 3 finished

Deadlock

```
Thread 1:

synchronized (alpha) {
  // using alpha
  // ...
}

synchronized (gamma) {
  synchronized (beta) {
   // using beta & gamma
   // ...
  }
}
// finished
```

```
Thread 2:

synchronized (gamma) {
  synchronized (alpha) {
    synchronized (beta) {
       // using alpha, beta, & gamma
       // ...
    }
  }
}
// finished
```

```
Thread 3:

synchronized (gamma) {
  synchronized (alpha) {
     // using alpha & gamma
     // ...
  }
}

synchronized (beta) {
  synchronized (gamma) {
     // using beta & gamma
     // ...
  }
}
// finished
```

Scenario B

```
Thread 1 finished
Thread 2 blocked on synchronized (beta)
Thread 3 blocked on 2nd synchronized (gamma)
```

Deadlock

```
Thread 1:

synchronized (alpha) {
  // using alpha
  // ...
}

synchronized (gamma) {
  synchronized (beta) {
   // using beta & gamma
   // ...
  }
}
// finished
```

```
Thread 2:

synchronized (gamma) {
  synchronized (alpha) {
    synchronized (beta) {
      // using alpha, beta, & gamma
      // ...
    }
  }
}
// finished
```

```
Thread 3:

synchronized (gamma) {
  synchronized (alpha) {
    // using alpha & gamma
    // ...
  }
}

synchronized (beta) {
  synchronized (gamma) {
    // using beta & gamma
    // ...
  }
}
// finished
```

Scenario C

```
Thread 1 running synchronized (beta)
Thread 2 blocked on synchronized (gamma)
Thread 3 blocked on 1st synchronized (gamma)
```

Deadlock

```
Thread 1:

synchronized (alpha) {
  // using alpha
  // ...
}

synchronized (gamma) {
  synchronized (beta) {
   // using beta & gamma
   // ...
  }
}
// finished
```

```
Thread 2:

synchronized (gamma) {
  synchronized (alpha) {
    synchronized (beta) {
      // using alpha, beta, & gamma
      // ...
    }
  }
}
// finished
```

```
Thread 3:

synchronized (gamma) {
  synchronized (alpha) {
     // using alpha & gamma
     // ...
  }
}

synchronized (beta) {
  synchronized (gamma) {
     // using beta & gamma
     // ...
  }
}
// finished
```

Scenario D

```
Thread 1 blocked on synchronized (beta)
Thread 2 finished
Thread 3 blocked on 2nd synchronized (gamma)
```

Deadlock

```
Thread 1:

synchronized (alpha) {
  // using alpha
  // ...
}

synchronized (gamma) {
  synchronized (beta) {
   // using beta & gamma
   // ...
  }
}
// finished
```

```
Thread 2:

synchronized (gamma) {
  synchronized (alpha) {
    synchronized (beta) {
       // using alpha, beta, & gamma
       // ...
    }
  }
}
// finished
```

```
Thread 3:

synchronized (gamma) {
  synchronized (alpha) {
     // using alpha & gamma
     // ...
  }
}

synchronized (beta) {
  synchronized (gamma) {
     // using beta & gamma
     // ...
  }
}
// finished
```

What about `alpha`?
Might it have deadlock with `gamma` and `beta`?

5 wait(), notify(), and notifyAll()

Guarded Blocks 保护块

- **Guarded block:** such a block begins by polling a condition that must be true before the block can proceed.
- **Suppose, for example `guardedJoy` is a method that must not proceed until a shared variable `joy` has been set by another thread.**
  - Such a method could simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting. 某些条件未得到满足，所以一直在空循环检测，直到条件被满足。这是极大浪费。

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {} //其他线程中更改joy之后，再往下执行
    System.out.println("Joy has been achieved!");
}
```

`wait()`, `notify()`, and `notifyAll()`

- **The following is defined for an arbitrary Java object `o`:**
  - `o.wait()`: release lock on `o`, enter `o`'s wait queue and wait
  - `o.notify()`: wake up one thread in `o`'s wait queue
  - `o.notifyAll()`: wake up all threads in `o`'s wait queue

Object.wait()

- `Object.wait()` causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- In other words, this method behaves exactly as if it simply performs the call `wait(0)`. 该操作使object所处的当前线程进入阻塞/等待状态，直到其他线程调用该对象的notify()操作

```
synchronized (obj) {

    while (<condition does not hold>)
       obj.wait();

    ... // Perform action appropriate to condition
}
```

Object.notify() /notifyAll()

- `Object.notify()` **wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.** 随机选择一个在该对象上调用wait方法的线程，解除其阻塞状态
  - A thread waits on an object's monitor by calling one of the wait methods.
  - The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object.
  - The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

`Object.notify() /notifyAll()`

- **This method should only be called by a thread that is the owner of this object's monitor.**
- **A thread becomes the owner of the object's monitor in one of three ways:**
  - By executing a `synchronized` instance method of that object.
  - By executing the body of a `synchronized` statement that synchronizes on the object.
  - For objects of type `Class`, by executing a `synchronized` static method of that class.

Using `wait()` in Guarded Blocks

- **The invocation of `wait()` does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for.**
- **The `Object.wait()` causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.**

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event,
    // which may not be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Using `wait()` in Guarded Blocks

- **When `wait()` is invoked, the thread releases the lock and suspends execution.**
- **At some future time, another thread will acquire the same lock and invoke `Object.notifyAll()`, informing all threads waiting on that lock that something important has happened:**

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

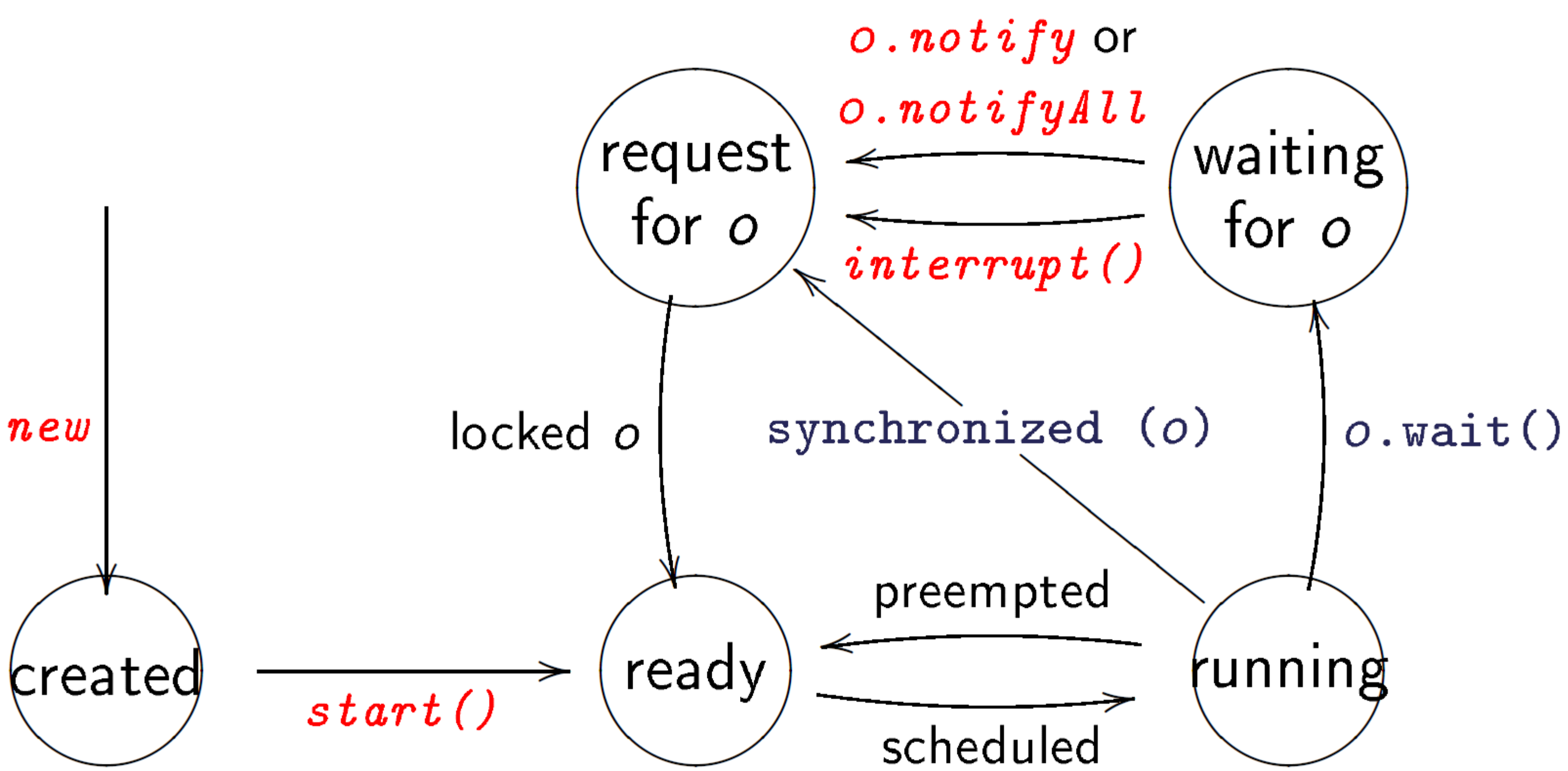
- **Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of `wait`.**
- **A complete example of `wait()` and `notifyAll()` can be found in http://www.tutorialspoint.com/java/lang/object_wait.htm**

`wait()`, `notify()`, and `notifyAll()`

- **A thread that calls methods on object `o` must have locked `o` beforehand, typically:**

```
synchronized (o) {
        ...
        o.wait () ;
        ...
}
```

State model for threads: locking and waiting



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

6 Summary

Goals of concurrent program design

- **Is a concurrent program *safe from bugs*?**
- **We care about three properties:**
  - **Safety**. Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Safety asks the question: Can you prove that some bad thing never happens ?
  - **Liveness**. Does the program keep running and eventually do what you want, or does it get stuck somewhere waiting forever for events that will never happen? Can you prove that some good thing eventually happens ? Deadlocks threaten liveness.
  - **Fairness**. Concurrent modules are given processing capacity to make progress on their computations. Fairness is mostly a matter for an OS' thread scheduler, but you can influence it by setting thread priorities.

Safety failure:
Incorrect computation

Liveness failure:
No computation at all

Concurrency in practice

- **What strategies are typically followed in real programs?**
  - **Library data structures either use no synchronization** (to offer high performance to single-threaded clients, while leaving it to multithreaded clients to add locking on top) or the **monitor pattern**.
  - **Mutable data structures with many parts typically use either coarse-grained locking or thread confinement.** Most graphical user interface toolkits follow one of these approaches, because a graphical user interface is basically a big mutable tree of mutable objects. Java Swing, the graphical user interface toolkit, uses thread confinement. Only a single dedicated thread is allowed to access Swing's tree. Other threads have to pass messages to that dedicated thread in order to access the tree.

Safety failures offer a false sense of security.
Liveness failures force you to confront the bug.
Temptation to favor liveness over safety.

Concurrency in practice

- **What strategies are typically followed in real programs?**
  - **Search often uses immutable datatypes.** It would be easy to make multithreaded, because all the datatypes involved were immutable. There would be no risk of either races or deadlocks.
  - **Operating systems often use fine-grained locks** in order to get high performance, and use lock ordering to deal with deadlock problems.
  - **Databases avoid race conditions using *transactions*** , which are similar to synchronized regions in that their effects are atomic, but they don't have to acquire locks, though a transaction may fail and be rolled back if it turns out that a race occurred. Databases can also manage locks, and handle locking order automatically.

    to be introduced in Database Systems course.

Summary

- Producing a concurrent program that is safe from bugs, easy to understand, and ready for change requires careful thinking.
  - Heisenbugs will skitter away as soon as you try to pin them down, so debugging simply isn’t an effective way to achieve correct threadsafe code.
  - Threads can interleave their operations in so many different ways that you will never be able to test even a small fraction of all possible executions.
- Make thread safety arguments about your datatypes, and document them in the code.

Summary

- Acquiring a lock allows a thread to have exclusive access to the data guarded by that lock, forcing other threads to block — as long as those threads are also trying to acquire that same lock.
- The *monitor pattern* guards the rep of a datatype with a single lock that is acquired by every method.
- Blocking caused by acquiring multiple locks creates the possibility of deadlock.

The end

May 17, 2020