Section 7: Thread Safety, issues and guidelines

Michelle Kuttel

mkuttel@cs.uct.ac.za

Thread safety

Writing thread-safe code is about managing an object's **state:**

- we need to protect **data** from concurrent access

worried about **shared, mutable** state

- **shared:** accessed by multiple threads
- **mutable**: value can change

Java frameworks that create threads

There are a number of Java frameworks that create threads and call your components from these threads, e.g:

- AWT and Swing create threads for managing user interface events
- Timer create threads for executing deferred tasks
- Component frameworks, such as **servlets** and **RMI**, create pools of threads an invoke component methods in these threads

This means that , if you use these frameworks, you **need** to ensure that your components are **thread-safe**

e.g. Timer class

- Timer is a convenience mechanism for scheduling tasks to run at a later time, either once or periodically
- TimerTasks are executed in a Thread managed by the Timer, not the application
- If TimerTask accesses data that is also accessed by other application threads, then not only must the TimerTask do so in a thread safe manner, but so must any other classes that access that data
  - easiest is to ensure that all objects accessed by TimerTask are themselves thread safe

What is a thread-safe class?

A class can be considered to be thread-safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment and with no additional synchronization of other coordination on the part of the calling code.

- no set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state.

Possible data races

Whenever:

- **more than one thread** accesses a given state variable
- **all accesses** must be coordinated using synchronization

Done in Java using `synchronized` keyword, or volatile variables, explicit locks, atomic variables

Checkpoint

- For safety, is it enough just declare every method of every shared object as *synchronized*?

Checkpoint contd.

Vector has every method synchronized.

- Is the following code atomic?

```
if (!vector.contains(element))
    vector.add(element);
```

The Java Monitor Pattern

- An object following this pattern encapsulates all its mutable stare and guards it with the object’s own intrinsic lock
- Used by many library classes:
  - Vector
  - HashTable
- Advantage is that it is simple

Concurrent Building Blocks in Java

- Synchronized collections:
  - e.g. Vector, Hashtable
  - achieve thread safety by serializing all access to collection's state
  - poor concurrency
    Only process one request at a time
    - All methods are locally sequential
    - Accept new messages only when ready
      - No other thread holds lock
      - Not engaged in another activity
    - But methods may make self-calls to other methods during same activity without blocking (due to reentrancy
  - may need additional locking to guard compound actions
    - iteration, navigation etc.

Types of race condition

The (poor) term "race condition" can refer to two *different* things resulting from lack of synchronization:

1. Data races: Simultaneous read/write or write/write of the same memory location
   - (for mortals) **always** an error, due to compiler & HW
2. Bad interleavings: Despite lack of data races, exposing bad intermediate state
   - "Bad" depends on your specification

slide adapted from:
Sophomoric Parallelism & Concurrency, Lecture 5

Guarding state with locks

- if synchronization is used to coordinate access to a variable, it is needed everywhere that variable is accessed.
- Furthermore, the **same lock**, must be used wherever the variable is accessed.
- the variable is then **guarded** by that lock
  - e.g. Vector class

Guarding state with locks

- Acquiring the lock associated with an object does NOT prevent other classes from accessing the object
  - it only prevents them from acquiring the same lock

Compound actions

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

Data race

Last lectures showed an example of an unsafe *read-modify-write* **compound action**, where resulting state is derived from the previous state

Another example is a *check-then-act* **compound action**

check-then-act

Code to find the maximum in a series of numbers. Each thread checks part of the series…

```
if (a[i] > cur_max)
        cur_max = a[i];
```

Data race

check-then-act: Lazy Initialization

This code is NOT thread-safe

```
@NotThreadSafe
public class LazyInitRace {

  private expensiveObject instance = null;
  public ExpensiveObject getInstance() {
   if (instance==null)
      instance = new ExpensiveObject();
   return instance;
}

}
```

Bad interleaving

Compound actions

read-modify-write and ***check-then-act*** are examples of **compound actions** that must be executed **atomically** in order to remain thread-safe.

Example

```
class Stack<E> {
  … // state used by isEmpty, push, pop
  synchronized boolean isEmpty() { … }
  synchronized void push(E val) { … }
  synchronized E pop() {
    if(isEmpty())
      throw new StackEmptyException();
    …
  }
  E peek() { // this is wrong
      E ans = pop();
      push(ans);
      return ans;
  }
}
```

peek, sequentially speaking

- In a sequential world, this code is of questionable *style*, but unquestionably *correct*
- The "algorithm" is the only way to write a **peek** helper method if all you had was this interface:

```
interface Stack<E> {
  boolean isEmpty();
  void push(E val);
  E pop();
}

class C {
  static <E> E myPeek(Stack<E> s){ ??? }
}
```

`peek`, concurrently speaking

- **`peek`** has no *overall* effect on the shared data
  – It is a "reader" not a "writer"
- But the way it's implemented creates an inconsistent *intermediate state*
  – Even though calls to **`push`** and **`pop`** are synchronized so there are no *data races* on the underlying array/list/whatever
- This intermediate state should not be exposed
  – Leads to several *bad interleavings*

peek and isEmpty

- Property we want: If there has been a **push** and no **pop**, then **isEmpty** returns **false**
- With **peek** as written, property can be violated – how?

Thread 1 (peek)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
boolean b = isEmpty()
```

Time ↓

peek and isEmpty

- Property we want: If there has been a **push** and no **pop**, then **isEmpty** returns **false**
- With **peek** as written, property can be violated – how?

Thread 1 (peek)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
boolean b = isEmpty()
```

Time

peek and push

- Property we want: Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?

Time ↓

Thread 1 (`peek`)

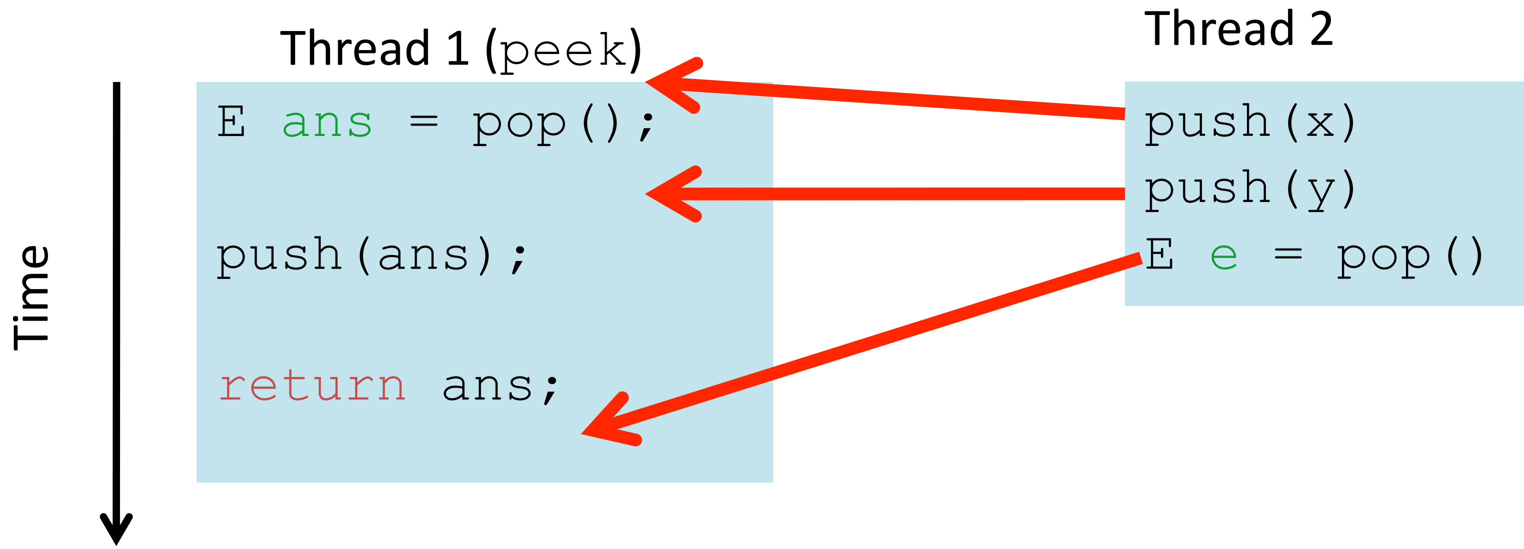
```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
push(y)
E e = pop()
```

peek and push

- Property we want: Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?



peek and pop

- Property we want: Values are returned from **`pop`** in LIFO order
- With **`peek`** as written, property can be violated – how?

Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)
push(y)
E e = pop()
```

Time

peek and peek

- Property we want: **peek** doesn't throw an exception if number of pushes exceeds number of pops
- With **peek** as written, property can be violated – how?

Time ↓

Thread 1 (peek)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
E ans = pop();

push(ans);

return ans;
```

peek and peek

- Property we want: **`peek`** doesn't throw an exception if number of pushes exceeds number of pops
- With **`peek`** as written, property can be violated – how?

Thread 1 (`peek`)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
E ans = pop();

push(ans);

return ans;
```

Time

The fix

- In short, **`peek`** is a compound action: needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
  - Re-entrant locks allow calls to **`push`** and **`pop`**

```
class Stack<E> {
  …
  synchronized E peek(){
     E ans = pop();
     push(ans);
     return ans;
  }
}
```

```
class C {
  <E> E myPeek(Stack<E> s){
    synchronized (s) {
      E ans = s.pop();
      s.push(ans);
      return ans;
    }
  }
}
```

The wrong "fix"

- Focus so far: problems from **`peek`** doing writes that lead to an incorrect intermediate state

- Tempting but wrong: If an implementation of **`peek`** (or **`isEmpty`**) does not write anything, then maybe we can skip the synchronization?

- Does not work due to *data races* with **`push`** and **`pop`**…

Example, again (no resizing or checking)

```
class Stack<E> {
  private E[] array = (E[])new Object[SIZE];
  int index = -1;
  boolean isEmpty() { // unsynchronized: wrong?!
    return index==-1;
  }
  synchronized void push(E val) {
    array[++index] = val;
  }
  synchronized E pop() {
    return array[index--];
  }
  E peek() { // unsynchronized: wrong!
    return array[index];
  }
}
```

Why wrong?

- It *looks like* **`isEmpty`** and **`peek`** can "get away with this" since **`push`** and **`pop`** adjust the state "in one tiny step"

- But this code is still *wrong* and depends on language-implementation details you cannot assume
  - Even "tiny steps" may require multiple steps in the implementation: **`array[++index] = val`** probably takes at least two steps
  - Code has a data race, allowing very strange behavior

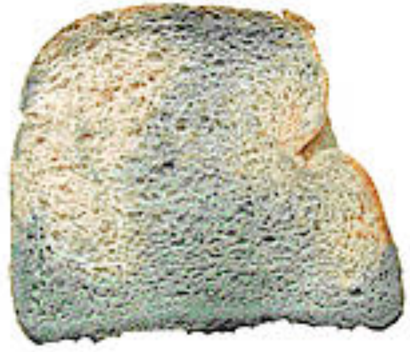
- Moral: Don't introduce a data race, even if every interleaving you can think of is correct

Sharing Objects: Visibility

- Synchronization is not only about atomicity
  - It is NOT TRUE that you only need synchronization when writing to variables.
- it is also about **memory visibility**:
  - when a thread modifies an object, we need to ensure that other threads can see the changes that were made.
  - without synchronization, this may not happen…

    …ever

Visibility and Stale data



Unless synchronization is used **every time** a shared variable is accessed, it is possible to see a stale value for that variable

Worse, staleness in **not** all-or-nothing:

some variable may be up-to-date, while others are stale

even more complicated if the stale data is an object reference, such as in a linked list

But it is easy to fix:

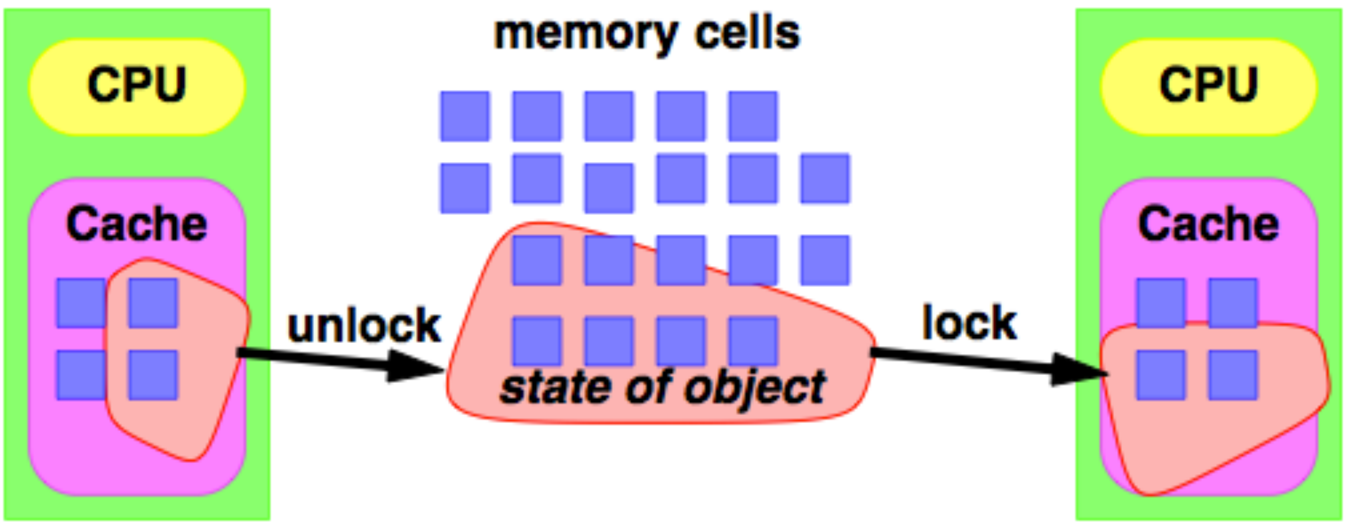
- Synchronized also has the side-effect of clearing locally cached values and forcing reloads from main storage

- so, synchronize all the getters and setters of shared values…on the SAME lock

Locks and Caching

Locking generates messages between threads and memory

- Lock acquisition forces reads from memory to thread cache
- Lock release forces writes of cached updates to memory



Locks and Caching

Without locking, there are NO promises about if and when caches will be flushed or reloaded

- Can lead to unsafe execution
- Can lead to nonsensical execution

Volatile Variables

`volatile` keyword controls per-variable flush/reload

When a field is declared volatile, they are not cached where they are hidden from other processes

- a read of a volatile variable **always** returns the most recent write by any thread.

Implementation:

- No locking, so **lighter weight** mechanism than `synchronized`.
  - no locking, so accessing variable cannot cause another thread to block
- slower than regular fields, faster than locks

But limited utility: fragile and code more opaque.

Really for experts: avoid them; use standard libraries instead

Volatile Variables

most common use of `volatile` is for a flag variable:

```
volatile boolean asleep;
while (!asleep)
  countSomeSheep();
```

While locking can guarantee **both** visibility and atomicity, volatile variables **can only guarantee visibility-**

NB volatile does **NOT** mean atomic!!!

And then we get reordering problems…

- The things that can go wrong are so counterintuitive…

Motivating memory-model issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {
  private int x = 0;
  private int y = 0;

  void f() {
    x = 1;
    y = 1;
  }
  void g() {
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

First understand why it looks like the assertion can't fail:

- Easy case: call to `g` ends before any call to `f` starts
- Easy case: at least one call to `f` completes before call to `g` starts
- If calls to `f` and `g` *interleave*…

Interleavings

There is no interleaving of **`f`** and **`g`** where the assertion fails

- Proof #1: Exhaustively consider all possible orderings of access to shared memory (there are 6)
- Proof #2: If **`!(b>=a)`**, then **`a==1`** and **`b==0`**. But if **`a==1`**, then **`a=y`** happened after **`y=1`**. And since programs execute in order, **`b=x`** happened after **`a=y`** and **`x=1`** happened before **`y=1`**. So by transitivity, **`b==1`**. Contradiction.

Thread 1: `f`

```
x = 1;

y = 1;
```

Thread 2: `g`

```
int a = y;

int b = x;

assert(b >= a);
```

Wrong

However, the code has a *data race*

- Two actually
- Recall: data race: unsynchronized read/write or write/write of same location

If code has data races, you cannot reason about it with interleavings!

- That's just the rules of Java (and C, C++, C#, …)
- (Else would slow down all programs just to "help" programs with data races, and that's not a good engineering trade-off)
- So the assertion can fail

Recall Guideline #0: No data races

How is this possible? -Reordering

There is no guarantee that operations in one thread will be performed in the order given in the program, as long as the reordering is not detectable from within *that* thread

…even if reordering is apparent to other threads!

Why

For performance reasons, the compiler and the hardware often reorder memory operations

- Take a compiler or computer architecture course to learn why

Thread 1: `f`

```
x = 1;

y = 1;
```

Thread 2: `g`

```
int a = y;

int b = x;

assert(b >= a);
```

Of course, you cannot just let them reorder anything they want

- Each thread executes in order after all!
- Consider: `x=17; y=x;`

The grand compromise

The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program

The compiler/hardware will never perform a memory reordering that affects the result of a data-race-free multi-threaded program

So: If no interleaving of your program has a data race, then you can *forget about all this reordering nonsense:* the result will be equivalent to some interleaving

Your job: Avoid data races

Compiler/hardware job: Give interleaving (illusion) *if you do your job*

Fixing our example

- Naturally, we can use synchronization to avoid data races
  - Then, indeed, the assertion cannot fail

```
class C {
  private int x = 0;
  private int y = 0;
  void f() {
    synchronized(this) { x = 1; }
    synchronized(this) { y = 1; }
  }
  void g() {
    int a, b;
    synchronized(this) { a = y; }
    synchronized(this) { b = x; }
    assert(b >= a);
  }
}
```

Code that's wrong

- Here is a more realistic example of code that is wrong
  - No *guarantee* Thread 2 will *ever* stop
  - But honestly it will "likely work in practice"

```
class C {
  boolean stop = false;
  void f() {
    while(!stop) {
      // draw a monster
    }
  }
  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: `f()`

Thread 2: `g()`

Checkpoint

What are all the possible outputs of this code?

```
public class possibleReordering {

    static int x=0, y=0;
    static int a=0, b=0;

    public static void main (String[] args) throws
  InterruptedException {
        Thread one = new Thread( new Runnable() {
            public void run() {
                a=1;
                x=b;
            }
        });
        Thread two = new Thread( new Runnable() {
            public void run() {
                b=1;
                y=a;
            }
        });
        one.start(); two.start();
        one.join(); two.join();
        System.out.println("(" + x + "," + y + ")");
    }
}
```

Outputs

(1,0)

(0,1)

(1,1)

(0,0)

!!!

Aside: Java Memory model

- Java has rules for which values may be seen by a read of shared memory that is updated by multiple threads.
- As the specification is similar to the *memory models* for different hardware architectures, these semantics are known as the *Java programming language memory model*.

Aside: Java Memory model

Java memory model requires maintenance of *within thread as-if-serial semantics*.

- each thread must has same result as if executed serial

The JVM defines a partial ordering called *happens-before* on all actions in a program

To guarantee that an action B sees the results of action A, there must be a *happens-before* relationship between them

If there isn't one, Java is free to reorder the actions

Aside: Java Memory model

The Java Memory Model is specified in ‘happens before’-rules, e.g.:

- **monitor lock rule**: a release of a lock happens before every subsequent acquire of the same lock.

Aside: Java Memory model

The Java Memory Model is specified in 'happens before'-rules, e.g.:

- **volatile variable rule**: a write of a volatile variable **happens before** every subsequent read of the same volatile variable

Non-atomic 64-bit operations

out-of-thin-air safety is a guarantee that, when a thread reads a variable without synchronization, it may see a stale value, but it **will be** a value that **was actually written at some point**

- **i.e. not a random value**

Non-atomic 64-bit operations

out-of-thin-air safety guarantee applies to all variables that are not declared `volatile`, **except** for 64-bit numeric variables

- the JVM can read or write these in 2 separate 32-bit operations
- shared mutable **`double`** **and** **`long`** values MUST be declared volatile or guarded by a lock

Policies and guidelines for thread safety

Guarding state with locks

- It is up to you to construct locking protocols or synchronization policies that let you access shared state safely

- Every **shared mutable** variable should be accessed by exactly one lock.
  - make it clear to maintainers which lock it is

- But mutable, unshared variables do not need to be locked
- And neither do immutable, shared variables

Most useful policies for using and sharing objects in a concurrent Java program

Thread-confined (thread-local)

- object owned exclusively by and confined to one thread
- can be modified by owning thread

Shared read-only (immutable)

- can be accessed by multiple threads without synchronization
- no modifications

Shared thread-safe

- performs synchronization internally, so can be freely accessed by multiple threads

Synchronized (Guarded)

- accessed only when a lock is held

Thread-local

Whenever possible, don't share resources: called **thread confinement**

- Easier to have each thread have its own thread-local *copy* of a resource than to have one with shared updates
- This is correct only if threads don't need to communicate through the resource
  - That is, multiple copies are a correct approach
  - Example: **`Random`** objects
- Note: Since each call-stack is thread-local, never need to synchronize on local variables
- if data is accessed only from a single thread, no synchronization is needed
  - its **usage** is automatically thread-safe, even if the object itself is not

In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory should be rare – minimize it

slide adapted from: Sophomoric Parallelism & Concurrency, Lecture 5

Thread Confinement

Swing uses thread confinement extensively:

- Swing visual components and data model objects are not thread safe
- safety achieved by confining them to the Swing event dispatch thread
  - NB code running in other threads should not access these objects
    - many concurrency errors in Swing applications are a result of this

Immutable

Immutable Objects are **always thread safe**

- they only have **one state**

Use of final guarantees initialization safety

So, make all fields final unless they need to be mutable.

Whenever possible, **don't update objects**

  – Make new objects instead

- One of the key tenets of *functional programming*
  – Generally helpful to avoid *side-effects*
  – Much more helpful in a concurrent setting

- If a location is only read, never written, then no synchronization is necessary!
  – Simultaneous reads are *not* races and *not* a problem

In practice, programmers usually over-use mutation – minimize it

slide adapted from: Sophomoric Parallelism & Concurrency, Lecture 5

The rest

After minimizing the amount of memory that is (1) thread-shared and (2) mutable, we need **guidelines** for how to use locks to keep other data consistent

Guideline #0: No data races

- Never allow two threads to read/write or write/write the same location at the same time

Necessary: In Java or C, a program with a data race is almost always wrong

Not sufficient: Our **`peek`** example had no data races

Consistent Locking

Guideline #1: For each location needing synchronization, have a lock that is always held when reading or writing the location

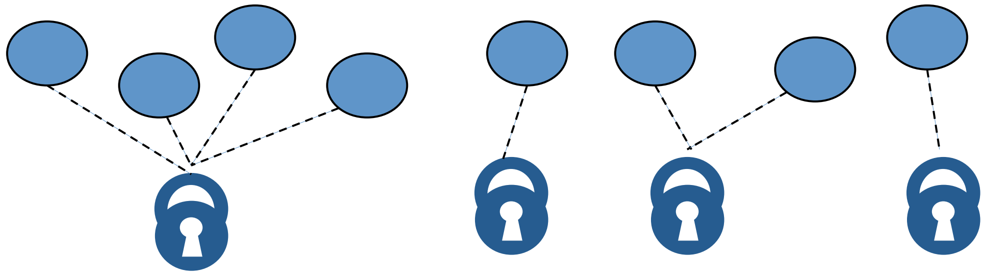
- We say the lock guards the location
- The same lock can (and often should) guard multiple locations
- **Clearly document the guard** for each location

- In Java, often the guard is the object containing the location
  - `this` inside the object's methods
  - But also often guard a larger structure with one lock to ensure mutual exclusion on the structure

this slide adapted from: Sophomoric Parallelism & Concurrency, Lecture 5

Consistent Locking continued

- The mapping from locations to guarding locks is *conceptual*
- It partitions the shared-&-mutable locations into "which lock"



Consistent locking is:

- *Not sufficient*: It prevents all data races but still allows bad interleavings
  - Our `peek` example used consistent locking
- *Not necessary*: Can change the locking protocol dynamically…

Consistent locking is an excellent guideline: "default assumption" about program design

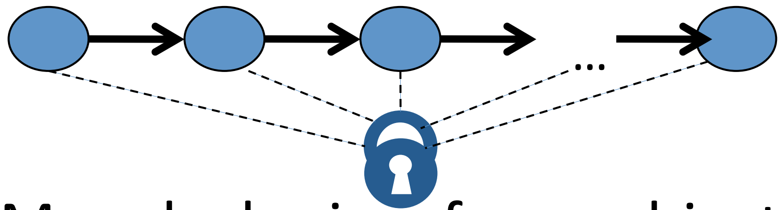
Locking caveats

- Whenever you use locking, you should be aware of what the code in the block is doing and how likely it is to take a long time to execute
- Holding a lock for a long time introduces the risk of liveness and performance problems
  - avoid holding locks during lengthy computations or during network of console I/O

Lock granularity

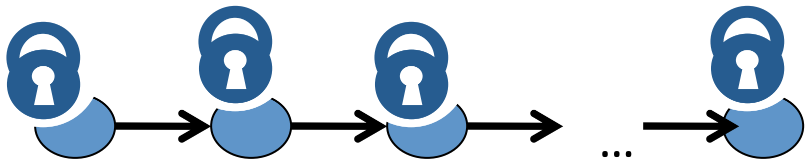
Coarse-grained: Fewer locks, i.e., more objects per lock

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts



Fine-grained: More locks, i.e., fewer objects per lock

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account



"Coarse-grained vs. fine-grained" is really a continuum

Trade-offs

Coarse-grained advantages

- Simpler to implement
- Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Much easier: operations that modify data-structure shape

Fine-grained advantages

- More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)

Guideline #2: Start with coarse-grained (simpler) and move to fine-grained (performance) only if *contention* on the coarser locks becomes an issue. Alas, often leads to bugs.

Example: Hashtable

- Coarse-grained: One lock for entire hashtable
- Fine-grained: One lock for each bucket

Which supports more concurrency for **`insert`** and **`lookup`**?

Which makes implementing **`resize`** easier?

  – How would you do it?

If a hashtable has a **`numElements`** field, maintaining it will destroy the benefits of using separate locks for each bucket

Critical-section granularity

A second, orthogonal granularity issue is critical-section size

- How much work to do while holding lock(s)

If critical sections run for too long:

- Performance loss because other threads are blocked

If critical sections are too short:

- Bugs because you broke up something where other threads should not be able to see intermediate state

Guideline #3: Don't do expensive computations or I/O in critical sections, but also don't introduce race conditions

Example

Suppose we want to change the value for a key in a hashtable without removing it from the table

– Assume **`lock`** guards the whole table

Papa Bear's critical section was too long

(table locked during expensive call)

```
synchronized(lock) {
  v1 = table.lookup(k);
  v2 = expensive(v1);
  table.remove(k);
  table.insert(k,v2);
}
```

Example

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume **`lock`** guards the whole table

Mama Bear's critical section was too short

(if another thread updated the entry, we will lose an update)

```
synchronized(lock) {
  v1 = table.lookup(k);
}
v2 = expensive(v1);
synchronized(lock) {
  table.remove(k);
  table.insert(k,v2);
}
```

Example

Suppose we want to change the value for a key in a hashtable without removing it from the table

– Assume **lock** guards the whole table

Baby Bear's critical section was just right

(if another update occurred, try our update again)

```
done = false;
while(!done) {
  synchronized(lock) {
    v1 = table.lookup(k);
  }
  v2 = expensive(v1);
  synchronized(lock) {
    if(table.lookup(k)==v1) {
      done = true;
      table.remove(k);
      table.insert(k,v2);
}}}
```

Atomicity

An operation is *atomic* if no other thread can see it partly executed

- Atomic as in "(appears) indivisible"
- Typically want ADT operations atomic, even to other threads running operations on the same ADT

Guideline #4: Think in terms of what operations need to be *atomic*

- Make critical sections just long enough to preserve atomicity
- *Then* design the locking protocol to implement the critical sections correctly

That is: Think about atomicity first and locks second

Don't roll your own

- It is rare that you should write your own data structure
  - Provided in standard libraries
  - Point of these lectures is to understand the key trade-offs and abstractions
- Especially true for concurrent data structures
  - Far too difficult to provide fine-grained synchronization without race conditions
  - Standard thread-safe libraries like **`ConcurrentHashMap`** written by world experts

Guideline #5: Use built-in libraries whenever they meet your needs

Concurrent Building Blocks in Java

- Synchronized collections achieve thread safety by serializing all access to the collection's state
  - poor concurrency, because of collection-wide lock
- **Concurrent collections** are designed for concurrent access from multiple threads:
    - ConcurrentHashMap
    - CopyOnWriteArrayList
- Replacing synchronized collections with concurrent collections can result in dramatic scalability improvement with little risk