Thread-unsafe code

- How can the following class be broken by multiple threads?

```
 1 public class Counter {
 2   private int c = 0;
 3   public void increment() {
 4     int old = c;
 5     c = old + 1;  // c++;
 6   }
 7   public void decrement() {
 8     int old = c;
 9     c = old - 1;  // c--;
10   }
11   public int value() {
12     return c;
13   }
14 }
```

Scenario that breaks it:

- Threads A and B start.
- A calls `increment` and runs to the end of line 4. It retrieves the `old` value of 0.
- B calls `decrement` and runs to the end of line 8. It retrieves the `old` value of 0.
- A sets c to its `old` (0) + 1.
- B sets c to its `old` (0) - 1.
- The final `value()` is -1, though after one increment and one decrement, it should be 0!

Synchronized blocks

```
// synchronized block:
// uses the given object as a lock
synchronized (object) {
    statement(s);
}
```

- Every Java object can act as a "lock" for concurrency.
  - A thread T_1 can ask to run a block of code, "synchronized" on a given object *O*.
    - ✓ If no other thread is using *O*, then T_1 locks the object and proceeds.
    - ✓ If another thread T_2 is already using *O*, then T_1 becomes blocked and must wait until T_1 is finished using O. Then T_1 can proceed.

Synchronized methods

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) { ... }

// synchronized static method: locks on the given class
public static synchronized type name(parameters) { ... }
```

- A synchronized method grabs the object or class's lock at the start, runs to completion, then releases the lock.
  - A shorthand for wrapping the entire body of the method in a `synchronized (this) {...}` block.
  - Useful for methods whose entire bodies should not be entered by multiple threads at the same time.

```
public synchronized void readFile(String name) {...}
```

Synchronized counter

```
public class Counter {
    private int c = 0;

    public synchronized void increment() {
        int old = c;
        c = old + 1;  // c++;
    }

    public synchronized void decrement() {
        int old = c;
        c = old - 1;  // c--;
    }

    public int value() {
        return c;
    }
}
```

- Should the `value` method be synchronized? Why/why not?

Races

A race condition occurs when the computation result depends on scheduling (how threads are interleaved)

Bugs that exist only due to concurrency

- No interleaved scheduling with 1 thread

Typically, problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

5

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;
  }
}
```

6

`peek`, sequentially speaking

- In a sequential world, this code is of questionable *style*, but unquestionably *correct*

7

`peek`, concurrently speaking

- `peek` has no *overall* effect on the shared data
  - It is a "reader" not a "writer"
- But the way it is 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
  - (A data race is simultaneous (unsynchronized) read/write or write/write of the same memory: more on this soon)
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

8

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;
```

Time

Thread 2

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

9

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;
```

Time

Thread 2

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

10

peek and push

- 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;
```

Time

Thread 2

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

11

peek and push

- 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;
```

Time

Thread 2

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

12

peek and pop

- 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()
```

13

peek and peek

- Property we want: **peek** does not 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;
```

14

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;
```

15

The fix

- In short, **peek** 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;
    }
  }
}
```

16

Example, again

```
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];
  }
}
```

17

Why wrong?

- **`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: Do not introduce a data race, even if every interleaving you can think of is correct

18

The distinction

The 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
   - **always** an error, due to compiler & HW
   - Original **`peek`** example has no data races
2. Bad interleavings: Despite lack of data races, exposing bad intermediate state
   - "Bad" depends on your specification
   - Original **`peek`** example had several

19

Getting it right

Avoiding race conditions on shared resources is difficult

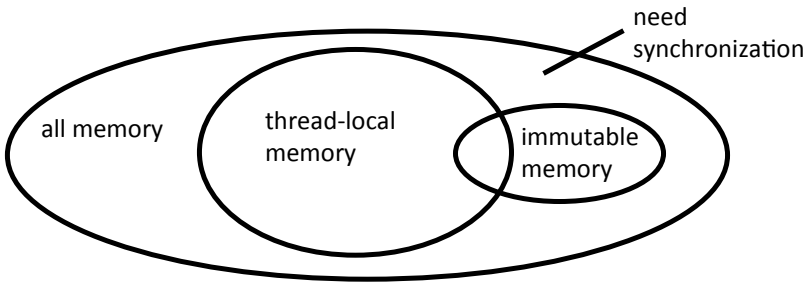
- Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work

20

3 choices

For every memory location (e.g., object field) in your program, you must obey at least one of the following:

1. Thread-local: Do not use the location in > 1 thread
2. Immutable: Do not write to the memory location
3. Synchronized: Use synchronization to control access to the location



21

Thread-local

Whenever possible, do not share resources

- 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 do not need to communicate through the resource
  - ✓ That is, multiple copies are a correct approach
  - ✓ Example: **`Random`** objects
- Note: Because each call-stack is thread-local, never need to synchronize on local variables

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

22

Immutable

Whenever possible, do not 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

23

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

24

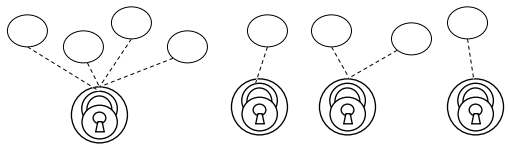
Consistent Locking

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

25

Consistent Locking continued



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…

26

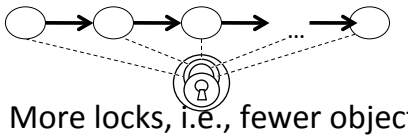
Beyond consistent locking

- Consistent locking is an excellent guideline
  - A "default assumption" about program design
- But it isn't required for correctness: Can have different program phases use different invariants
  - Provided all threads coordinate moving to the next phase

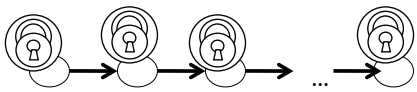
27

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

28

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.

29

Example: Hashtable

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

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

Which makes implementing **`resize`** easier?
- How would you do it?

Maintaining a **`numElements`** field for the table will destroy the benefits of using separate locks for each bucket
- Why?

30

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

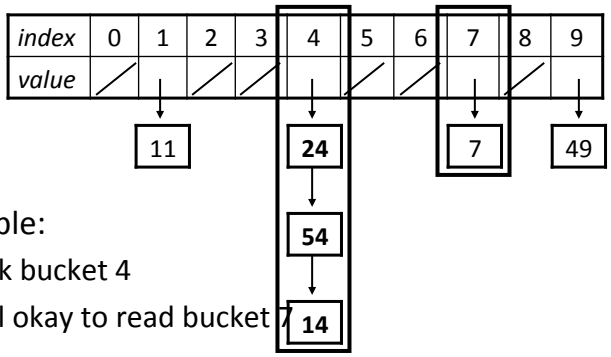
If critical sections are too short:
- Bugs because you broke up something where other threads should not be able to see intermediate state

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

31

Fine-grained critical sections

- Technically, the shared resource is the linked list in the given hash bucket `elements[h]`, not the *entire* hash table array.



- Example:
  - ✓lock bucket 4
  - ✓still okay to read bucket 7

Fine-grained locking

Time

```
// keep a lock for each
   bucket

// Thread 1:  add(42);
public void add(E value) {
  int h = hash(value);  //
  2
  Node n = new Node(value);
  ...
  ...
  ...
  ...
  synchronized (locks[h]) {
    n.next = elements[h];
    elements[h] = n;
    size++;
  }
}
```

```
  ...
  ...
  ...
  ...
  ...
// Thread 2:  add(72);
public void add(E value) {
  int h = hash(value);  // 2
  Node n = new Node(value);
  ...
  synchronized (locks[h]) {
    ... blocked ...
    ... blocked ...
    ... blocked ...
    n.next = elements[h];
    elements[h] = n;
    size++;
  }
}
```

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

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);
}
```

34

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

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);
}
```

35

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

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);
}}}
```

36

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

37

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

38

Synchronized collections

- Java provides thread-safe collection *wrappers* via static methods in the `Collections` class:

Method
`Collections.synchronizedCollection(`**coll**`)`
`Collections.synchronizedList(`**list**`)`
`Collections.synchronizedMap(`**map**`)`
`Collections.synchronizedSet(`**set**`)`

```
Set<String> words = new HashSet<String>();
words = Collections.synchronizedSet(words);
```

  - These are essentially the same as wrapping each operation on the collection in a `synchronized` block.
    - ✓Simpler, but not more efficient, than the preceding code.

Concurrent collections

- New package `java.util.concurrent` contains collections that are optimized to be safe for use by multiple threads:
  - `class` **`ConcurrentHashMap`**`<K, V> implements Map<K, V>`
  - `class` **`ConcurrentLinkedDeque`**`<E> implements Deque<E>`
  - `class` **`ConcurrentSkipListSet`**`<E> implements Set<E>`
  - `class` **`CopyOnWriteArrayList`**`<E> implements List<E>`
- These classes are generally faster than using a synchronized version of the normal collections because multiple threads are actually able to use them at the same time, to a degree.
  - hash map: one thread in each hash bucket at a time
  - deque: one thread modifying each end of the deque (front/back)
  - ...

Object lock methods

- *Every* Java object has a built-in internal "lock".
  - A thread can "wait" on an object's lock, causing it to pause.
  - Another thread can "notify" on an object's lock, unpausing any other thread(s) that are currently waiting on that lock.
  - An implementation of *monitors*, a classic concurrency construct.

method	description
`notify`()	unblocks one random thread waiting on this object's lock
`notifyAll`()	unblocks all threads waiting on this object's lock
`wait`() **`wait`**(**ms**)	causes the current thread to wait (block) on this object's lock, indefinitely or for a given # of ms

  - These methods are not often used directly; but they are used internally by other concurrency constructs

Wait

- Check condition in a synchronized block
  - If true, continue execution
  - If false, call wait()

```
synchronized(lockObject)
{ while( ! condition ){ lockObject.wait();}
  action;
}
```

- Properties of wait()
  - releases the implicit lock (of the synchr. block)!
  - threads having called wait() can't be scheduled!

Notify

- Activate threads in wait set
  - Single: notify()
  - All: notifyAll()

```
synchronized(lockObject) {
  establish_the_condition;
  lockObject.notifyAll()
}
```

43

Comments

- Implementation of wait set performs queuing
- The while() loop ensures that the condition is checked prior to the action
- Waiting threads consume no (ok few) resources
  - wait() is blocking call
- Can be interrupted by an exception
  - wait()/notify() is the simplest programming interface to conditions

44

The volatile keyword

```
private volatile type name;
```

- **volatile field**: An indication to the VM that multiple threads may try to access/update the field's value at the same time.
  - Causes Java to immediately flush any internal caches any time the field's value changes, so that later threads that try to read the value will always see the new value (never the stale old value).
  - Allows limited safe concurrent access to a field inside an object even if another thread may modify the field's value.
  - Does not solve all concurrency issues; should be replaced by `synchronized` blocks if more complex access is needed.

Deadlock

- **liveness**: Ability for a multithreaded program to run promptly.
- **deadlock**: Situation where two or more threads are blocked forever, waiting for each other.
  - Example: Each is waiting for the other's locked resource.
  - Example: Each has too large of a `synchronized` block.
- **livelock**: Situation where two or more threads are caught in an infinite cycle of responding to each other.
- **starvation**: Situation where one or more threads are unable to make progress because of another "greedy" thread.
  - Example: thread with a long-running synchronized method

New classes for locking

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

Class/interface	description
`Lock`	an interface for controlling access to a shared resource
`ReentrantLock`	a class that implements `Lock`
`ReadWriteLock`	like `Lock` but separates read operations from writes
`Condition`	a particular shared resource that can be waited upon; conditions are acquired by asking for one from a `Lock`

- These classes offer higher granularity and control than the `synchronized` keyword can provide.
  - Not needed by most programs.
  - `java.util.concurrent` also contains blocking data structures.