CSE 332: Data Abstractions

Lecture 21: Programming with Locks and Critical Sections

Ruth Anderson
Spring 2013
Announcements

• No class on Monday May 27
• Homework 7 – due Friday May 31

• **Project 3** – the last programming project!
  – Version 1 & 2 - **Tues May 28, 2013 11PM**
  – ALL Code - Tues June 4, 2013 11PM
  – Experiments & Writeup - Thurs June 6, 2013, 11PM
Outline

Done:
- The semantics of locks
- Locks in Java
- Using locks for mutual exclusion: bank-account example

This lecture:
- More bad interleavings (learn to spot these!)
- Guidelines/idioms for shared-memory and using locks correctly
- Coarse-grained vs. fine-grained

Next lecture:
- Readers/writer locks
- Deadlock
- Condition variables
- Data races and memory-consistency models
Race Conditions

A race condition occurs when the computation result depends on scheduling (how threads are interleaved)
- If T1 and T2 happened to get scheduled in a certain way, things go wrong
- We, as programmers, cannot control scheduling of threads;
- Thus we need to write programs that work independent of scheduling

Race conditions are bugs that exist only due to concurrency
- No interleaved scheduling problems with only 1 thread!

Typically, problem is that some intermediate state can be seen by another thread; screws up other thread
- Consider a ‘partial’ insert in a linked list; say, a new node has been added to the end, but ‘back’ and ‘count’ haven’t been updated
Race Conditions:
Data Races vs. Bad Interleavings

We will make a big distinction between *data races* and *bad interleavings*, both kinds of race-condition bugs

– Confusion often results from not distinguishing these or using the ambiguous “race condition” to mean only one
Data Races (briefly)

- A data race is a specific type of race condition that can happen in 2 ways:
  - Two different threads potentially write a variable at the same time
  - One thread potentially writes a variable while another reads the variable
- Not a race: simultaneous reads provide no errors
- ‘Potentially’ is important
  - We claim the code itself has a data race independent of any particular actual execution
- Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present…through bad interleavings (our focus for this lecture).
Stack Example (pseudocode)

class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    synchronized boolean isEmpty() {
        return index===-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
Example of a Race Condition, but not a Data Race

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:

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

class C {
    static <E> E myPeek(Stack<E> s) { ??? }
}
```
Problems with `peek`:

- `peek` has no *overall* effect on the shared data
  - It is a “reader” not a “writer”
  - State should be the same after it executes as before

- 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
    - Can’t access ‘top’ simultaneously
  - There is still a *race condition* though

- This intermediate state should not be exposed
  - Leads to several *bad interleavings*
Example 1: peek and isEmpty

- **Property we want**: If there has been a push (and no pop), then `isEmpty` should return `false`.

- With `peek` as written, property can be violated – how?

```java
Thread 1 (peek)
E ans = pop();
push(ans);
return ans;

Thread 2
push(x)
boolean b = isEmpty()
```
Example 1: peek and isEmpty

- **Property we want:** If there has been a push (and no pop), then isEmpty should return false.

- With peek as written, property can be violated – how?

```java
E ans = pop();
push(ans);
return ans;
```

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

Thread 1 (peek)          Thread 2

5/24/2013
Example 1: peek and isEmpty

- **Property we want**: If there has been a push (and no pop), then `isEmpty` should return `false`.

- With `peek` as written, property can be violated – how?

  ```java
  E ans = pop();
  push(ans);
  return ans;
  ```

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

  It can be violated if things occur in this order:
  1. T2: push(x)
  2. T1: pop()
  3. T2: boolean b = isEmpty()
Example 2: peek and push

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

```java
Thread 1 (peek)
E ans = pop();
push(ans);
return ans;

Thread 2
push(x)
push(y)
E e = pop()
```
Example 2: peek and push

- **Property we want:** Values are returned from `pop` in LIFO order

- With `peek` as written, property can be violated – how?

```java
E ans = pop();
push(ans);
return ans;
```

Thread 1 (peek)       Thread 2
E ans = pop();
push(x)
push(y)
E e = pop()
Example 3: peek and pop

- **Property we want**: Values are returned from `pop` in LIFO order

- With `peek` as written, property can be violated – how?

```cpp
E ans = pop();
push(ans);
return ans;
```

Thread 1 (peek)

Thread 2

push(x)
push(y)
E e = pop()
Example 4: peek and peek

• **Property we want:** `peek` doesn’t throw an exception unless stack is empty

• With `peek` as written, property can be violated – how?

  Thread 1 (**peek**)
  ```java
  E ans = pop();
  push(ans);
  return ans;
  ```

  Thread 2 (**peek**)
  ```java
  E ans = pop();
  push(ans);
  return ans;
  ```

5/24/2013
Example 4: peek and peek

- **Property we want:** `peek` doesn’t throw an exception unless stack is empty

- With `peek` as written, property can be violated – how?

```java
Thread 1 (peek)
E ans = pop();
push(ans);
return ans;

Thread 2 (peek)
E ans = pop();
push(ans);
return ans;
```
The fix

- In short, `peek` needs synchronization to disallow interleavings
  - The key is to make a `larger critical section`
- That intermediate state of `peek` needs to be protected
- Use re-entrant locks; will allow calls to `push` and `pop`
- Code on right is a `peek` external to the `Stack` class

```java
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, (pseudocode not complete)

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 `is-empty` 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
    • Compiler optimizations may break it in ways you had not anticipated
    • We’ll talk about this more in the future

• Moral: Do not introduce a data race, even if every interleaving you can think of is correct
The distinction

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 (next lecture)
   - 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* had several
Getting it right

Avoiding race conditions on shared resources is difficult
- What ‘seems fine’ in a sequential world can get you into trouble when multiple threads are involved
- Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work

Rest of lecture distills key ideas and trade-offs
- Parts paraphrased from “Java Concurrency in Practice”
  - Chapter 2 (rest of book more advanced)
- But none of this is specific to Java or a particular book!
- May be hard to appreciate in beginning, but come back to these guidelines over the years – don’t try to be fancy!
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
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
Immutable

Whenever possible, do not update objects
  – Make new objects instead!

• One of the key tenets of functional programming (see CSE 341)
  – 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
The rest: Keep it synchronized

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* (use locks!)
  - Even if it ‘seems safe’

**Necessary:**
- a Java or C program with a data race is almost always wrong
- Even if our reasoning tells us otherwise; ex: compiler optimizations

**But Not sufficient:** Our **peek** example had no data races, and it’s still wrong…
**Consistent Locking**

**Guideline #1:** Use consistent locking

- 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 (ex. multiple fields in a class)

- 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
Consistent Locking (continued)

- The mapping from locations to guarding locks is *conceptual*
  - Must be enforced by you as the programmer
- It partitions the shared-and-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, but still had exposed intermediate states (and allowed potential bad interleavings)
- *Not necessary*: Can change the locking protocol dynamically…
Beyond consistent locking

• Consistent locking is an excellent guideline
  – A “default assumption” about program design
  – You will save yourself many a headache using this guideline

• But it isn’t required for correctness: Can have different program phases use different invariants
  – Provided all threads coordinate moving to the next phase

• Example from Project 3, Version 5:
  – A shared grid being updated, so use a lock for each entry
  – But after the grid is filled out, all threads except 1 terminate
    • So synchronization no longer necessary (thread local)
  – And later the grid becomes immutable
    • So synchronization is doubly unnecessary
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 for operations that modify data-structure shape

**Fine-grained advantages:**
- More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
- Can make multi-node operations more difficult: say, rotations in an AVL tree

**Guideline #2:** *Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue.*
Example: Separate Chaining 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, why?
Example: Separate Chaining Hashtable

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

Which supports more concurrency for insert and lookup?
  Fine-grained; allows simultaneous access to diff. buckets

Which makes implementing resize easier?
  - How would you do it?
  - Coarse-grained; just grab one lock and proceed

If a hashtable has a numElements field, maintaining it will destroy the benefits of using separate locks for each bucket, why?
Updating it each insert w/o a lock would be a data race
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?
   –

If critical sections are too short?
   –
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; keep it as small as possible but still be correct
Example 1: Critical-section granularity

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
- **expensive()** takes in the old value, and computes a new one, but takes a long time

```java
synchronized(lock) {
    v1 = table.lookup(k);
    v2 = expensive(v1);
    table.remove(k);
    table.insert(k, v2);
}
```
Example 1: Critical-section granularity

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
- `expensive()` takes in the old value, and computes a new one, but takes a long time

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

Papa Bear’s critical section was too long

(table locked during expensive call)
Example 2: Critical-section granularity

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

```java
synchronized(lock) {
    v1 = table.lookup(k);
}

v2 = expensive(v1);
synchronized(lock) {
    table.remove(k);
    table.insert(k, v2);
}
```
Example 2: Critical-section granularity

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

```java
synchronized (lock) {
    v1 = table.lookup(k);
}

v2 = expensive(v1);

synchronized (lock) {
    table.remove(k);
    table.insert(k, v2);
}
```

Mama Bear’s critical section was too short
(if another thread updated the entry, we will lose an update)
Example 3: Critical-section granularity

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

```java
done = false;
while (!done) {
    synchronized (lock) {
        v1 = table.lookup(k);
    }
    v2 = expensive(v1);
    synchronized (lock) {
        if (table.lookup(k) == v1) {
            done = true; // I can exit the loop!
            table.remove(k);
            table.insert(k, v2);
        }
    }
}
```
Example 3: Critical-section granularity

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

```java
done = false;
while (!done) {
    synchronized (lock) {
        v1 = table.lookup(k);
    }
    v2 = expensive(v1);
    synchronized (lock) {
        if (table.lookup(k) == v1) {
            done = true; // I can exit the loop!
            table.remove(k);
            table.insert(k, v2);
        }
    }
}
```

Baby Bear’s critical section was just right

(if another update occurred, try our update again)
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

• In “real life”, it is unusual to have to write your own data structure from scratch
  – Implementations provided in standard libraries
  – Point of CSE332 is to understand the key trade-offs, abstractions, and analysis of such implementations

• 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