Concurrency: where are we

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

Upcoming lectures:
- Readers/writer locks
- Deadlock
- Condition variables
- Data races and memory-consistency models

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

Example

class Stack<E> {
    ...
    synchronized boolean isEmpty() { ... }
    synchronized void push(E val) { ... }
    synchronized E pop(E val) {
        if(isEmpty())
            throw new StackEmptyException();
        ...
    }
    E peek() {
        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(E val);
}

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 wrong 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?

```java
E ans = pop();
push(ans);
boolean b = isEmpty();
return ans;
```

**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?

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

```
Thread 2
push(x)  
push(y)  
E e = pop();
```

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

```
Thread 2
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;
```
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?

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

```java
class Stack<E> {
    ...  
    synchronized E peek() {
        E ans = pop();
        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)

```java
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(E val) {
        return array[index--];
    }
    // unsynchronized: wrong!
    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
    - Important discussion in future lecture

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

Getting it right

Avoiding race conditions on shared resources is difficult
- Decades of bugs has 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!

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: Don’t use the location in > 1 thread
2. Immutable: Don’t write to the memory location
3. Synchronized: Use synchronization to control access to the location

Thread-local

Whenever possible, don’t 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 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

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

Whenever possible, don’t update objects  
– Make new objects instead  

• One of the key tenets of *functional programming* (see CSE341)  
  – 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**

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

---

**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 higher-level races (exposed intermediate states)  
  – Our `peek` example used consistent locking  

• *Not necessary:* Can change the locking protocol dynamically…
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
- 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
    - Makes synchronization 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: 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

```java
synchronized(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

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;
                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

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.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

• It is rare that you should write your own data structure
  – Provided in standard libraries
  – Point of CSE332 is to understand the key trade-offs and
  abstractions
• Especially true for concurrent data structures
  – Far too difficult to provide fine-grained synchronization
without data races
  – Standard thread-safe libraries like ConcurrentHashMap
written by world experts

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