Announcements

- Homework 7 – due Friday March 4th at the BEGINNING of lecture!
- Project 3 – the last programming project!
  - Version 1 & 2 - Tues March 1, 2011 11PM - (10% of overall grade)
  - ALL Code - Tues March 8, 2011 11PM - (65% of overall grade)
  - Writeup - Thursday March 10, 2011, 11PM - (25% of overall grade)

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

Races

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; result is that we need to write programs that work independent of scheduling

Race conditions are bugs that exist only due to concurrency
- No interleaved scheduling with 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

Data Races

- A data race is a specific type of race condition that can happen in 2 ways:
  - Two different threads can potentially write a variable at the same time
  - One thread can potentially write a variable while another reads the variable
  - Simultaneous reads are fine; not a data race, and nothing bad would happen
  - Potentially is important; we say the code itself has a data race – it is independent of an actual execution
- Data races are bad, but we can still have a race condition, and bad behavior, when no data races are present

Stack Example

```java
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

```java
class Stack<E> {
    --
    synchronized boolean isEmpty() { -- }
    synchronized void push(E val) { -- }
    synchronized E pop(E val) {
        if(isEmpty())
            throw new StackEmptyException();
        --
    } // peek()
    E ans = pop();
    push(ans);
    return ans;
}
```

- Maybe we’re writing `peek` in an external class that only has access to Stack’s `push` and `pop`
- In a sequential world, this code is of questionable style, but correct

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();
}
```

```java
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’s implemented creates an inconsistent intermediate state
  - 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; errors can occur

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?

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1 (peek)</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E ans = pop();</td>
<td>push(x)</td>
</tr>
<tr>
<td></td>
<td>push(ans);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return ans;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>boolean b = isEmpty()</td>
<td>push(ans);</td>
</tr>
</tbody>
</table>

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?

```
Thread 1 (peek)
E ans = pop();
push(ans);
return ans;
```
```
Thread 2
push(x)
push(y)
E o = 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?

```
Thread 1 (peek)
E ans = pop();
push(x);
push(y)
E o = pop()
```
```
Thread 2
push(ans);
return ans;
```

Example 2: peek and pop (again)

- 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 o = pop()
```

Example 3: 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)
E ans = pop();
push(ans);
return ans;
```
```
Thread 2 (peek)
E ans = pop();
push(ans);
return ans;
```

Example 3: 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)
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

```
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();
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`…**

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, which may result in strange behavior
    - Compiler optimizations may break it in ways you had not anticipated
    - We’ll talk about this more in the future
- 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
- What ‘seems fine’ in a sequential world can get you into trouble when race conditions are involved
- 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!

Pick one of these 3 choices for memory:

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 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: In Java or C, a 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

Consistent Locking (continued)
• The mapping from locations to guarding locks is conceptual, and is something that you have to enforce as a programmer
• 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, but 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 locking techniques
  – 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 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`?
- Fine-grained; allows simultaneous access to different buckets

Which makes implementing `resize` easier?
- 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

---

**Example: 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

**Papa Bear’s critical section was too long**

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

---

**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: 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: 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

• It is rare that you should write your own data structure
  – Provided in standard libraries
  – Point of CSE 332 is to understand the key trade-offs, abstractions and analysis

• 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