Lecture 26: Using Threads Safely

CS 62
Spring 2011
Kim Bruce & Kevin Coogan

Some slides based on those from Dan Grossman, U. of Washington.

Providing Safe Access

• For every memory location (e.g., object field) in your program, you must obey at least one of the following:
  • Thread-local: Don’t access the location in > 1 thread
  • Immutable: Don’t write to the memory location
  • 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 key tenets of functional programming
  - Hopefully you study this in 52
  - Generally helpful to avoid side-effects
  - Much more helpful in a concurrent setting
- If a location is only read, never written, no synchronization is necessary!
  - Simultaneous reads are not races and not a problem
- *Programmers over-use mutation – minimize it.*

Dealing with the Rest

- Guideline: 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

Worse Than You Think!

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

- Assertion always true w/ single threaded.
- Looks always true for multithreaded.
  - OK if f not called at all
  - OK after f completes
  - Looks OK if in middle of f
- But have race condition

Memory Reordering

- For performance reasons, compiler and hardware reorder memory operations.
- But, but, ...
  - 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 need not worry: result will be equivalent to some interleaving
### A Second Fix

- If label field `volatile`, accesses don't count as data races
- Implementation forces memory consistency
  - though slower!
- Should have used this in CS 51 w/shared variables.
- Really for experts -- better to use locks.

### 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: ops that modify data-structure shape
- **Fine-grained advantages**
  - More simultaneous access (performance when coarse-grained would lead to unnecessary blocking)
- **Guideline**:
  - 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.

### 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**: Don’t do expensive computations or I/O in critical sections, but also don’t introduce race conditions
Example: ArrayList

• Granularity:
  • One lock for entire list or
  • One lock per slot

• Critical Section size
  • Suppose get access to element, do something expensive to see if needs an update and then update
    • If too large, then all other accesses blocked
    • If too small, then element in slot may change while check.

Don’t Roll Your Own!

• Most data structures provided in standard libraries
  • Point of 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: Use built-in libraries whenever they meet your needs Vector vs ArrayList

Deadlock

```java
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt, BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

• What locks are held at a.deposit(amt)?
• Is this a problem?
Deadlock

• Suppose have separate threads, each transferring to each others’ account

Thread 1: x.transferTo(1, y)
  acquire lock for x
  do withdraw from x
  block on lock for y

Thread 2: y.transferTo(1, x)
  acquire lock for y
  do withdraw from y
  block on lock for x

Deadlock

• A deadlock occurs when there are threads T₁, ..., Tₙ such that:
  • For i=1,..,n-1, Tᵢ is waiting for a resource held by Tᵢ₊₁
  • Tₙ is waiting for a resource held by T₁
  • In other words, there is a cycle of waiting
  • Formalize as a graph of dependencies with cycles bad
  • Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

A Last Example

• Bounded buffer is a queue with a fixed size.
  • Like event queue
  • Implemented in an array where wraps around.
• Producer threads do work and enqueue result
• Consumer threads dequeue results and perform work on them.
• Must synchronize access to the queue.

Attempt 1

class Buffer<E> {
  E[] array = (E[])new Object[SIZE];
  ... // front, back fields, isEmpty, isFull methods
  synchronized void enqueue(E elt) {
    if(isFull()) {
      ???
    } else {
      ... add to array and adjust back ...
    }
  }
  synchronized E dequeue() {
    if(isEmpty()) {
      ???
    } else {
      ... take from array and adjust front ...
    }
  }
}
Waiting

- enqueue to full buffer should not raise exception
  - Wait until there is room
- dequeue from empty buffer should not raise exception
  - Wait until there is data
- Bad approach is “spin lock”

What we want ...

- Thread should wait until has needed resources
  - Release lock and wait to be notified
- Needs operating systems support
- “Condition variable” that informs waiters when conditions have changed.
- See BoundedBuffer.java
  - uses “this” as condition variable

Once Again: Use Existing Classes!

- Java libraries contain thread-safe data structures.
  - See java.util.concurrent.BlockingQueue<E> interface
    - ArrayBlockingQueue
    - LinkedBlockingQueue
  - ConcurrentHashMap
  - Vector