Programming with Threads

Cormac Flanagan

Sequential Program Execution

void m() {

m()

Multithreaded Program Execution

• Decompose program into pieces that can run in parallel
• Advantages
  – exploit multiple processors
  – threads make progress, even if others block
  – more responsive

Multithreaded Program Execution

Race Conditions

class Ref {
    int i;
    void add(int r) {
        i = i + r;
    }
}
Race Conditions

```java
class Ref {
    int i;
    void add(Ref r) {
        i = i + r.i;
    }
}
Ref x = new Ref(0);
Ref y = new Ref(3);
x.add(y);
x.add(y);
assert x.i == 6;
```

A race condition occurs if
- two threads access a shared variable at the same time
- at least one of those accesses is a write

Lock-Based Synchronization

```java
class Ref {
    int i;                // guarded by this
    void add(Ref r) {
        i = i + r.i;
    }
}
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    synchronized (x) {
        synchronized (y) {
            x.add(y); // two calls happen
            x.add(y); // in parallel
        }
        assert x.i == 6;
    }
}
```

- Lock can be held by at most one thread at a time
- Java objects are locks
- Each field guarded by a lock
- Correct lock acquired before accessing field
- Ensures race freedom
Problems with Locking: Lock Contention

- Shared Hashtable
  - locking discipline?

Problems with Locking: Lock Contention

- Shared Hashtable
  - locking discipline: 1 lock for whole table
  - only one thread can operate on hashtable at a time

Avoiding Lock Contention

- Lock at finer granularity
  - 1 lock per row
  - multiple thread can operate on hashtable at a time
  - more complex, trade-off

Fine Grain Locking in Linux

- Linux originally had (roughly) a single, global lock protecting all kernel data structures
  - poor performance on multiprocessors
- Linux 2.4 introduced finer-grain locking
  - hard to get right
  - Linux 2.4 log has 36 synchronization bug fixes

Problems with Locking: Deadlock

- Suppose
  - Thread A locks object M1
  - Thread B locks object M2
  - Thread A blocks trying to lock M2
  - Thread B blocks trying to lock M1
- Solution
  - Use partial order
    - Decide that M1 < M2
    - If you want to hold M1 and M2, you must lock M1 before M2

Problems with Locking: Priority Inversion

- Suppose
  - Thread C (low priority) locks object M and keeps on running
  - Thread B (medium priority) pre-empts C and keeps on running
  - Thread A (high priority) pre-empts B then tries to lock M
- A won’t make progress until B blocks and C releases lock
Implementing a Buffer (version 1)

```java
class Buffer {
    Object elem; // null if empty

    void put(Object o) {
        while (elem != null) {}  
        elem = o;
    }

    Object get() {
        while (elem == null) {}  
        Object r = elem;
        elem = null;
        return r;
    }
}
```

Implementing a Buffer (version 2)

```java
class Buffer {
    Object elem; // null if empty

    void put(Object o) {
        synchronized (this) {
            while (elem != null) {}  
            elem = o;
        }
    }

    Object get() {
        synchronized (this) {
            while (elem == null) {}  
            Object r = elem;
            elem = null;
            return r;
        }
    }
}
```

Implementing a Buffer (version 3)

```java
class Buffer {
    Object elem; // null if empty

    void put(Object o) {
        synchronized (this) {
            while (elem != null) { this.wait(); }  
            elem = o;
            this.notify();
        }
    }

    Object get() {
        synchronized (this) {
            while (elem == null) { this.wait(); }  
            Object r = elem;
            elem = null;
            this.notify();
            return r;
        }
    }
}
```

Implementing a Buffer (version 4)

```java
class Buffer {
    Object elem; // null if empty

    void put(Object o) {
        synchronized (this) {
            while (elem != null) { this.wait(); }  
            elem = o;
            this.notifyAll();
        }
    }

    Object get() {
        synchronized (this) {
            while (elem == null) { this.wait(); }  
            Object r = elem;
            elem = null;
            this.notifyAll();
            return r;
        }
    }
}
```

- Thread creation
- Locking (contention, deadlock, priority inversion)
- Wait & notify

wait() releases the lock; sleeps until notified; reacquires the lock; and returns
notify() notifies one waiting thread
notifyAll() notifies all waiting threads
**Memory Models**

- If program is race-free
  - then behaves intuitively
  - as if on a sequentially-consistent memory model
- If program has race conditions
  - then you may get weird, unintuitive behaviors
  - don’t do this
    - or at least read the memory model

**Software Validation & Verification**

- Standard approach: Testing
- Multithreaded software
  - increasing widespread (Java, C#, GUIs, servers)
  - testing inadequate due to
    - test coverage
    - scheduling coverage
- An important application for static analysis!
  - race freedom
  - atomicity

**Problems With Locking**

- Only works if you remember to do it
  - inadvertent programming errors cause races
- Race conditions are difficult to
  - detect
  - reproduce
  - eliminate
Therac – 25

- Linear accelerator to deliver x-rays and electron beams for the treatment of cancer
- Third-generation machine
  - replaced expensive hardware safety interlocks with software controls
- Race condition in software controls
  - six serious overdoses

Testing

- Sequential program testing
  - does program behave correctly for all inputs?
  - test cases uses sample inputs
- Multithreaded program testing
  - does program behave correctly for all inputs and all interleavings?
  - much harder!
- Type systems
  - great for *no such method*, etc
  - for race conditions?

Type System For Race Freedom

- Static type system prevents race conditions
- Programmer specifies synchronization discipline
  - lock protecting each field
  - locks held on entry to each method
- Type checker checks synchronization discipline
  - checks field accessed only when lock held
  - checks for all inputs and all interleavings
  - theorem: well-typed programs are race-free

Verifying Race Freedom with Types

```java
class Ref {  
    int i guarded_by this;  
    void add(Ref r) requires this, r {  
        i = i + r.i;  
    }  
}  
Ref x = new Ref(0);  
Ref y = new Ref(3);  
parallel {  
    synchronized (x,y) { x.add(y); }  
    synchronized (x,y) { x.add(y); }  
}  
assert x.i == 6;  
```

Verifying Race Freedom with Types

```java
class Ref {  
    int i guarded_by this;  
    void add(Ref r) requires this, r {  
        i = i + r.i;  
    }  
}  
Ref x = new Ref(0);  
Ref y = new Ref(3);  
parallel {  
    synchronized (x,y) { x.add(y); }  
    synchronized (x,y) { x.add(y); }  
}  
assert x.i == 6; 
```
Verifying Race Freedom with Types

class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
  synchronized (x,y) { x.add(y); }
  synchronized (x,y) { x.add(y); }
}
assert x.i == 6;

Verifying Race Freedom with Types

Soundness Theorem:
Well-typed programs are race-free

Escapes

- Some good programs have “benign races”
  - allow program to escape type system

```java
class Account {
  private int balance guarded_by this;
  public Account(int n) {
    balance = n; // no_warn
  }
}
```

- Java has casts to escape from the type system
  ```java
  Object o;
  String s = (String)o;
  ```

class Account {
  private int balance guarded_by this;
  public Account(int n) {
    balance = n; // no_warn
  }
}

Java has casts to escape from the type system

IndexOutOfBoundsException

class Vector {
  Object elementData[] guarded_by this;
  int elementCount     guarded_by this;
  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount - 1);
  }
  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n ; i >= 0 ; i--)
      if (elem.equals(elementData[i])) return i;
    return -1;
  }
  synchronized boolean remove(int index) {
    return false;
  }
  synchronized void trimToSize() {
    return;
  }
}

class Vector {
  Object elementData[] guarded_by this;
  int elementCount     guarded_by this;
  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount - 1);
  }
  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n ; i >= 0 ; i--)
      if (elem.equals(elementData[i])) return i;
    return -1;
  }
  synchronized boolean remove(int index) {
    return false;
  }
  synchronized void trimToSize() {
    return;
  }
}

IndexOutOfBoundsException
Part II: Types for Atomicity

Internal Synchronization

```java
class Ref {
    int i;
    ...?
    void inc() {
        synchronized (this) {
            int t = i;
            i = t + 1;
        }
    }
    ...
}
```

Ref.inc(...)
- race-free
- behaves correctly in a multithreaded context

Internal Synchronization Ver. 2

```java
class Ref {
    int i;
    ...?
    void inc() {
        int t = i;
        synchronized (this) {
            i = t + 1;
        }
    }
    ...
}
```

Ref.inc(...)
- race-free
- behaves incorrectly in a multithreaded context

Race freedom does not prevent errors due to unexpected interactions between threads

Atomicity

- The method Ref.inc() is atomic if concurrent threads do not interfere with its behavior
  - (maximal non-interference property)
- Guarantees that for every execution
  - there is a serial execution with same behavior

Atomicity
- Canonical property
  - (cmp. linearizability, serializability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (80%+) are atomic
  - many interfaces described as “thread-safe”
- Can verify atomicity using a type system
  - atomicity violations often indicate errors
  - leverages Lipton’s theory of reduction

Reduction [Lipton 75]

acq(this) x y i=i Y i=i+1 Z rel(this)
Checking Atomicity

atomic void inc() {
    int t;
    synchronized (this) {
        t = i;
        i = t + 1;
    }
}

R: right-mover lock acquire
L: left-mover lock release
B: both-mover race-free variable access
A: atomic conflicting variable access

• Reducible blocks have form: (R|B)* [A] (L|B)*

Checking Atomicity (cont.)

atomic void inc() {
    int t;
    synchronized (this) {
        t = i;
        i = t + 1;
    }
    synchronized (this) {
        i = t + 1;
    }
}

R: right-mover lock acquire
L: left-mover lock release
B: both-mover race-free variable access
A: atomic conflicting variable access

java.lang.StringBuffer

/**
   ... used by the compiler to implement the binary string concatenation operator ...

   StringBuffers are safe for use by multiple threads as the methods are synchronized so that all the operations on any particular instance behave as if they were executed in order that is consistent with the order of the method calls made by each of the individual threads involved.
*/

public atomic class StringBuffer { ... }

java.lang.StringBuffer

class StringBuffer {
    private int count;
    synchronized int length() { return count; }
    synchronized void getChars(...) { ... }
    atomic synchronized void append(StringBuffer sb) {
        int len = sb.length();
        ... sb.getChars(..., len,...);  
    }
}

false