Announcements

- Rainbow grades: HW1-9, Quiz1-6, Exam1-2
- Grading: HW10-11, Quiz7-8
- HW12 due December 11th
  - Submitly config coming up, sorry for the delay...
- Final exam, December 17th 3-6pm
  - Check for conflicts, let us know ASAP. Thanks!
  - Practice problems and back tests coming up

Final Exam is Cumulative

- Programming Language Syntax (Ch. 2.1-2.3.3)
- Logic Programming and Prolog (Ch. 12)
- Scoping (Ch. 3.1-3.3)
- Programming Language Semantics (Sc. Ch. 4.1-4.3)
- Functional programming (Ch. 11)
  - Scheme and Haskell, map/fold questions
- Lambda calculus (Ch. 11 Companion)
- Data abstraction: Types (Ch. 7, 8)
- Control abstraction: Parameter Passing (Ch. 9.1-9.3)
- Object-oriented languages (10.1-10.2)
- Scala
- Concurrency (13.1). “What can go wrong?” questions
- Dynamic languages (Ch. 14 optional)
- Comparative Programming Languages

Last Class

- Intro to Concurrency
- Concurrency in Java
  - Threads
  - Synchronized blocks
  - The Executor framework
  - What can go wrong with threads?

Today's Lecture Outline

- Transaction server (HW12)
  - Overview
  - Consistency with two-phase locking
- What can go wrong?
- Data race detection
- Alternatives: the Actor model
- Memory consistency models

Transaction Server (HW12)

- Executing transactions
  - In parallel
  - Transactions are atomic --- they execute at once, as a single indivisible operation
- Any outcome that corresponds to a serialized execution is consistent/correct

A: 0, B: 1, C: 2, D: 3
Transaction 1  Transaction 2  Transaction 3
B=C; A=A+B  B=D+A  D=C

Transaction Server (HW12)

- Transactions execute under the optimistic (i.e., speculative) concurrency model
- Each transaction proceeds optimistically, i.e.,
  - expecting that no conflicts will occur
- If a transaction detects a conflict with another transaction, it aborts and retries later

Transaction 1  Transaction 2  Transaction 3
B=C; A=A+B  B=D+A  E=C
- What conflicts do we have here?
Conflicts

- A conflict occurs if two transactions access the same account “simultaneously” and at least one access is a write.

Transaction 1  Transaction 2  Transaction 3
B=C; A=A+B  B=D+A  E=C

A write-write conflict between T1 and T2 (on B)
A read-write conflict between T1 and T2 (on A and B)

Class Account

```plaintext
value
writer     // Only one writer thread is allowed
readers    // Many reader threads are allowed

peek()     // peek the value of Account object
            // Can be called before open (...)
update()   // Must be called after account has
            // been OPENED (i.e., locked) for
            // writing by current transaction
```

Overview of Implementation

- Task executes the transaction in local cache.
  - Read and compute in local cache.
- Task keeps track of accounts that are written or read.
- Task writes back from cache into global accounts (may fail due to conflicts, in which case task must abort and retry transaction).

Example

Global shared accounts: A: 0  B: 1  C: 2

Task runs transaction A=B+C; B=A+B

Task works in local cache:

**Original values:** A: 0, B: 1, C: 2

**Cache values:** A: 3, B: 4, C: 2

written:  A: yes, B: yes, C: no
read:    A: yes, B: yes, C: yes

Write back

- Now, task needs to write local cache to global shared accounts.
- Here is an outline (however, there are details you need to figure out yourself!)

**Phase 1.** Open (i.e., lock) all accounts

Account.open checks for conflicts

Wrap Account.open. Catch AbortException

Once all accounts are successfully opened (i.e., locked), write back (commit) accounts.

**Phase 2.** Close (i.e., unlock) all accounts
One more thing...

- I didn’t specify in what order transaction “opens all accounts”
- What can happen if transaction opens the accounts in the order it sees them
  - E.g., A = B +10 opens A then B, and B = A - 10 opens B then A
  - DEADLOCK!
- Solution: each transaction opens accounts in the same well-defined order, from A to Z

What Can Go Wrong?

- New types of bugs occur in concurrent programs
  - Race conditions
  - Atomicity violations
  - Deadlocks
- There is nondeterminism in concurrency, which makes reasoning about program behavior extremely difficult

What Can Go Wrong? JDK 1.4.2

```java
public abstract class Writer {
    protected Object lock;
    Writer() { this.lock = this; }
    public void write(char ch) { ... }
    public void write(String str){ ... }
}

public class PrintWriter extends Writer {
    protected Writer out;
    public PrintWriter(Writer out) {
        super(); this.out = out;
    }
}
```

What Can Go Wrong? (cont.)

```java
public void println(int x) {
    synchronized (lock) {
        out.write(Integer.toString(x))
    }
}
```

Concurrent Programming is Difficult

- Concurrent programming is about managing shared mutable state
  - Exponential number of interleavings of thread operations
- OO concurrency: complex shared mutable state
  - Defense: design principles to reduce complexity
  - Defense: immutable classes, objects, or references
  - Defense: avoid representation exposure

Today’s Lecture Outline

- Transaction server (HW12)
  - Overview
  - Consistency with two-phase locking
- What can go wrong?
- Data race detection
- Alternatives: the Actor model
- Memory consistency models
Data Race Detection

- Dynamic data race detectors --- instrument program to monitor its execution and detect data races at runtime
  - E.g., ThreadSanitizer is a publicly available tool
  - Happens-before race detectors
- Static data race detectors --- analyze program before program execution, and report potential data races
  - Less successful than dynamic detectors

Happens-before Race Detectors

- Most popular dynamic race detectors are based on the happens-before ordering relation
  - Happens-before relation due to Leslie Lamport (Turing Award 2013)
  - Happens-before is a partial order on all thread operations
    - Within a thread, operations are totally ordered
    - Between threads, operations are ordered, if they are synchronized with one another

Happens-before: Example

Thread A:
- lock(o)
- r1 = v
- r1 += 1
- v = r1
- unlock(o)

Thread B:
- lock(o)
- r2 = v
- r2 += 1
- v = r2
- unlock(o)

Happens-before: Another Example

Thread A:
- y = y+1
- lock(o)
- v = v+1
- unlock(o)

Thread B:
- lock(o)
- v = v+1
- unlock(o)
- y = y+1

Happens-before Race Detectors

- If two threads access a shared variable, where 1) at least one access is a write, and
  2) accesses are not ordered by happens-before, detector reports a data race
- Notoriously difficult to implement
- Slowdown. Some detectors run 10x to 30x
- Race report depends on schedule!
Lockset-based Race Detectors

- Late 90’s. Dynamic race detectors based on the lockset algorithm
  - Savage et al., ACM TOPLAS, Vol. 15, No. 4, Nov. 97, Eraser: A Dynamic Data Race Detector for Multithread Programs
  - Improves on happens-before
  - Efficiency is still an issue
  - May have false positives

Lockset-based Race Detectors

- Race detector maintains lockset $C(v)$ for each shared variable $v$
  - Lockset is initialized to all locks and refined as program executes. If lockset becomes empty, issue a warning
  - Many variations of lockset-based race detectors

Let $\text{locks}_\text{held}(t)$ be the locks held by thread $t$

Initialize $C(v)$ to the set of all locks
On each access to $v$

$C(v) = C(v) \cap \text{locks}_\text{held}(t)$
if $C(v)$ is empty issue a warning

Today’s Lecture Outline

- Transaction server (HW12)
  - Overview
  - Consistency with two-phase locking

- What can go wrong?
  - Data race detection
  - Alternatives programming abstractions: the Actor model

- Memory consistency models

The Actor Model

- The Actor Model is a message-passing model (in contrast to Java’s shared-memory model)
  - Communication is explicit via message send/receive
  - Synchronization is implicit, i.e., no need for explicit synchronization with locks

- An actor is a separate thread of computation
  - Actors share nothing!
  - Each actor has a “mailbox” (infinite message queue)
  - An actor sends messages to other actors
  - An actor consumes messages (typically) in the order they are received

Scala Actors

- Actors are objects that extend the `scala.actors.Actor` class
  - method `receive` handles incoming messages
  - `receiver ! message` sends message `message` from this Actor to Actor `receiver`

- Important! `scala.actors` was deprecated in 2.10. and removed from 2.11. and onward
  - Scala now uses Akka Actors
  - Code I wrote for lecture uses old Scala actors; revert to 2.10 to run it; avoids dependencies!

The Actor Model

- Due to Carl Hewitt (1970s) developed as part of AI research

- Actor languages in the real world: Erlang, Scala, SALSA
Ping and Pong Actors

```scala
class Ping(count: Int, pong: Actor) extends Actor {
  def act() { // Java's run
    var pingsLeft = count - 1
    pong ! Ping
    while (true) {
      receive {
        case Pong => if (pingsLeft % 1000 == 0) Console.println("Ping: pong")
        pong ! Ping
        pingsLeft = pingsLeft - 1
        else {
          Console.println("Ping: stop")
          pong ! Stop
          exit()
        }
      }
    }
  }
}
```

Call to `receive` suspends this actor until a Pong message is received. Once received, execution proceeds with if statement.

### Producer Consumer

- Classical concurrency problem
- Producers produce items in shared buffer
- Consumers consume items from buffer
- Each producer, consumer runs in its own thread

```scala
object PingPong {
  def main(args: Array[String]) {
    val pong = new Pong
    val ping = new Ping(100000, pong)
    pong.start // Java's start
    ping.start
  }
} $ scala PingPong // yields what?
Pong : ping 0
Ping : pong
Pong : 1000
Ping : pong
Pong : ping 99000
Ping : pong
Pong : stop
Ping : stop
```

Solution in Java: BoundedBuffer

```java
class BoundedBuffer {

  protected int numSlots;
  // size of buffer (queue)
  private int[] buffer; // buffer (queue)
  private int takeOut = 0, putIn = 0;
  // front (takeOut) and back (putIn) of circular queue
  private int count = 0;
  // number of items currently in queue
}
```

### BoundedBuffer.put (i.e., produce)

```java
class BoundedBuffer {

  public synchronized void put(int value) throws InterruptedException {
    while (count == numSlots)
      wait(); // blocks current thread
    buffer[putIn] = value;
    putIn = (putIn + 1) % numSlots;
    count++;
    notifyAll(); // notifies all threads
  }
  // waiting on lock
```

sender is a method in Actor. Refers to actor that sent the last message.
public synchronized int get() throws InterruptedException {
    while (count == 0)
        wait();
    int value = buffer[takeOut];
    takeOut = (takeOut + 1) % numSlots;
    count--;
    notifyAll();
    return value;
}

class Producer implements Runnable {
    BoundedBuffer buffer;...
    void run() {
        int i;
        for (i=0;i<100;i++)
            buffer.put(i);
        Thread.sleep(100); // milliseconds
    }
}

class Consumer implements Runnable {
    BoundedBuffer buffer;...
    void run() {
        int i;
        for (i=0;i<50;i++)
            v = buffer.get();
        Thread.sleep(500); // milliseconds
    }
}

public static void main {
    // shared mutable state buffer
    BoundedBuffer buffer = new BoundedBuffer(5);
    ExecutorService pool = Executors.newFixedThreadPool(3);
    Runnable prod1 = new Producer(buffer);
    Runnable cons1 = new Consumer(buffer);
    Runnable cons2 = new Consumer(buffer);
    pool.execute(prod1);
    pool.execute(cons1);
    pool.execute(cons2);
    pool.shutdown();
}

Scala. Producer and Consumer Actors

class Producer(c : Actor) extends Actor {
    def produce(x: Int) {
        receive {
            case Next =>
                c ! Some(x)
            case Stop => exit()
        }
    }
    def act() {
        var value = 0;
        loop {
            value = value+1;
            produce(value)
        }
    }
}

class Consumer(c : Actor) extends Actor {
    def consume = {
        c ! Next;
        receive {
            case Some(x) => x
        }
    }
    def act() {
        var v = 0;
        while (count < 25) {
            v = consume;
            // process v
        }
    }
}

Scala. Coordinator Actor

class Coordinator extends Actor {
    var producer : Actor = null;
    var consumer : Actor = null;
    … code to initialize producer and consumer
    def act() {
        loop {
            receive { // waits for message from Consumer
                case Next => {
                    producer ! Next; // sends to Producer
                    receive { // relays response to Consumer
                        case Some(x) => consumer ! Some(x)
                    }
                    case Stop => { producer ! Stop; exit() }
                }
            }
        }
    }
}
Scala. Launching the Actors

```scala
object Producers {
  def main(args : Array[String]) {
    val coordinator = new Coordinator;
    val producer = new Producer(coordinator);
    val consumer = new Consumer(coordinator);
    coord.init(producer,consumer);
    producer.start;
    coordinator.start;
    consumer.start;
  }
}
```

Key Points

- No “shared state” between actors!
- Consumer and Producer actors communicate via `explicit` message sends/receives
  - Producer sends item!
- No `explicit` synchronization to control order of thread operations, i.e., no `synchronized`, `wait`, etc.
  - In our examples, `receive` causes thread to wait. `Implicit` ordering of operations
- There is also `asynchronous` message passing

Shared-memory vs. Message-passing

- Compare and contrast shared-memory models with message-passing
- Pros of shared-memory?
- Pros of message-passing?

Today’s Lecture Outline

- Transaction server (HW12)
  - Overview
  - Consistency with two-phase locking
- What can go wrong?
- Data race detection
- Alternatives programming abstractions: the Actor model
- Memory consistency models

Memory Consistency Models

- When more than one location is written at about the same time, we must worry about the order the `writes` become visible to different processors
- **Sequential consistency** is the intuitive model
  - We want all `writes` to become visible at the same time to all processors
  - We want a given processor’s `writes` to be visible in the order they were performed
  - Sequential consistency is so hard, that processors simply do not provide it!

Memory Consistency Models (Sc. p. 611)

Initially: `inspected = false; X = 0`

Processor A

```
inspected := true X := 1
xa := X
```

Processor B

```
ib := inspected
```

Processor A and Processor B execute their respective code at about the same time.
What values do you expect for `xa` and `ib`?
Memory Consistency Models (Sc. p. 611)

Initially: $\text{inspected} = \text{false}; \ X = 0$
Processor A Processor B

$\text{inspected} := \text{true} \quad X := 1$

$xa := X \quad \text{ib} := \text{inspected}$

In fact, processors buffer writes and they don’t become visible to other processors right away! It is possible to get $xa = 0$ and $\text{ib} = \text{false}$!

volatile keyword in Java

- $\text{volatile } x$ forces reads and writes on $x$ to be ordered across threads
  - Intuitively, a read of $x$ (by one processor) will see the most recently written value (by different one). Buffering is not allowed on volatile variables

How can we “fix” our example?
Initially: $\text{inspected} = \text{false}; \ X = 0$
Thread A Thread B

$\text{inspected} := \text{true} \quad X := 1$

$xa := X \quad \text{ib} := \text{inspected}$

Java Memory Model

- Java maintains “sequentially consistency”
- A thread can buffer its writes until it writes a volatile variable or exits a synchronized block
- A thread can use cached (buffered) values until it reads a volatile variable or enters a synchronized block