Programming Language Concepts: Lecture 13

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 13, 09 March 2009

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  1. Cars on the bridge going in the same direction $\Rightarrow$ can cross
  2. No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  3. Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.
- When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more.

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway synchronization primitives
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

An example . . .

- Design a class Bridge to implement consistent one-way access for cars on the highway synchronization primitives
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)

- Bridge has a public method

```
public void cross(int id, boolean d, int s)
```

  - id is identity of car
  - d indicates direction
    - true is North
    - false is South
  - s indicates time taken to cross (milliseconds)

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  1. A car is stuck waiting for the direction to change
     `Car 7 going North stuck at Thu Mar 13 23:00:11 IST 2009`
  2. The direction changes
     `Car 5 switches bridge direction to North at Thu Mar 13 23:00:14 IST 2009`
  3. A car enters the bridge.
     `Car 8 going North enters bridge at Thu Mar 13 23:00:14 IST 2003`
  4. A car leaves the bridge.
     `Car 16 leaves at Thu Mar 13 23:00:15 IST 2003`

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
    1. A car is stuck waiting for the direction to change
       `Car 7 going North stuck at Thu Mar 13 23:00:11 IST 2009`
    2. The direction changes
       `Car 5 switches bridge direction to North at Thu Mar 13 23:00:14 IST 2009`
    3. A car enters the bridge.
       `Car 8 going North enters bridge at Thu Mar 13 23:00:14 IST 2003`
    4. A car leaves the bridge.
       `Car 16 leaves at Thu Mar 13 23:00:15 IST 2003`
- Use `java.util.Date` to generate time stamps

Analysis

- The "data" that is shared is the `Bridge`

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — an `int`
  - Current direction of bridge — a `boolean`

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — an `int`
  - Current direction of bridge — a `boolean`
- The method

  ```
  public void cross(int id, boolean d, int s)
  ```

  changes the state of the bridge

Analysis

- The “data” that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — an `int`
  - Current direction of bridge — a `boolean`
- The method

  ```
  public void cross(int id, boolean d, int s)
  ```

  changes the state of the bridge
  - Concurrent execution of `cross` can cause problems ...

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — an `int`
  - Current direction of bridge — a `boolean`
- The method

  ```
  public void cross(int id, boolean d, int s)
  ```

  changes the state of the bridge
  - Concurrent execution of `cross` can cause problems . . .
  - . . . but making `cross` a synchronized method is too restrictive
    - Only one car on the bridge at a time
    - Problem description explicitly disallows such a solution

Analysis . . .

- Break up `cross` into a sequence of actions

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
- Make `enter` and `leave` synchronized

Analysis ...

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
- Make `enter` and `leave` synchronized
- `travel` is just a means to let time elapse — use `sleep`

Analysis . . .

Code for `cross`

```
public void cross(int id, boolean d, int s){

    // Get onto the bridge (if you can!)
    enter(id,d);

    // Takes time to cross the bridge
    try{
        Thread.sleep(s);
    }
    catch(InterruptedException e){}

    // Get off the bridge
    leave(id);
}
```

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter
- Otherwise, `wait()` for the state of the bridge to change

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter
- Otherwise, `wait()` for the state of the bridge to change
- In each case, print a diagnostic message

Code for enter

```
private synchronized void enter(int id, boolean d){
    Date date;

    // While there are cars going in the wrong direction
    while (d != direction && bcount > 0){

        date = new Date();
        System.out.println("Car "+id+" going "+direction_name(d)+"

        // Wait for our turn
        try{
            wait();
        }
        catch (InterruptedException e){}
    }

    ...

}
```

Code for enter

```
private synchronized void enter(int id, boolean d){
    ...
    while (d != direction && bcount > 0){ ... wait() ...}
    ...

    // Switch direction, if needed
    if (d != direction){
        direction = d;
        date = new Date();
        System.out.println("Car "+id+" switches bridge direction
           to "+direction_name(direction)+" at "+date);
    }

    // Register our presence on the bridge
    bcount++;

    date = new Date();
    System.out.println("Car "+id+" going "+direction_name(d)+"
       enters bridge at "+date);
}
```

Analysis . . .

Leaving the bridge is much simpler

- Decrement the car count

Analysis . . .

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars

Analysis . . .

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars

  . . . provided car count is zero

Analysis . . .

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars

  . . . provided car count is zero

```
private synchronized void leave(int id){
    Date date = new Date();
    System.out.println("Car "+id+" leaves at "+date);

    // "Check out"
    bcount--;

    // If everyone on the bridge has checked out, notify the
    // cars waiting on the opposite side
    if (bcount == 0){
        notifyAll();
    }
}
```

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors ... to lock and unlock shared data

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors . . . to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors . . . to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well
- With multicore architectures, concurrent programming will become more ubiquitous
- Goal
  - Design a new mechanism for reliable, modular concurrent programming with shared data

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors . . . to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well
- With multicore architectures, concurrent programming will become more ubiquitous
- Goal
  - Design a new mechanism for reliable, modular concurrent programming with shared data
  - Software Transactional Memory!

The problem with locks

A bank account class

```
class Account {
  Int balance;

  synchronized void withdraw( int n ) {
    balance = balance - n;
  }

  synchronized void deposit( int n ) {
    withdraw( -n );
  }
}
```

- Each object has a lock
- synchronized methods acquire and release locks

The problem with locks . . .

How do we transfer money from one account to another?

The problem with locks . . .

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

The problem with locks . . .

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

Is there a problem?

The problem with locks ...

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

Is there a problem?

- Intermediate state when money has left `from` and not been deposited in `to` should not be visible!
- Having `withdraw` and `deposit` synchronized does not help

The problem with locks . . .

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

The problem with locks . . .

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

The problem with locks . . .

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

- Two concurrent transfers in opposite directions between accounts `i` and `j` can deadlock!

The problem with locks ...

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

The problem with locks . . .

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

The problem with locks . . .

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

- Need to know all possible locks in advance

The problem with locks . . .

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `from` is a Super Savings Account in which most of the money is in a medium term fixed deposit `fromFD`?

The problem with locks . . .

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `from` is a Super Savings Account in which most of the money is in a medium term fixed deposit `fromFD`?
- `from.withdraw(amt)` may require an additional transfer from `fromFD` to `from`
  - `transfer` may not know anything about `fromFD`
  - Even if it did, it has to acquire a third lock

The problem with locks . . .

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `transfer` can block in case of insufficient funds?
  - Wait on a condition variable (monitor queue)
  - Becomes more complex as number of locks increase

The problem with locks . . .

- Take too few locks — data integrity is compromised
- Take too many locks — deadlocks, lack of concurrency
- Take wrong locks, or in wrong order — connection between lock and data it protects is informal
- Error recovery — how to recover from errors without leaving system in an inconsistent state?
- Lost wake-ups, erroneous retries — Easy to forget to signal a waiting thread, recheck condition after wake-up

The problem with locks . . .

- Take too few locks — data integrity is compromised
- Take too many locks — deadlocks, lack of concurrency
- Take wrong locks, or in wrong order — connection between lock and data it protects is informal
- Error recovery — how to recover from errors without leaving system in an inconsistent state?
- Lost wake-ups, erroneous retries — Easy to forget to signal a waiting thread, recheck condition after wake-up

Lack of modularity

Cannot easily make use of smaller programs to build larger ones

- Combining `withdraw` and `deposit` to create `transfer` requires exposing locks

Transactions

- Import idea of transactions from databases
  - Hardware support for transactions in memory
    [Herlihy,Moss 1993]
- Instead, move transaction support to run time software
  - Software Transactional Memory [Shavit,Touitou 1995]
- An implementation in Haskell
  [Harris, Marlow, Peyton Jones, Herlihy 2005]
  - Tutorial presentation
    Simon Peyton Jones: Beautiful concurrency,
    in *Beautiful code*, ed. Greg Wilson, OReilly (2007)

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)
  - Maintain a transaction log for each transaction, noting down values that were written and read
  - If a value is written in a transaction and read later, look it up in the log
  - At the end of the transaction, use log to check consistency

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)
  - Maintain a transaction log for each transaction, noting down values that were written and read
  - If a value is written in a transaction and read later, look it up in the log
  - At the end of the transaction, use log to check consistency
- If no inconsistency was seen, commit the transaction
- Otherwise, roll back and retry

Transactions . . .

Use `atomic` to indicate scope of transactions

```
void withdraw( int n ) {
  atomic{ balance = balance - n; }
}

void deposit( int n ) {
  atomic{ withdraw( -n ); }
}
```

Transactions . . .

Use atomic to indicate scope of transactions

```
void withdraw( int n ) {
  atomic{ balance = balance - n; }
}

void deposit( int n ) {
  atomic{ withdraw( -n ); }
}
```

Now, building a correct version of transfer is not difficult

```
void transfer( Account from, Account to, Int amount ) {
  atomic { from.withdraw( amount );
           to.deposit( amount ); }
}
```

Transaction interference

Independent transactions updating the same object

```
atomic{                                 // Transaction 1
  if a.getName().equals("B")
     a.setVal(8);
}

atomic{                                 // Transaction 2
  int previous = a.getVal();
  a.setVal(previous+1);
}
```

Transaction interference

Independent transactions updating the same object

```
atomic{                              // Transaction 1
  if a.getName().equals("B")
     a.setVal(8);
}

atomic{                              // Transaction 2
  int previous = a.getVal();
  a.setVal(previous+1);
}
```

- If Transaction 1 executes between first and second instruction of Transation 2, transaction log shows that value of `previous` is inconsistent
- Transaction 2 should roll back and reexecute

Transactions . . .

What else do we need?

Transactions . . .

What else do we need?

- Blocking
  - If amount to be withdrawn is more than current balance, wait

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    if (amount < from.balance) retry;
    from.withdraw ( amount );
    to.deposit( amount );
  }
}
```

Transactions . . .

What else do we need?

- Blocking
  - If amount to be withdrawn is more than current balance, wait

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    if (amount < from.balance) retry;
    from.withdraw ( amount );
    to.deposit( amount );
  }
}
```

- `retry` suspends transaction without any partial, inconsistent side-effects
- Transaction log indicates possible variables that forced `retry`
- Wait till one of these variables changes before attempting to rerun transaction from scratch

Transactions . . .

What else do we need?

Transactions . . .

What else do we need?

- Nested `atomic` allows sequential composition
- How about choosing between transactions with alternatives
  - If amount to be withdrawn is more than current balance, move money from linked fixed deposit

Transactions ...

What else do we need?

- Nested `atomic` allows sequential composition
- How about choosing between transactions with alternatives
  - If amount to be withdrawn is more than current balance, move money from linked fixed deposit

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    atomic{ from.withdraw ( amount ); }
    orElse
    atomic{ LinkedFD[from].withdraw ( amount ); }

    to.deposit( amount );
  }
}
```

What could go wrong?

```
void b( Account from, Account to, Int amount ) {
  atomic {
    x = a.getVal();
    y = b.getVal();
    if (x > y){ launchMissiles(); }
    ...
  }
}
```

What could go wrong?

```
void b( Account from, Account to, Int amount ) {
  atomic {
    x = a.getVal();
    y = b.getVal();
    if (x > y){ launchMissiles(); }
    ...
  }
}
```

- If an inconsistency is found later, the transaction should roll back and retry
- How do we recall the missiles that have been launched?
- Need a strong type system to ensure that transactions affect only transactional memory

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full
- Reasonable to expect that value in `a` is pushed back into `q1`.

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full
- Reasonable to expect that value in `a` is pushed back into `q1`.

How about

```
try { atomic{
        a = q1.extract();  q2.insert(a);
    }
catch (QueueFullException e) { a = q3.extract() } ;
```

- What is the state of `q1`?

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes “sequential” code
  - Implementation determines granularity of concurrency — e.g. using transaction logs

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`
- Need to restrict what transactions can encompass — `LaunchMissiles()`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`
- Need to restrict what transactions can encompass — `LaunchMissiles()`
- Exceptions and transactions interact in a complex manner