Who's afraid of concurrent programming?

Madhavan Mukund

Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

ACM, Chennai Professional Chapter
13 November 2010

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access

Concurrent programming

- Multiprocessing
  - Single processor executes several computations “in parallel”
  - Time-slicing to share access
- Logically parallel actions within a single application
  - Clicking `Stop` terminates a download in a browser
  - User-interface is running in parallel with network access
- Process
  - Private set of local variables
  - Time-slicing involves saving the state of one process and loading the suspended state of another
- Threads
  - Operated on same local variables
  - Communicate via “shared memory”
  - Context switches are easier

The challenge of concurrent programming

- Concurrent programming is difficult
  - Carefully coordinate access to shared data
  - Race conditions may create scheduler-dependent bugs
    - Hard to detect and reproduce

The challenge of concurrent programming

- Concurrent programming is difficult
  - Carefully coordinate access to shared data
  - Race conditions may create scheduler-dependent bugs
    - Hard to detect and reproduce
- Programming languages offer features to support concurrent programming
  - Synchronization mechanisms: semaphores, locks, monitors
  - Still have to deal with deadlocks, granularity issues

The challenge of concurrent programming

- Concurrent programming is difficult
  - Carefully coordinate access to shared data
  - Race conditions may create scheduler-dependent bugs
    - Hard to detect and reproduce
- Programming languages offer features to support concurrent programming
  - Synchronization mechanisms: semaphores, locks, monitors
  - Still have to deal with deadlocks, granularity issues
- Fortunately, concurrent programming is usually left to "specialists"
  - Operating system schedulers
  - Webservers
  - ...

Multicore architectures

- Physical constraints make it impossible to further shrink and speed up CPUs
- Instead, pack multiple CPU "cores" on a single chip
  - 2 cores are standard today ("dual core")

Multicore architectures

- Physical constraints make it impossible to further shrink and speed up CPUs
- Instead, pack multiple CPU "cores" on a single chip
  - 2 cores are standard today ("dual core")
- To speed up applications, need to exploit the underlying parallelism in hardware

School of thought

Multicore architectures will make concurrent programming more ubiquitous

Multicore architectures

- Physical constraints make it impossible to further shrink and speed up CPUs
- Instead, pack multiple CPU "cores" on a single chip
  - 2 cores are standard today ("dual core")
- To speed up applications, need to exploit the underlying parallelism in hardware

School of thought

Multicore architectures will make concurrent programming more ubiquitous

- If so, we'd better make it easier to write and debug concurrent programs!

Race conditions

- Shared variables must be updated consistently

```
Thread 0
...
m = n;
m++;
n = m;
```

```
Thread 1
...
k = n;
k++;
n = k;
```

- Expect `n` to increase by 2 . . .

Race conditions

- Shared variables must be updated consistently

```
Thread 0
...
m = n;
m++;
n = m;
```

```
Thread 1
...
k = n;
k++;
n = k;
```

- Expect `n` to increase by 2 . . .
- . . . but, time-slicing may order execution as follows

```
Thread 1: m = n;
Thread 1: m++;
Thread 2: k = n;   // k gets the original value of n
Thread 2: k++;
Thread 1: n = m;
Thread 2: n = k;   // Same value as that set by Thread 1
```

Race conditions

- Even a direct update to a single variable is problematic

```
Thread 0
...
n++;
```

```
Thread 1
...
n++;
```

- `n++` typically breaks up as three steps
  - Load `n` from memory to register `r`
  - Increment `r`
  - Store value of `r` back at memory location `n`
- Uncontrolled interleaving can again produce inconsistent updates

Peterson's algorithm

```
Thread 0

...
request_0 = true;
turn = 1;

while (request_1 &&
       turn != 0){}
  // "Busy" wait

// Enter critical section
   ...
// Leave critical section
request_0 = false;
...
```

```
Thread 1

...
request_1 = true;
turn = 0;

while (request_0 &&
       turn != 1){}
  // "Busy" wait

// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

- If both try simultaneously, `turn` decides who goes through
- If only one is alive, `request` for that process is stuck at false and `turn` is irrelevant

Peterson experiment

- Two parallel threads
  - Each increments a shared integer `accumulate` in a loop 500,000 times
  - Critical section protected by Peterson's algorithms
  - Expected final value of `accumulate` is 1,000,000

Peterson experiment

- Two parallel threads
  - Each increments a shared integer `accumulate` in a loop 500,000 times
  - Critical section protected by Peterson's algorithms
  - Expected final value of `accumulate` is 1,000,000
- Implementation using `pthreads` in `C`
  - Intel Core 2 Duo, MacOS
    - 20–30% of runs show inconsistent updates
  - Intel Xeon, single core, Linux
    - 80% of runs show inconsistent updates!
- What's going on?

A simpler example

- Initially, shared values `x = y = 0`.

Thread 0	Thread 1
`x = 1;`	`y = 1;`
`r0 = y;`	`r1 = x;`

A simpler example

- Initially, shared values `x = y = 0`.

```
Thread 0

x = 1;
r0 = y;
```

```
Thread 1

y = 1;
r1 = x;
```

- Possible outcomes

r0	r1
0	1
1	0
1	1

`Thread 0` completes before `Thread 1` starts

`Thread 1` completes before `Thread 0` starts

`Interleaving occurs`

A simpler example

- Initially, shared values `x = y = 0`.

```
Thread 0                Thread 1

x = 1;                  y = 1;
r0 = y;                 r1 = x;
```

- Possible outcomes

r0	r1
0	1
1	0
1	1

`Thread 0` completes before `Thread 1` starts
`Thread 1` completes before `Thread 0` starts
`Interleaving occurs`

- Experimentally, `r0 = 0` and `r1 = 0` are also observed!
- Instructions are being reordered!

Sequential consistency

- Multiple sequential threads read and write to shared memory

Sequential Consistency [Lamport 1979]

. . . the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

In other words . . .

- Computations of different processes are interleaved
- Program order is preserved for each process

Relaxing sequential consistency

- Instructions in a process may be executed out of order
- Compiler optimization

```
r1 = x
r2 = x
if (r1 == r2)
   y = 1
```

```
y = 1
r1 = x
r2 = x
if (true)
```

- Hardware: reduce latency of writes

Relaxing program order

```
Thread 0

x = 1
r0 = y
```

```
Thread 1

r1 = x
```

Relaxing program order

```
Thread 0                    Thread 1

x = 1                       r1 = x
r0 = y
```

- Sequentially consistent schedule requires 3 steps

```
x = 1    [Step 1]           r1 = x    [Step 2]
r0 = y   [Step 3]
```

Relaxing program order

```
Thread 0                    Thread 1

x = 1                       r1 = x
r0 = y
```

- Sequentially consistent schedule requires 3 steps

```
x = 1    [Step 1]           r1 = x   [Step 2]
r0 = y   [Step 3]
```

- Reordering allows parallel access to disjoint variables, 2 step schedule

```
r0 = y   [Step 1]           r1 = x   [Step 1]
x = 1    [Step 2]
```

Relaxing memory models

Relaxed hardware memory models in use for years!

- TSO Total Store Ordering
  `read(y)` can "overtake" `write(x)`
  `x = 5; r = y` $\mapsto$ `r = y; x = 5`
  - As though all writes are buffered in a single queue

Relaxing memory models

Relaxed hardware memory models in use for years!

- TSO Total Store Ordering
  `read(y)` can "overtake" `write(x)`
  `x = 5; r = y` $\mapsto$ `r = y; x = 5`
  - As though all writes are buffered in a single queue

- PSO Partial Store Ordering
  `write(y)` can "overtake" `write(x)`
  `x = 5; y = 7` $\mapsto$ `y = 7; x = 5`
  - Each location has a separate write buffer

Relaxing memory models

Relaxed hardware memory models in use for years!

- TSO Total Store Ordering
  `read(y)` can "overtake" `write(x)`
  `x = 5; r = y` $\mapsto$ `r = y; x = 5`
  - As though all writes are buffered in a single queue
- PSO Partial Store Ordering
  `write(y)` can "overtake" `write(x)`
  `x = 5; y = 7` $\mapsto$ `y = 7; x = 5`
  - Each location has a separate write buffer
- RMO Relaxed Memory Ordering
  `read(y)` can "overtake" `read(x)` and `read(y)`
  `x = 5; r = x; y = 7` $\mapsto$ `y = 7; x = 5; r = x`

Examples

- Intel x86, SPARC, AMD typically implement TSO
- PowerPC typically implements RMO

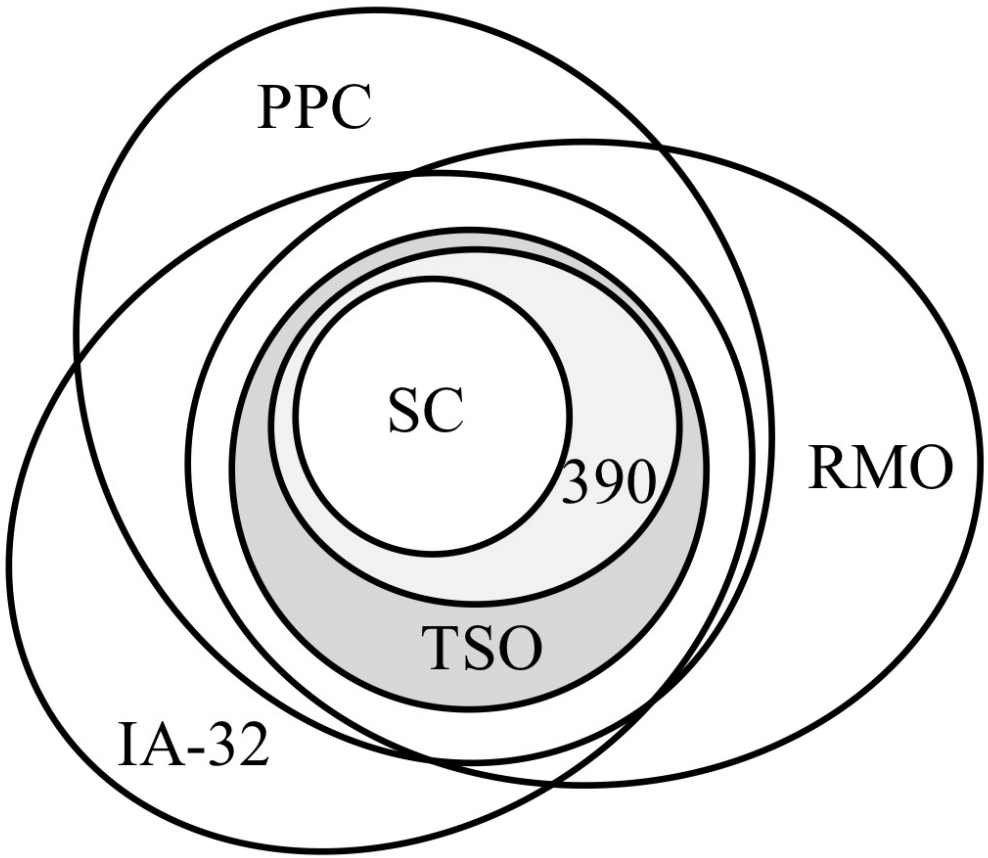
Examples

- Intel x86, SPARC, AMD typically implement TSO
- PowerPC typically implements RMO

- Hardware manufacturers reluctant to fully document memory models they implement

- Avoid commitment to maintain compatibility as hardware evolves!

The memory model zoo



Programming with weak memory models

- How can programming languages implement constructs like locks etc which require sequential consistency?

Programming with weak memory models

- How can programming languages implement constructs like locks etc which require sequential consistency?
- Hardware provides special instructions (`mfence`, ...) to restrict relaxation
- Compilers can use these "fence" instructions to build "barriers" that guarantee sensible semantics sychronization constructs provided in the programming language

Data races

- Using fences, etc, programming languages can provide their own memory models

Data races

- Using fences, etc, programming languages can provide their own memory models
- Data race

  Two adjacent conflicting operations that can be swapped
  - Conflicting memory operations
    - Affect same location, at least one is a write
  - Interleave operations of all threads
    - Adjacent operations from different threads can be swapped

Data races

- Using fences, etc, programming languages can provide their own memory models
- Data race

  Two adjacent conflicting operations that can be swapped
  - Conflicting memory operations
    - Affect same location, at least one is a write
  - Interleave operations of all threads
    - Adjacent operations from different threads can be swapped

Java memory model guarantee

Programs free of data races respect sequential consistency

Java Memory Model

- If the program is not data race-free, all bets are off!
  - Can signal to Java that a shared variable should be synchronized across threads: `volatile`
  - Declare `request_0`, `request_1`, `turn` as `volatile` to implement Peterson's algorithm

Java Memory Model

- If the program is not data race-free, all bets are off!
  - Can signal to Java that a shared variable should be synchronized across threads: `volatile`
  - Declare `request_0`, `request_1`, `turn` as `volatile` to implement Peterson's algorithm
- Java memory model is very complex, not completely robust
  - Check if `P1 || P2` is admissible by incrementally building up a valid execution
  - There are examples where `P1 || P2` has no valid execution, but `P1 ; P2` is admissible!
    - Sequentialization of parallel threads should correspond to a valid schedule!

Other languages

- C++ memory model still being formalized
- Thread libraries like `pthreads` give no guarantees, as we have seen!

Concurrent programming for the masses?

- Concurrent programs are already hard to design and implement correctly
- Locks etc ensure freedom from data races and help overcome complications of relaxed memory models
  - But lock-based programs are tricky to get right
- Can we present a better abstraction to the programmer?
  - Borrow the notion of a transaction from databases
  - Programmer describes "indivisible" units of code
  - Underlying system guarantees atomicity
  - 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 );
  }
}
```

- In Java, 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?

```
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();
}
```

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();
}
```

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`?
- `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

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
- 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")
     s.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?

- 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?

- 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(); }
    ...
  }
}
```

- 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);
}
```

- 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 = q1.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
- 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

Summary

- Multicore technology will make concurrent programming more ubiquitous
- Concurrent programming is already difficult
- Memory models that depart from sequential consistency make life eve n more complex
- Existing lock based techniques do not scale up
- STMs could provide a modular framework for coordinating shared data
  - Not a magic bullet, but allows us to focus on coordination issues at higher level
- Lots of areas still to be explored