Advances in Programming Languages

APL5: Further language concurrency mechanisms

David Aspinall
(including slides by Ian Stark)

School of Informatics
The University of Edinburgh

Tuesday 5th October 2010
Semester 1 Week 3

Programming-Language Techniques for Concurrency

This is the third in a block of lectures presenting some programming-language techniques for managing concurrency.

- Introduction, basic Java concurrency
- Concurrency abstractions in Java
- Concurrency in some other languages

Outline

1. Concurrency mechanisms
2. Actors
3. Software Transactional Memory
4. Summary

Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.

Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.

Various language paradigms have been followed, e.g.:

locks and conditions: tasks share memory and exclude and signal one another using shared memory (e.g., Java);

synchronous message passing: tasks share communication channels and use *rendezvous* to communicate (e.g., Ada, CML, Go);

asynchronous message passing: a task offers a *mail box* which receives messages (e.g. Erlang, Scala Actors);

lock-free algorithms or transactional memory: tasks share memory but detect and repair conflicts (e.g., libraries in Haskell, Clojure)

Concurrency mechanisms

There is a large design space for concurrent language mechanisms. Two requirements are *separation*, to prevent inconsistent access to shared resources, and *co-operation* for communication between tasks.

Various language paradigms have been followed, e.g.:

locks and conditions: tasks share memory and exclude and signal one another using shared memory (e.g., Java);

synchronous message passing: tasks share communication channels and use *rendezvous* to communicate (e.g., Ada, CML, Go);

asynchronous message passing: a task offers a *mail box* which receives messages (e.g. Erlang, Scala Actors);

lock-free algorithms or transactional memory: tasks share memory but detect and repair conflicts (e.g., libraries in Haskell, Clojure)

Language designs have also been influenced by mathematical models used to capture and analyse the essence of concurrent systems, for example, *CSP*, π-*calculus*, the *join calculus*, and the *ambient calculus*.

Reminder: the problems with locks

Deadlock when two threads try to acquire the same locks in different orders;

Reminder: the problems with locks

Deadlock when two threads try to acquire the same locks in different orders;

Priority inversion when the scheduler preempts a lower-priority thread that holds a lock needed for a higher-priority one;

Reminder: the problems with locks

Deadlock when two threads try to acquire the same locks in different orders;

Priority inversion when the scheduler preempts a lower-priority thread that holds a lock needed for a higher-priority one;

Convoying when threads waiting on a lock held by a de-scheduled thread queue up, causing a traffic jam;

Reminder: the problems with locks

Deadlock when two threads try to acquire the same locks in different orders;

Priority inversion when the scheduler preempts a lower-priority thread that holds a lock needed for a higher-priority one;

Convoying when threads waiting on a lock held by a de-scheduled thread queue up, causing a traffic jam;

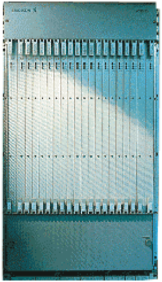
Lack of compositionality there is **no easy way to compose larger thread-safe programs from smaller ones**

Outline

Scala and Erlang

Scala is a functional object-oriented language that compiles to the Java Virtual Machine. It allows full interoperability with Java. Scala is designed by Martin Odersky and his team at EPFL, Lausanne, Switzerland.

Scala's concurrency is based on the *Actor model* also used in several other languages. A notable commercial success story is Ericsson's language *Erlang* designed for massively concurrent telecommunications equipment.



Ericsson AXD 301 multiservice 10–160Gbit/s switch

Nortel 8661 SSL Acceleration Ethernet Routing Switch

Asynchronous message passing

An *actor* is a process abstraction that interacts with other actors by message passing. Message sending is asynchronous. Each actor has a *mail box* which buffers incoming messages. Messages are processed by matching.

Sending

```
actor ! message

// sender is the last actor
// we received from
sender ! message

// shorthand for above
reply(message)
```

Receiving

```
receive {
  case pattern => action
   ...
  case pattern => action
}
```

Example: ping pong

```
class Ping(pong: Actor)
        extends Actor {
  def act() {
    var pings = 0;
    pong ! Ping
    while (true) {
      receive {
        case Pong =>
          pong ! Ping
          pings += 1
          if (pings % 1000 == 0)
            Console.println(
                "Ping: pong "+pings)
          }
    }}}
```

```
class Pong extends Actor {
  def act() {
    var pongs = 0
    while (true) {
      receive {
        case Ping =>
          sender ! Pong
          pongs += 1
      }}}}

object pingpong
       extends Application {
  val pong = new Pong
  val ping = new Ping(pong)
  ping.start
  pong.start
}
```

Reply-response protocols

Actors often take part in sequences of message exchanges, which are more synchronous in nature. There is a special encoding for writing these.

Sending and receiving

actor !? message

is like

```
actor ! (self, message)
receive {
  case pattern => ...
}
```

Event-based actors

Actors are either *thread-based* or *event-based*. Thread based actors block on **receive** calls. Event-based actors provide an alternative which uses a more lightweight mechanism.

Event based receiving

```
react {
  case pattern => action
   ...
  case pattern => action
}
```

A **react** statement encapsulates the rest of a computation for an actor and never returns. The event-based framework generates tasks that process messages and suspend and resume actors, using *continuations* derived from the **react** blocks.

Example: bounded buffer in Scala

```
class BoundedBuffer[T](N: int) {
  private case class Put(x: T)
  private case object Get
  private case object Stop

  def put(x: T) {
     buffer !? Put(x)
  }

  def get: T =
   (buffer !? Get).asInstanceOf[T]

  def stop() {
     buffer !? Stop
  }

private val buffer = actor {
  val buf = new Array[T](N)
  var in = 0; var out = 0; var n = 0
  loop {
    react {
      case Put(x) if n < N =>
        buf(in) = x
        in = (in + 1) % N
        n = n + 1; reply()
      case Get if n > 0 =>
        val r = buf(out)
        out = (out + 1) % N
        n = n - 1; reply(r)
      case Stop => reply()
      exit("stopped")
    }
}}
```

Outline

Software Transactional Memory

Transactional Memory is a lock-free way of managing shared memory between concurrent tasks, inspired by transaction processing in databases. It was proposed and refined by Herlihy, Moss, Shavit and others.

The basic ideas are:

- memory accesses are grouped into *transactions*: sequences of reads and writes;
- each transaction is committed atomically from the point of view of other transactions;
- transactions may be *aborted* and retried.

In practice, transactions are executed with *optimistic concurrency*, detecting interference. If two transactions conflict by reading and writing the same location, one will be aborted and retried.

Software Transactional Memory (STM) is an implementation in software, as part of a library or language runtime.

Example: synchronized unbounded buffer in Java

```
class SynchronizedQueue<T> {

 Node sentinel = new Node(null);
 Node head = sentinel;
 Node tail = sentinel;

 class Node {
     T item;
     Node next;
     Node(T item) {
         this.item = item;
     }
 }
```

```
public synchronized void put(T item) {
    Node node = new Node(item);
    node.next = tail;
    tail = node;
    notifyAll();
}

public synchronized T get()
      throws InterruptedException {
   while (head == tail) {
       wait();
   }
   T item = head.item;
   head = head.next;
   return item;
}

...
```

Example: lock-free unbounded buffer in Java

```
import
 j.u.c.atomic.AtomicReference;

class LockFreeQueue<T> {
 AtomicReference<Node> head;
 AtomicReference<Node> tail;

 class Node {
  T item;
  AtomicReference<Node> next;
  Node(T item) {
   this.item = item;
   next = new
   AtomicReference
               <Node>(null);
  }
 }
```

```
public void put(T item) {
 Node node = new Node(item);
 while (true) {
   Node last = tail.get();
   Node next = last.next.get();
   if (last == tail.get()) {
     if (next == null) {
       if (last.next.
            compareAndSet(next, node)) {
         tail.compareAndSet(last, node);
         return;
       }
     } else {
         tail.compareAndSet(last,next);
     }
   }
 }
}
```

Example: STM unbounded buffer in (fantasy) STM-Java

```
class STMQueue<T> {
  Node sentinel = new Node(null);
  Node head = sentinel;
  Node tail = sentinel;

  class Node {
      T item;
      Node next;
      Node(T item) {
          this.item = item;
      }
  }
```

```
public void put(T item) {
  atomic {
    Node node = new Node(item);
    node.next = tail;
    tail = node;
  }
}

public T get() {
  atomic {
    if (head == tail) {
        retry;
    }
    T item = head.item;
    head = head.next;
    return item;
  }
}
```

Software Transactional Memory in Haskell

The *STM* library for the Glasgow Haskell Compiler (GHC) provides elegant high-level language support for STMs implemented by Simon Peyton Jones and others.

- Transactions are first-class values of *monadic* type STM a
- Transactions access shared memory in *transaction variables*, via `readTVar` and `writeTVar` operations.
- Transactions can block with `retry`
- Transactions can be freely composed with monadic sequencing, nested `atomically` blocks and `orElse` choices.

See Chapter 24 of Beautiful Code, edited by Greg Wilson, O'Reilly 2007.

Outline

Summary

Message Passing Concurrency with Actors

- Each actor has a *mail box*, which receives messages asynchronously.
- Actors sift through received messages by *pattern-matching*.
- Scala actors can be either *thread-based* or *event-based*. Thread-based actors block JVM threads when waiting; event-based actors use task management within a JVM thread to allow cheaper context switching.

Concurrency with Transactional Memory

- Transactions are sequences of operations committed atomically.
- Transactions can be aborted and retried.
- They can be composed elegantly and cleanly.
- STM implementations hide a lot of clever tricks.

Homework

- To prepare for the next lectures, familiarise/remind yourself of Haskell:

  http://blob.inf.ed.ac.uk/aplcourse/2010/02/haskell-resources/
  http://learnyouahaskell.com/

- In each of Scala (using actors) and Haskell (using STMs):
  - By rounding out the code fragments give, give complete implementations of unbounded and bounded queues and test them;
  - Try re-implementing your pigeon fancier program (or another example, e.g., the Dining Philosophers or Santa Claus).

References

TBC.