Outline

1 Opening

2 Actors

3 Software Transactional Memory

4 Summary
This is the third in a block of lectures looking at programming-language techniques for concurrent programs and concurrent architectures.

- Introduction, basic Java concurrency
- Concurrency abstractions
- Some other programming-language approaches to concurrency
- Cautionary tales in concurrency
Some challenges with locks

Locks provide a sound mechanism for enforcing separation and safely implementing shared-memory concurrency. They have many pitfalls, though, and can themselves introduce new difficulties. For example:

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

**Priority inversion** when a scheduler preempts a lower-priority thread that holds a lock needed for a higher-priority one (more on this in a later lecture)

**Lock Convoys / Thundering Herd** when many threads are waiting on a single lock;

**Lack of compositionality** as there is no straightforward way to build large thread-safe programs by composing small thread-safe programs.
Homework from Monday

1. Do this

Find out what the `addAndGet` method on a Java `AtomicLong` object does. Why is that useful? Java 8 introduced a `LongAdder` class. Find out what it does, and how it can make code faster.

2. Read this

The remaining sections of the Java Concurrency tutorial

http://java.sun.com/docs/books/tutorial/essential/concurrency

- Liveness
- Guarded Blocks
- Immutable Objects
- High Level Concurrency
Ways to program concurrency

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

Many different language paradigms have been proposed to manage this, including:

**Locks and conditions**: tasks share memory and use that to exclude and signal one another (e.g., Java);

**Actor-style message-passing**: a task offers a *mailbox* which receives messages (e.g. Scala/Akka, Erlang);

**Channel-based message-passing**: tasks send and receive messages over named *channels* (e.g. Go, Rust, Concurrent ML);

A key distinction here is between *shared-memory concurrency* like that from Java, and *share-nothing concurrency* where all co-operation is through message-passing.
Ways to program concurrency

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

Many different language paradigms have been proposed to manage this, including:

**Futures and promises:** computations are executed some time between when they are created and when required (e.g., Oz, E);

**Dataflow/datastream programming:** computations are described as a network of data dependencies (e.g., Lustre, Hume);

**Lock-free algorithms / transactional memory:** tasks share memory but detect and repair conflicts (e.g., libraries in C, C++, ...).

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

*Scala* is a strongly-typed object-oriented language that compiles to the Java Virtual Machine. This allows full interoperability with Java, both providing and using libraries and code. Scala draws extensively on features of functional programming languages, offering among other things:

- Pattern matching
- Algebraic datatypes (with *case classes*)
- Parameterized types
- Type inference
- Higher-order functions
- Lazy (call-by-need) evaluation

Scala is designed by Martin Odersky and his team at EPFL, Lausanne, Switzerland.

http://www.scala-lang.org
The Actor model

The *actor model* is used for concurrency in several programming languages.

- An *actor* is a process abstraction that interacts with other actors by *message passing*. Message sending is *asynchronous* (non-blocking).

- Each actor has a *mailbox* which buffers incoming messages until they are processed by *pattern matching*.

- Actors may create new actors, and can communicate the mailbox addresses of other actors.

- Actors have private variables, which can be updated in response to messages, but no actor can directly access the state of another.

Although an underlying implementation will use threads or processes to run actor code, this isn’t explicit in the model.

It’s usual to provide some support for *distribution* of actors, which is made easier by the lack of shared state.
Akka: Actors in Scala

Scala supports actor programming through the Akka toolkit (now also available for other JVM and .NET languages)

### Sending

```scala
// send message to
// another actor
actor ! message

// sender() is the actor
// who sent the last message
sender() ! message
```

### Receiving

```scala
// Pick up messages
// and act on them

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

Note: Actor syntax in Scala has changed over time. This shows Scala 2.12 and Akka 2.5
Example: Counter

case object Reset

case object Increment

case object Decrement

case object Value

class Counter extends Actor {

  var count = 0

  def receive = {
    case Reset => count = 0
    case Increment => count = count + 1
    case Decrement => count = count - 1
    case Value => sender() ! count
  }
}
Example: Bounded buffer

```scala
case class Put(x:String)
case object Get

class BoundedBuffer(size: Int) extends Actor {

  var buffer = new Array[String](size) \ Rotating buffer holding up to 'size' many strings
  var in, out, n = 0 \ Buffer input position, output position, current contents

  def receive = {
    case Put(x) if (n < size) => {
      buffer(in) = x
      in = (in + 1) % size
      n = n+1
    }

    case Get if (n > 0) => {
      val r = buffer(out)
      out = (out + 1) % size
      n = n-1
      sender() ! r
    }
  }
}
```
Supervision

As Scala/Akka actors create subsidiary actors and delegate work to them, these build up into a hierarchy of parent and child actors.

This hierarchy is used for supervision in Scala, where actors take responsibility for managing the other actors they have created. This is often taken to be a key component of the actor model.

When an actor fails in some way, its supervisor is notified and takes the action defined by its supervisorStrategy, chosen from:

- **Resume** — Just let the actor continue
- **Restart** — Re-initialise the actor state and start again
- **Stop** — Terminate and remove the actor
- **Escalate** — Signal the next level up in the hierarchy

There are also standard strategies to propagate these actions, for example:

- **OneForOneStrategy** — Apply action to just the failing actor
- **AllForOneStrategy** — Apply action to all supervised actors
Erlang

Scala’s actor concurrency is based closely on the *Erlang* system, developed by Ericsson for programming highly resilient and massively concurrent telecommunications equipment.

Ericsson AXD 301 multiservice 10–160Gbit/s switch

Nortel 8661 SSL Acceleration Ethernet Routing Switch
Outline

1 Opening

2 Actors

3 Software Transactional Memory

4 Summary
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.

- 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.
Using STM

Coding with transactional memory

Repeat
{
    Start transaction
    ...
    Read shared memory, navigate around data structures, perform computation, write shared memory, etc.
    ...
    End transaction
}
Until (transaction succeeds without interference)

- No lock is held during the transaction.
- Multiple threads may work over the same memory at the same time.
- Most of the time, all is well and this goes really fast.
How can that possibly work?

Software transactional memory is one of a range of *lock-free algorithms* and *lock-free datastructures* that seek to enable shared-memory concurrency while avoiding some of the difficulties caused by locks.

These lock-free algorithms rely on specialist atomic instructions:

- **TAS** Test-and-set (address)
- **CAS** Compare-and-swap (address,old,new)
- **LL/SC** Load-link (address) / Store conditional (address,value)

Each of these can emulate all the previous ones. Almost all current processors provide either CAS or LL/SC in hardware.

Lock-free algorithms use these instructions to make atomic changes to large datastructures, usually by switching pointers at carefully-chosen between carefully-chosen substructures.
Example implementation

Keir Fraser.

Practical Lock-Freedom.
http://www.cl.cam.ac.uk/research/srg/netos/lock-free/
Keir Fraser.

*Practical Lock-Freedom.*


http://www.cl.cam.ac.uk/research/srg/netos/lock-free/
Example STM library API

```c
stm* new_stm (int object_size) // Create and discard elastic STM
void free_stm (stm *mem) // region for objects of this size

(stm_obj*, void*) new_object(stm *mem) // Allocate and release individual
void free_object (stm *mem, stm obj *o) // objects in the given region

stm_tx* new_transaction (stm *mem) // Begin a transaction on this region

void* open_for_reading (stm_tx *t, stm_obj *o) // Obtain direct
void* open_for_writing (stm_tx *t, stm_obj *o) // access to object

bool commit_transaction (stm_tx *t) // Complete a transaction

void abort_transaction (stm_tx *t) // Bail out now
bool validate_transaction (stm_tx *t) // Check if already conflicted
```
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 be freely composed with monadic sequencing, nested *atomically* blocks and `orElse` choices.

Summary

Message-Passing Concurrency with Actors

- Each actor has a *mailbox*, which receives messages asynchronously.
- Actors respond to received messages by *pattern-matching*.
- A *supervision hierarchy* makes sure that a system keeps going in the event of local or temporary failure.

Concurrency with Transactional Memory

- Transactions are sequences of operations committed atomically.
- Collisions are detected, transactions can be aborted and retried.
- They compose elegantly and cleanly.
- STM implementations hide a lot of clever tricks.
1. Homework: Read this

Channel-based message-passing concurrency

The Go language provides concurrent goroutines and named channels for communication between them. Read these two articles about this.

- An Introduction to Programming in Go: Concurrency
  https://www.golang-book.com/books/intro/10
- Visualizing Concurrency in Go
  https://divan.github.io/posts/go_concurrency_visualize

Optional Extras

Watch https://blog.golang.org/concurrency-is-not-parallelism
  and https://is.gd/goconcurrencypatterns

Read A Tour of Go: Goroutines  https://tour.golang.org/concurrency
  and Go by Example: Goroutines  https://gobyexample.com/goroutines
2. Homework: Do this

Who uses Erlang? What do they do with it?

- Read up about Erlang, its use of actors, supervision, and live code replacement.
- Find an example of some software / a system / a service / a company that uses the language and where someone has written about that.
- Post the example to the mailing list or Piazza.

For example:

Demonware: Erlang and First-Person Shooters
10s of millions of Call of Duty Black Ops fans loadtest Erlang

Presentation to Erlang Factory, London 2011

http://is.gd/erlangfps
GRiSP: DIVE INTO A NEW EXPERIENCE OF BUILDING WIRELESS EMBEDDED SYSTEMS

Create amazing Internet of Things designs without soldering or dropping down to C. Right out of the box, GRiSP-Base boots into Erlang VM running on real bare metal. It features on-board wireless networking 802.11b/g/n WLAN and connectors for standard PMOD sensor and actuator modules.