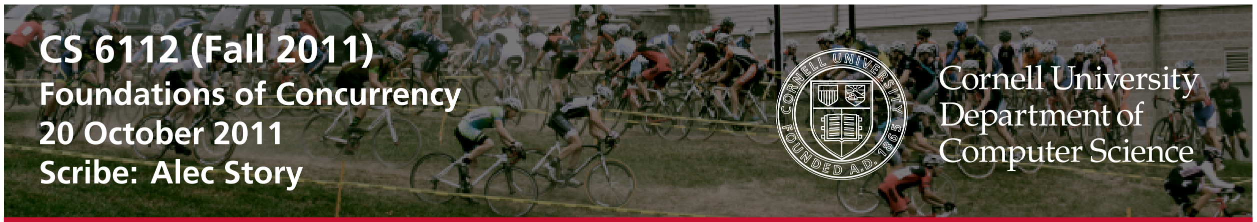

CS 6112 (Fall 2011)
Foundations of Concurrency
20 October 2011
Scribe: Alec Story

Cornell University
Department of
Computer Science

Actors are a fairly old idea for concurrency, but a lot of the old work is confusing and hard to read.

Actors have mailboxes, and can send messages, change state (sort of like recursion in CCS), and spawn new actors, with their own mailboxes.

1 Scala Actors

Threads versus events:

threads	events
block for input	register
control own flow	dependent on event loop and continuations
more natural to program in	inversion of control
context switching, state and locking	less expensive; just a mailbox
	messages are grabbed out of mailbox arbitrarily

The paper proposes a duality between threads and events, mapping thread actions (fork, join, wait) onto event actions (start, end, continue). The idea being that you don't need to re-write your program to switch between the two concurrency implementations. Their transformations leave some room for improvement, though; send-receive on the event side becomes a substantial amount of fork, block and join on the thread side.

An example of the Scala code:

```
val orderMgr = actor {
  while (true) receive{
    case Order(s, item) =>
      val o = handleOrder(s, item)
      s ! Ack(.)
    case Cancel(s, o) =>
      if (o.pending) {
        cancel(o)
        s ! Ack(.)
      } else {
        s! NoAck
      } case x => junk += x
  }
}

val customer = actor {
  orderMgr ! order(self, item)
  receive {
    case Order o => ...
  }
}
```

The Scala actors library includes both threads and events, mixing them together makes programs somewhat easier to write.

So, orderMgr will run until it blocks on receive, the runtime will see everyone blocked and add threads, then the customer sends to the orderMgr's mailbox, the orderMgr unblocks, and so on.

Another interesting thing is the `!?` construct, which sends a message with the sender's address, and waits for a reply, which is quite convenient.

Example of a bounded buffer:

```
class Buffer (N: Int) extends Actor {
  val buf = new Array [Int] (N)
  var in = 0; var out = 0; var n = 0
  def reaction : PartialFunction[Any, Unit] = {
    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)
  }
  def act(): Unit = while true receive(reaction)
}
```

Matches that fail silently consume the message, dropping it.

This formulation integrates pretty well with Scala's type system, and can be extended easily, for example to fetch two items at once:

```
class Buffer2 (N: Int) extends Buffer {
  override reaction : PartialFunction = {
    super.reaction orElse {
      Get2 if n > 1 =>
        ...
    }
  }
}
```

orElse is a special keyword to help with this sort of overriding.

Some other definitions for reference:

```
trait Actor {
  val mailbox = new Queue[Any]
  def ! (msg : Arg) = Unit ...
  def receive [R] (f: PartialFunction[Any, R]): R
}

abstract class Function1[-a, +b] { // - indicates contravariant
  def apply (x : a) : b            // + indicates covariant
}

abstract class PartialFunction[-a, +b] extends Function1[a, b] {
  def isDefined (x:a): Boolean
}
```

The idea here is that functions that may not be defined over the entire domain are partial functions.

Mailboxes may be a queue, but the semantics given only guarantee a bag.

The paper unifies events and threads through the react construct, which *doesn't return anything*:

```
def react(f: PartialFunction[Any, Unit]): Nothing =
  synchronized {
    mailbox.dequeueFirst(f.isDefinedAt) match {
      case Some(msg) =>
        schedule(new Task({ () => f(msg) }))
      case None =>
        continuation = f
        isDetached = true
        waitingFor = f.isDefinedAt
  }
  throw new SuspendActorException
}
```

This transforms the partial function f into a full continuation. The special exception thrown by react is caught by the Task running it, which acts as a control transfer to suspend execution:

```
class Task(cont: () => Unit) {
  def run() {
    try { cont() } // invoke continuation
    catch { case _: SuspendActorException =>
      // do nothing }
  }
}
```

This is in particular used when there are no more messages to read.

Useful combinators for react:

`andThen` hooks up continuations to invoke another

`loop(body)` becomes `body andThen loop(body)`

Pipes, figures 6 and 7 in the paper follow. They all implement more-or-less the same behavior: either write some bytes to a pipe or read from it, then switch roles:

Thread-based:

```
class Proc(write: Boolean, exch: Barrier) extends Thread {
  ...
  override def run() {
    if (write) writeData
    else readData
    exch.await
    if (write) readData
    else writeData
  }
}

def writeData {
  fill(buf)
  disp.register(sink, writeHnd)
  var finished = false
  while (!finished) {
    dataReady.await
    dataReady.reset
    if (bytesWritten == 32*1024)
      finished = true
    else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink, writeHnd)
    }
  }
}

val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten += sink.write(buf)
    dataReady.await
  }
}
```

We use buffers to creat non-blocking IO, register the barrier asynchronously, and then, in a loop, fill the buffer, and wait for the writeHandler to empty.

Event-based:

```
class Proc(write: Boolean, pool: Executor) {
  ...
  var last = false
  if (write) writeData
  else readData
  ...
  def writeData {
    fill(buf)
    disp.register(...)
  }
}

val task = new Runnable {
  def run() {
    if (bytesWritten == 32*1024) {
      if (!last) {
        last = true; readData
      }
    } else {
      if (!buf.hasRemaining)
        fill(buf)
      disp.register(sink, writeHnd)
    }
  }
}

val writeHnd = new WriteHandler {
  def handleWrite() {
    bytesWritten += sink.write(buf)
    pool.execute(task)
  }
}
```

Actor-based:

```
class Proc(write: Boolean, other: Actor) extends Actor)
  ...
  def act() {
    { if (write)
        writeData
      else
        readData
    } andThen {
      other ! Exchange
      react {
        case Exchange =>
          if (write)
            readData
          else
            writeData }
    }
  }
}
```

constructs, but are easier to reason about and come up with than the event-based version, which was the goal of this design.