CONCURRENCY AND THE ACTOR MODEL

© University of Liverpool

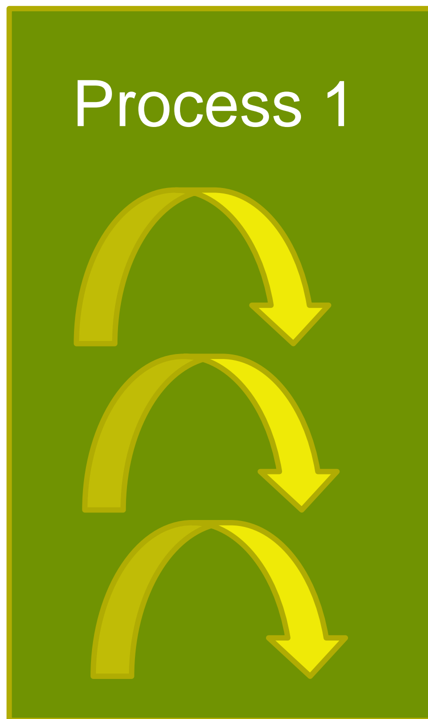
Concurrency review

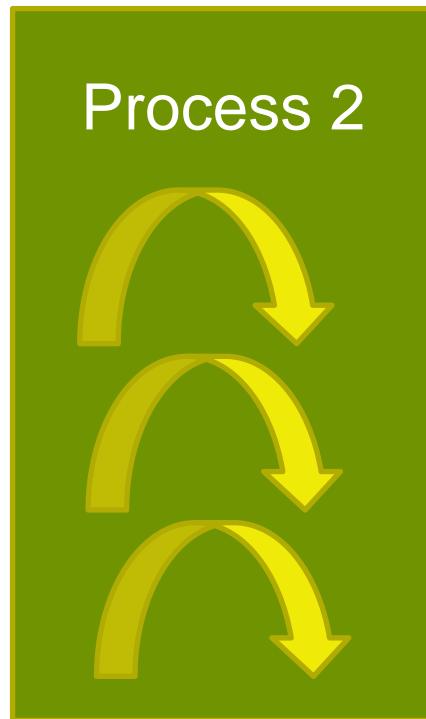
- Multi-tasking
  - Task
    - Processes with isolated address space
    - Programming examples
      - Unix fork() processes
      - IPC via
        - Pipes, FIFOs
        - Shared memory
      - Fork() returns
        - -1 for error
        - 0 for child
        - >0 for parent (pid of parent)

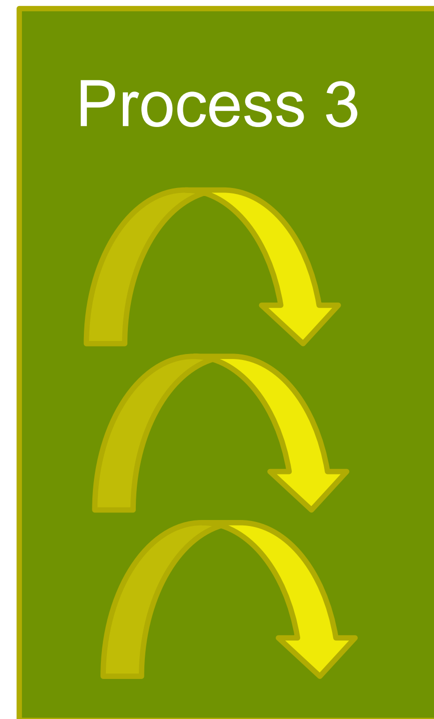
Unix fork example

```
main()
{
   pid_t pID = fork();
   if (pID == 0)                // child
   {
      // Code only executed by child process
         else if (pID < 0)    // fork failed
    {
          exit(1);
     }
else                                   // parent
    {
// Code only executed by parent process
}
```

Threads and processes







Standard concurrency in Java review

- Threads
  - Shared memory, flow of execution
  - Uses Thread or Runnable
- Mutex
  - Monitors
- Thread synchronization
  - wait
  - notify

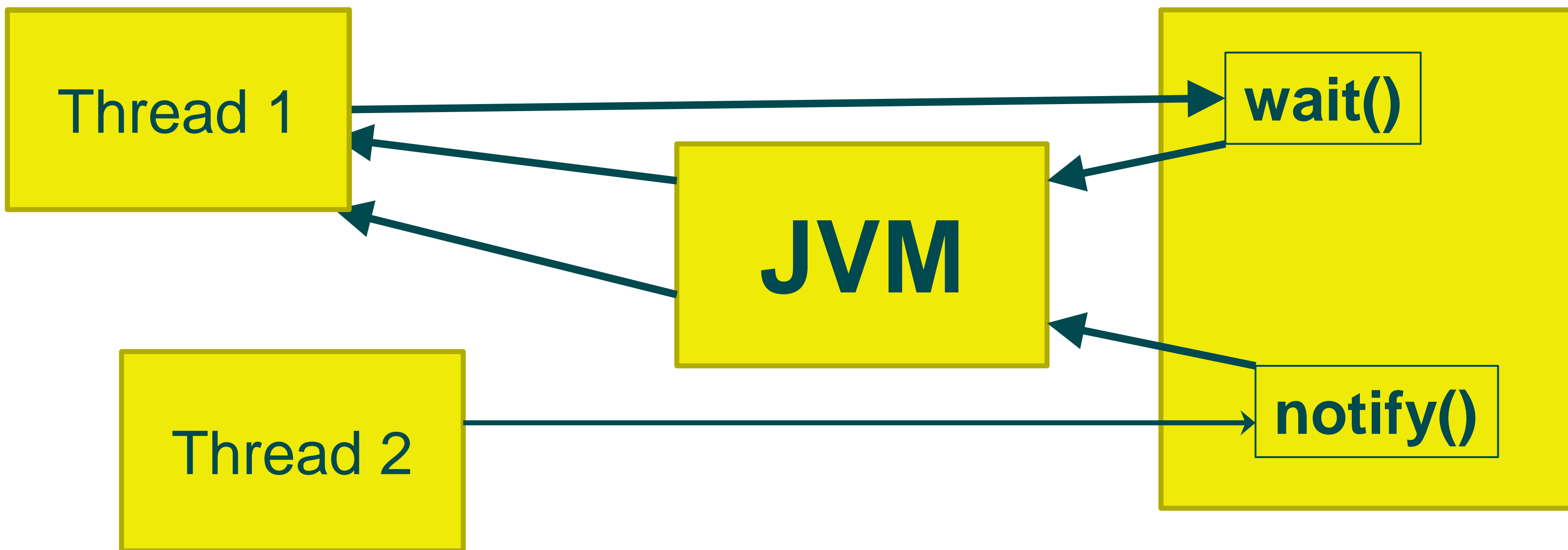
© University of Liverpool

Green Threads and threads

- Green threads
  - Scheduled and switched by the JVM
- Native threads
  - Supported by the OS
  - Scheduling depends on OS
- When we look at Actor languages concurrency the difference is important

© University of Liverpool

Concurrency and deadlock



Deadlock (circular dependency)

Code thread 1
object1.wait()
object2.signal()

→ **object1** **wait()**

Code thread 2
object2.wait()
object3.signal()

→ **object2** **wait()**

Code thread 3
object3.wait()
object1.signal()

→ **object1** **wait()**

Synchronization deadlock

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          int amountToTransfer) {
  synchronized (fromAccount) {
    synchronized (toAccount) {
      if
(fromAccount.hasSufficientBalance(amountToTrans
fer) {
        fromAccount.debit(amountToTransfer);
        toAccount.credit(amountToTransfer);
      }
    }
  }
}
```

```
transferMoney(accountOne, accountTwo, amount);
```

```
transferMoney(accountTwo, accountOne, amount);
```

Double lock issue

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {  //1
      if (instance == null)          //2
        instance = new Singleton();  //3
    }
  }
  return instance;
}
```

Deadlock as race condition errors

- Very hard to test for…

```
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          int amountToTransfer) {
  synchronized (fromAccount) {

    synchronized (toAccount) {
      if
(fromAccount.hasSufficientBalance(amountToTrans
fer) {
        fromAccount.debit(amountToTransfer);
        toAccount.credit(amountToTransfer);
      }
    }
```

Fairness

- Multiple threads waiting for
  - Synchronization locks
  - Notifications
- Java does not guarantee
  - To wake up threads dependent in waiting time
  - Not to starve threads out

Summary

- Thread shared models are prone to
  - Deadlocks
  - Unfairness
  - Data corruption issues
  - Hard to test code (race conditions)
- Notice this is
  - Inherent in the model
  - Must be dealt with by careful design decisions

Current Trends

- Powerful multi-core architecture
- Large amounts of memory
- High speed networks
- Easy to support
  - Many threads
  - Fast message
- Hard to design
  - Complex concurrent systems
- Hardware is powerful/software is complex

© University of Liverpool

Actor model

- Much simplified approach to concurrency
- Each actor
  - Receives messages in an inbox
  - Sends messages to other Actors
- The actor contains state BUT
  - State is not shared between actors
  - Messages sent are immutable and owned by the receiver

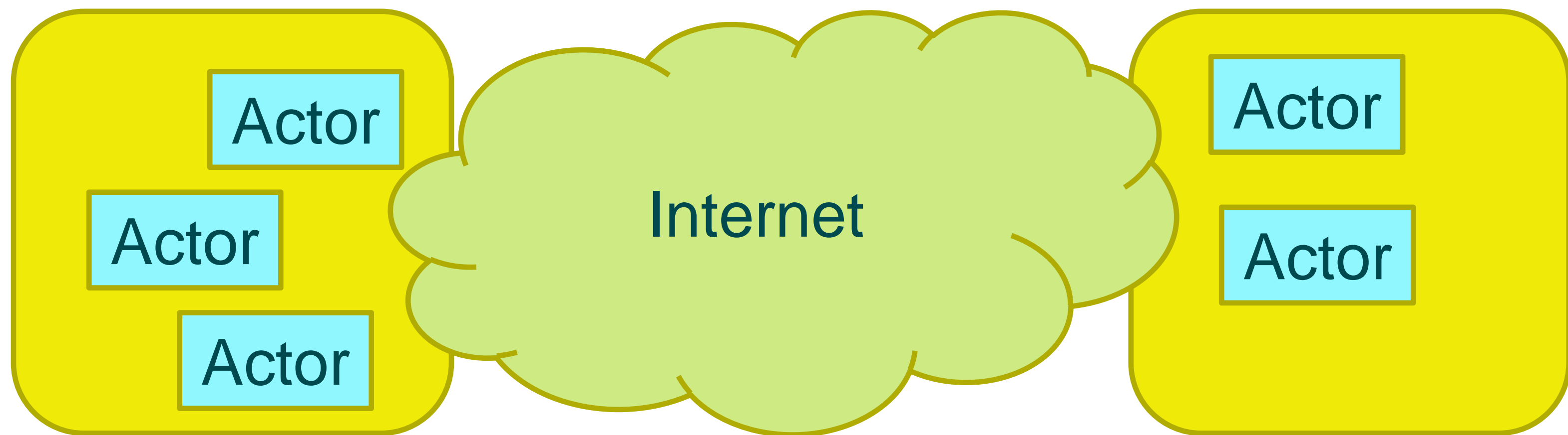
© University of Liverpool

Actor model

- Each Actor has an independent thread
- Each Actor's thread processes messages out of the inbox
- There is no need to share data and therefore no need for deadlock
- The Actor can
  - Create new Actors
  - Send message to other Actors

Actors and Networks

- Actors don’t have to on the same system
- This helps with scaling/redundancy etc.



Java and state sharing

- If objects are not shared then call by reference is not allowed
- Message myMess=new Message(“Hello!”);
- myActor.sendMessage(myMess);   // shared state

- // Replace with call by value..
- Message myMess=new Message(“Hello!”);

myActor.sendMessage(myMess.clone());

// N.B. clone is garbage collected after call terminates

© University of Liverpool

Safe Messaging

- Some Java Actor frameworks allow
  - Access to other Actors mailboxes
  - Memory sharing via messages
- Solutions
  - Cloning
  - Immutable objects (e.g. Strings)
  - Linear types
    - pointerA=pointerB   (pointerB now invalid)

Messaging handling issues

- Zero copy
  - Pass by reference
  - Very fast
  - Keep object as immutable
  - Use scalar types
  - Not useful with remote actors
- Full copy (deep copy)
  - Can be very slow (often the bottle neck for the application)

Message handling

- All messages are sent asynchronously
  - Non-blocking mode
- Message do not have to be buffered
- Messages do not have to arrive in order they were sent (datagram support)
- Addressing
  - Actor's address (mailing address)
  - Addresses can come from
    - Message received
    - Know since this Actor created the Actor

© University of Liverpool

Scheduling and fair-scheduling

- The choice of which actor gets to execute next and for how long is done by a part of the system called the scheduler
- An actor is non-blocked if it is processing a message or if its mailbox is not empty, otherwise the actor is blocked
- A scheduler is fair if it does not starve a non-blocked actor, i.e. all nonblocked actors eventually execute
- Fair scheduling makes it easier to reason about programs and program composition
- Otherwise some correct program (in isolation) may never get processing time when composed with other programs

Message fairness

- All messages are assumed to be delivered
- All messages are assumed to be processed (eventually)
- This is to help avoid deadlock and starvation

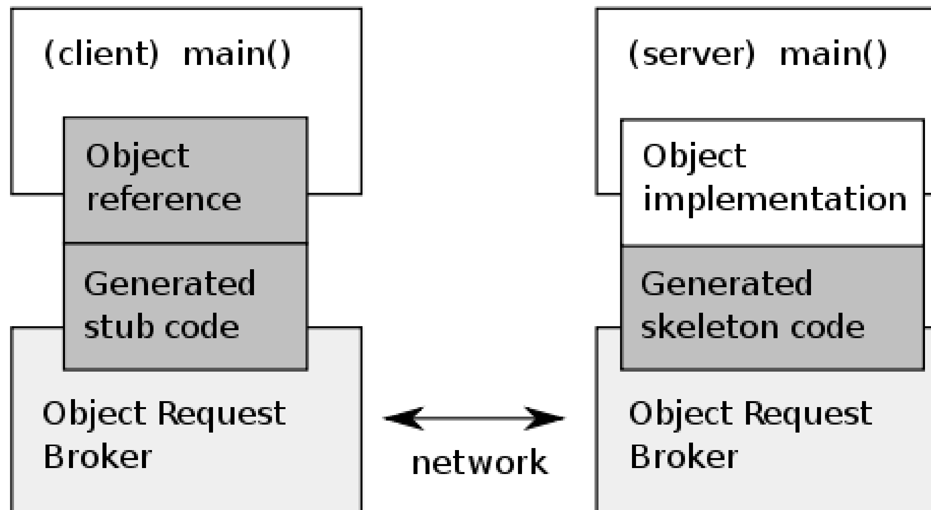
© University of Liverpool

Location Transparency

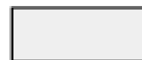
- Addressing is the same wherever
  - The sender is (local or remote)
  - The receiver is (local or remote)
- Location Transparency examples
  - Mobile MISDN
  - Object Request Broker (CORBA)
- Useful technology
  - Brokers, Anycasting and multicasting

© University of Liverpool

Object Request Broker



Key:

 ORB vendor-supplied code

ORB vendor-tool generated code

User-defined application code

© University of Liverpool

Locality of reference

- Bring Actors closer together
  - Possibly on the same machine
- Why
  - Cut down bandwidth, failures and latency (delays)
- How
  - Process migration
  - Pick closer Actors first
- Contradicts?
  - Location transparancy

Transparent Migration

- Movement of Actors
- Features
  - Capture state
  - Creation of new Actor
  - Deletion of old Actor
  - Restoring state
  - Handling addressing
- All this should be transparent to
  - All application code

© University of Liverpool

Mobility

- Strong or Weak (Fuggetta et al)
- Strong
  - Movement of code + execution state
- Weak
  - Movement of code + optional initialized state
  - Can be used to move idle actor

Mobility uses

- Locality of reference
  - Moving Actors closer
- Redundancy
  - Fail over to other hardware
- Load balancing
  - Moving Actor to less loaded component
- Re-configuration
  - Moving to new hardware
  - Moving to mobile client

© University of Liverpool

Synchronization and Actors

- Actor sends messages, with return address
- Senders
  - Monitors for mailbox for reply
- Standard service in
  - Scala
  - Actor Foundry
  - many more..

© University of Liverpool

Synchronisation constraints

- Examples
  - Buffer that fills up
  - Service is offline at certain hours
- Solution
  - Defer the processing of the message
  - Store new message in saved message queue
  - Every time new message received, checked saved queue

Synchronisation constraints

- Scalar Example

```
@Disable(messageName = "put")
public Boolean disablePut(Integer x) {
        if (bufferReady) {
                return (tail == bufferSize);
        }
        else {
          return true;
        }
}
```

Actor languages and frameworks

- Actor foundry
  - Safe (by-copy) as well as efficient (zero-copy) messaging
  - Message ordering using Local Synchronization Constraints
  - Pattern-matching (Multiple Dispatch)
  - Fair scheduling
  - Mobility
  - Location independence

© University of Liverpool

Erlang

- Ericsson in house language
- Functional approach
- Very high performance
- Hot swap module handling
- Message passing
- Location independence
- No classes, no OO

Erlang and variables and FP

- Not really variables
  - A=5
    - if A is unbounded 5 assigned to A
  - B=A
    - Ok
  - A=10
    - ERROR   A is already bound to 5
  - A=5
    - Just returns 5

© University of Liverpool

Referential transparency

- Every time you call a function with the same arguments
  - You get the same return value
- Feature of Erlang and other FP languages
- Gives higher levels of code stability due to state not being used in functions

Tail recursion

- Allows you to call a recursively call a method without pushing anything on to the stack

```
function a(state) {

a(new_state);        // does not let
                     // stack grow, like a loop
}
```

Scala

- Function + OOP
- Integrated with Java JDK
  - Call any JDK method from scala
- Full support for Actor programming
- Highly scalable
- Dynamic Typing
- Lots of inference
  - Example   var message=“Hello”
  - No need for String type definition

© University of Liverpool

Scala Hello World

```
object HelloWorld {
  def main(args: Array[String]): Unit= {
    var hello="Hello World...";
    val myValue="Freddy";
    println(hello+myValue);
  }
}
```

No need for static, object defines singleton class instance

Scala standard class definition

```
class Person(val surname: String,val forename: String)
{
    var weight=0f;      // weight of person
    def fn:  String  =  forename + surname
    def setWeight(w : Float) : Unit = {
          this.weight=w;
    }
    def getWeight(metric: Boolean) : Double = {
        if (metric)
        {
             weight            // returns value
        }
        else {
          var imperial=weight*2.2
          imperial             // returns value
        }
    }
}
```

Extra constructors

```
def this(surname: String ) = {
        this(surname,null)
        this
    }
```

Actor example

```
class EmailMessage(val message: String)
{


}
```

Actor Example

```
import scala.actors.Actor
import scala.actors.Actor._

class EmailClient(val username: String) extends
Actor {
  def act() {
    while (true) {
      receive {
        case incoming: EmailMessage =>
          println("Got message for " + username +
"message is " + incoming.message);
      }
    }

  }
}
```

Actor Example

```
import scala.actors.Actor
class Mailbox(userclient: Actor) extends Actor {
  def act()  {                    // actor thread
        val a=1;
        val b=1;
        var incoming=new EmailMessage("Hello");
        while (true) {
          receive {
                case incoming : EmailMessage =>
                        println("Got message now
sending"+incoming.message);
                        userclient ! incoming


          }
        }
  }
}
```

Actor Example

```
object Main {

  def main(args: Array[String]) : Unit = {
       val client=new EmailClient("Seb");
      val mb=new Mailbox(client);
      mb.start();
      client.start();
      mb ! new EmailMessage("Hello how are
you?");

  }
```

Summary

- Actors can
  - Remove dead lock
  - Improve
    - Scalability, redundancy
  - Allow for full mobility of code
  - Be local or remote
  - Create new actors

© University of Liverpool