Concurrent Models of Computation

Edward A. Lee

Robert S. Pepper Distinguished Professor, UC Berkeley

EECS 219D

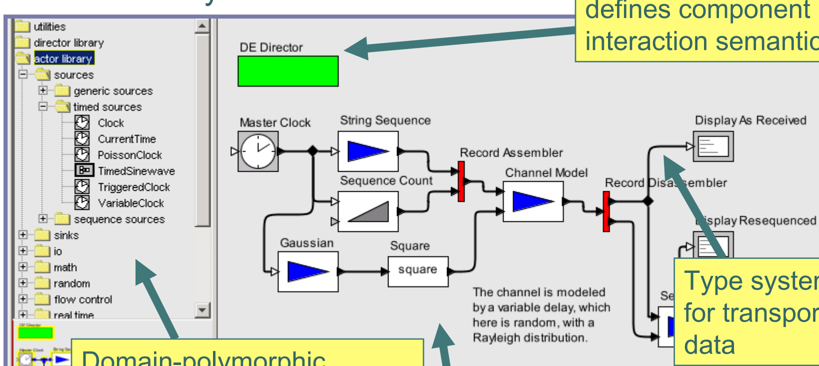
Concurrent Models of Computation

Fall 2011

Week 5: Threads

Ptolemy II: Framework for Experimenting with Alternative Concurrent Models of Computation

Basic Ptolemy II infrastructure:



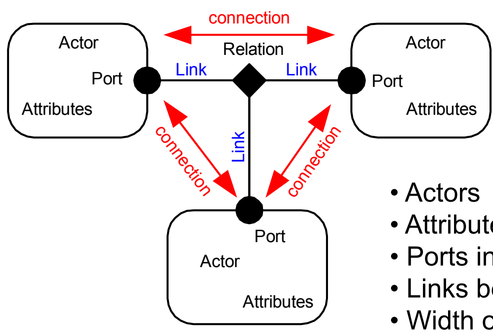
Director from a library defines component interaction semantics

Type system for transported data

Domain-polymorphic component library.

Visual editor supporting an abstract syntax

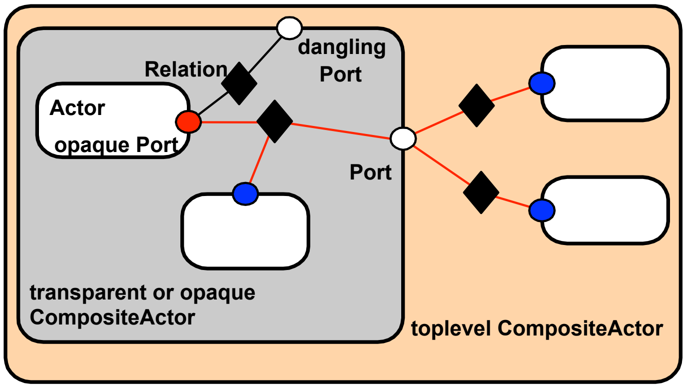
The Basic Abstract Syntax



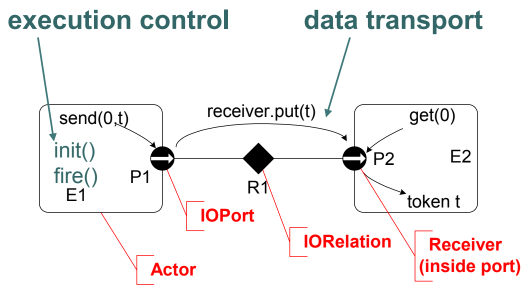
- Actors
- Attributes on actors (parameters)
- Ports in actors
- Links between ports
- Width on links (channels)
- Hierarchy

Concrete syntaxes:
- XML
- Visual pictures
- Actor languages (Cal, StreamIT, …)

Hierarchy - Composite Components



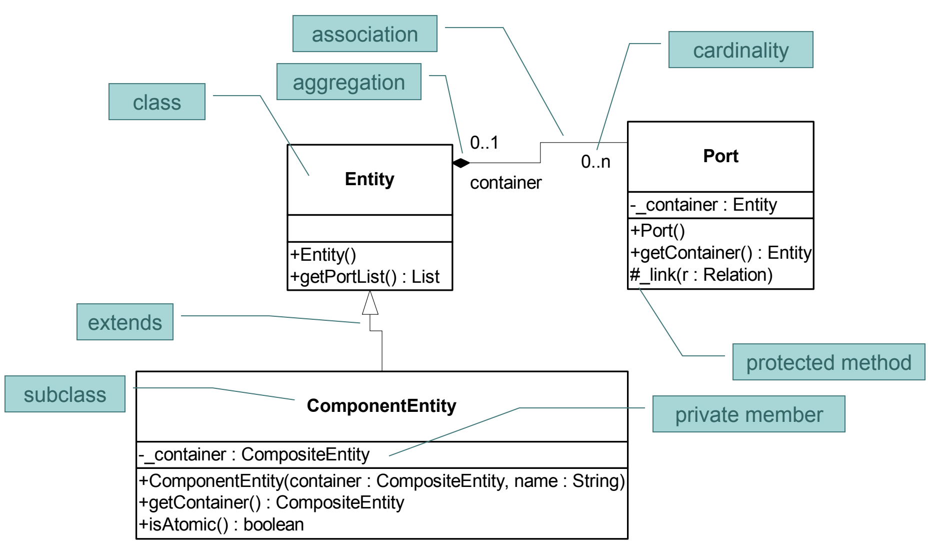
Abstract Semantics
of *Actor-Oriented* Models of Computation



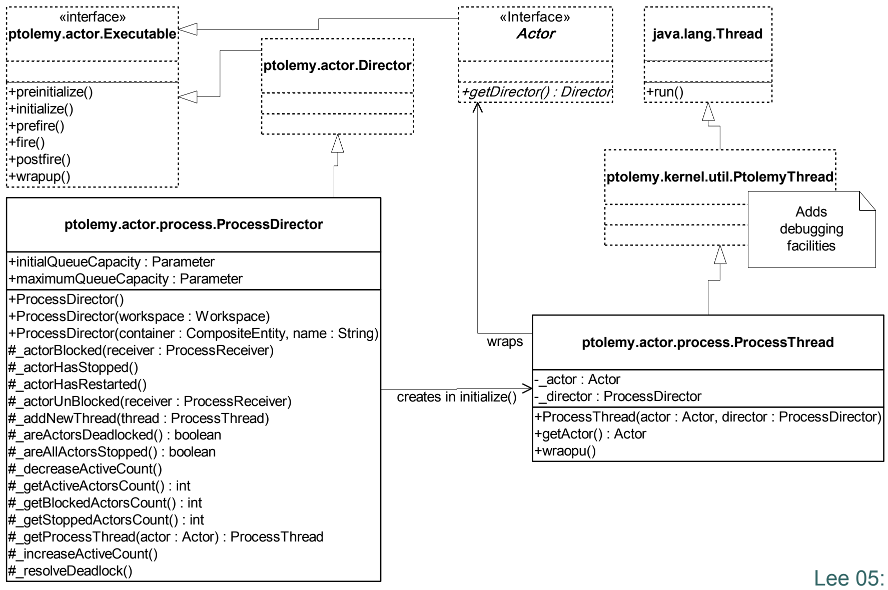
Actor-Oriented Models of Computation that we have implemented:

- dataflow (several variants)
- process networks
- distributed process networks
- Click (push/pull)
- continuous-time
- CSP (rendezvous)
- discrete events
- distributed discrete events
- synchronous/reactive
- time-driven (several variants)
- …

Notation: UML Static Structure Diagrams



Instance of ProcessThread Wraps Every Actor



ProcessThread Implementation (Outline)

```
_director._increaseActiveCount();
try {
    _actor.initialize();
    boolean iterate = true;
    while (iterate) {
        if (_actor.prefire()) {
            _actor.fire();
            iterate = _actor.postfire();
        }
    }
} finally {
    try {
        wrapup();
    } finally {
        _director._decreaseActiveCount();
    }
}
```

Subtleties:

- The threads may never terminate on their own (a common situation).
- The model may deadlock (all active actors are waiting for input data)
- Execution may be paused by pushing the pause button.
- An actor may be deleted while it is executing.
- Any actor method may throw an exception.
- Buffers may grow without bound.

Typical fire() Method of an Actor

```
/** Compute the absolute value of the input.
 *  If there is no input, then produce no output.
 *  @exception IllegalActionException If there is
 *   no director.
 */
public void fire() throws IllegalActionException {
    if (input.hasToken(0)) {
        ScalarToken in = (ScalarToken)input.get(0);
        output.send(0, in.absolute());
    }
}
```

The get() method is behaviorally polymorphic: what it does depends on the director.

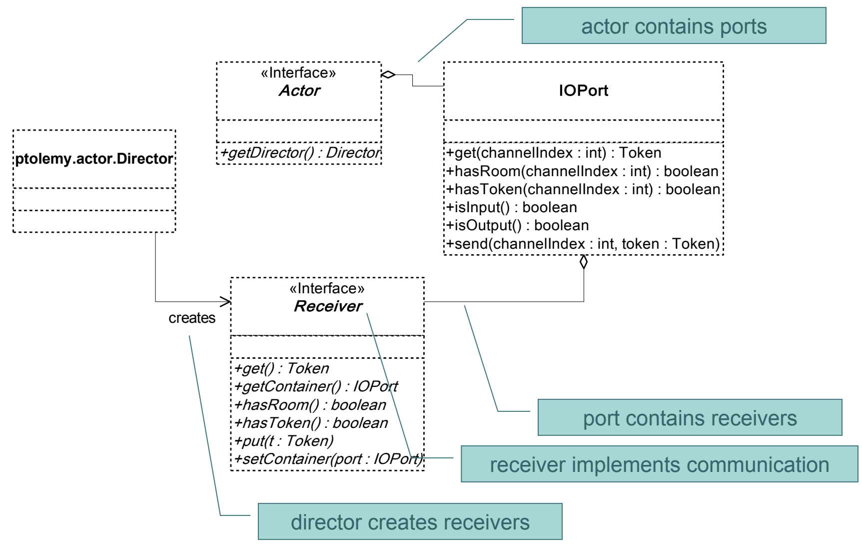
In PN, hasToken() always returns *true*, and the get() method blocks if there is no data.

Sketch of get() and send() Methods of IOPort

```
public Token get(int channelIndex) {
   Receiver[] localReceivers = getReceivers();
   return localReceivers[channelIndex].get();
}

public void send(int channelIndex, Token token) {
   Receiver[] farReceivers = getRemoteReceivers();
   farReceivers[channelIndex].put(token);
}
```

Ports and Receivers



Process Networks Receiver Outline

```
public class PNQueueReceiver extends QueueReceiver
       implements ProcessReceiver {

    private boolean _readBlocked;

    public boolean hasToken() {
        return true;
    }

    public synchronized Token get() {
        ...
    }

    public synchronized void put(Token token) {
        ...
    }
}
```

flag indicating whether the consumer thread is blocked.

always indicate that a token is available

acquire a lock on the receiver before executing put() or get()

get() Method (Simplified)

```
public synchronized Token get() {
    PNDirector director = ... get director ...;
    while (!super.hasToken()) {
        _readBlocked = true;
        director._actorBlocked(this);
        while (_readBlocked) {
            try {
                wait();
            } catch (InterruptedException e) {
                throw new TerminateProcessException("");
            }
        }
    }
    return result = super.get();
}
```

super class returns true only if there is a token in the queue

notify the director that the consumer thread is blocked

release the lock on the receiver and stall the thread

use this exception to stop execution of the actor thread

super class returns the first token in the queue.

Lee 05: 13

put() Method (Simplified)

```
public synchronized void put(Token token) {
    PNDirector director = ... get director ...;
    super.put(token);
    if (_readBlocked) {
        director._actorUnBlocked(this);
        _readBlocked = false;
        notifyAll();
    }
}
```

notify the director that the consumer thread unblocks.

wake up all threads that are blocked on wait().

Lee 05: 14

Subtleties

- Director must be able to detect deadlock.
  - It keeps track of blocked threads
- Stopping execution is tricky
  - When to stop a thread?
  - How to stop a thread?
- Non-blocking writes are problematic in practice
  - Unbounded memory usage
  - Use Parks' strategy:
    - Bound the buffers
    - Block on writes when buffer is full
    - On deadlock, increase buffers sizes for actors blocked on writes
    - Provably executes in bounded memory if that is possible (subtle).

Stopping Threads

"Why is Thread.stop deprecated?

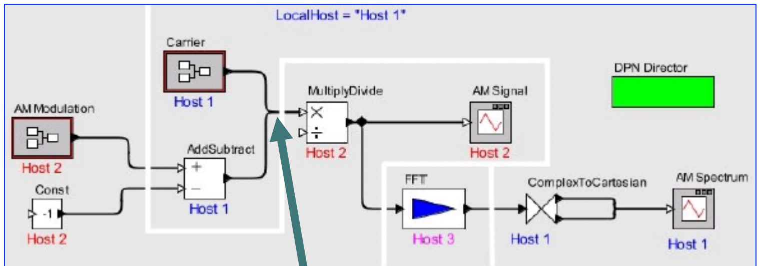
Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be *damaged*. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future."

Java JDK 1.4 documentation.

Thread.suspend() and resume() are similarly deprecated.

Thread.destroy() is unimplemented.

Distributed Process Networks



Transport mechanism between hosts is provided by the director (via receivers). Transparently provides guaranteed delivery and ordered messages.

Created by Dominique Ragot, Thales Communications

Threads

Threads dominate concurrent software.

- *Threads*: Sequential computation with shared memory.
- *Interrupts*: Threads started by the hardware.

Incomprehensible interactions between threads are the sources of many problems:

- Deadlock
- Priority inversion
- Scheduling anomalies
- Timing variability
- Nondeterminism
- Buffer overruns
- System crashes

My Claim

Nontrivial software written with threads is incomprehensible to humans. It cannot deliver repeatable and predictable timing, except in trivial cases.

Consider a Simple Example

"The *Observer pattern* defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically."

Design Patterns, Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN: 0201633612):

Observer Pattern in Java

```
public void addListener(listener) {…}

public void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Will this work in a multithreaded context?

Thanks to Mark S. Miller for the details of this example.

Observer Pattern With Mutual Exclusion (Mutexes)

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

Javasoft recommends against this. What's wrong with it?

Mutexes are Minefields

```
public synchronized void addListener(listener) {…}

public synchronized void setValue(newValue) {
    myValue = newValue;

    for (int i = 0; i < myListeners.length; i++) {
        myListeners[i].valueChanged(newValue)
    }
}
```

valueChanged() may attempt to acquire a lock on some other object and stall. If the holder of that lock calls addListener(), deadlock!

Lee 05: 23

After years of use without problems, a Ptolemy Project code review found code that was not thread safe. It was fixed in this way. Three days later, a user in Germany reported a deadlock that had not shown up in the test suite.

Simple Observer Pattern Becomes Not So Simple

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make copy of listeners to avoid race conditions

notify each listener <u>outside</u> of synchronized block to avoid deadlock

This still isn't right.
What's wrong with it?

Lee 05: 25

Simple Observer Pattern: How to Make It Right?

```
public synchronized void addListener(listener) {…}

public void setValue(newValue) {
    synchronized(this) {
        myValue = newValue;
        listeners = myListeners.clone();
    }

    for (int i = 0; i < listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value changes in the wrong order!

Lee 05: 26

If the simplest design patterns yield such problems, what about non-trivial designs?

```
/**
CrossRefList is a list that maintains pointers to other CrossRefLists.
…
@author Geroncio Galicia, Contributor: Edward A. Lee
@version $Id: CrossRefList.java,v 1.78 2004/04/29 14:50:00 eal Exp $
@since Ptolemy II 0.2
@Pt.ProposedRating Green (eal)
@Pt.AcceptedRating Green (bart)
*/
public final class CrossRefList implements Serializable  {
    …
    protected class CrossRef implements Serializable{
        …
        // NOTE: It is essential that this method not be
        // synchronized, since it is called by _farContainer(),
        // which is.  Having it synchronized can lead to
        // deadlock.  Fortunately, it is an atomic action,
        // so it need not be synchronized.
        private Object _nearContainer() {
            return _container;
        }

        private synchronized Object _farContainer() {
            if (_far != null) return _far._nearContainer();
            else return null;
        }
        …
    }
}
```

Code that had been in use for four years, central to Ptolemy II, with an extensive test suite with 100% code coverage, design reviewed to yellow, then code reviewed to green in 2000, causes a deadlock during a demo on April 26, 2004.

What it Feels Like to Use the *synchronized* Keyword in Java

Image "borrowed" from an Iomega advertisement for Y2K software and disk drives, Scientific American, September 1999.

Perhaps Concurrency is Just Hard…

Sutter and Larus observe:

> *"humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."*

H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7), 2005.

Lee 05: 29

Is Concurrency Hard?

It is not concurrency that is hard…

Lee 05: 30

…It is Threads that are Hard!

Threads are sequential processes that share memory. From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept).

Imagine if the physical world did that…

Lee 05: 31

Succinct Problem Statement

Threads are wildly nondeterministic.

The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design).

Lee 05: 32

We Can Incrementally Improve Threads

Object Oriented programming
Coding rules (Acquire locks in the same order…)
Libraries (Stapl, Java 5.0, …)
Patterns (MapReduce, …)
Transactions (Databases, …)
Formal verification (Blast, thread checkers, …)
Enhanced languages (Split-C, Cilk, Guava, …)
Enhanced mechanisms (Promises, futures, …)

But is it enough to refine a mechanism with flawed foundations?

The Result: Brittle Designs

Small changes have big consequences…

Patrick Lardieri, *Lockheed Martin ATL*, about a vehicle management system in the JSF program:

> "Changing the instruction memory layout of the Flight Control Systems Control Law process to optimize 'Built in Test' processing led to an unexpected performance change - System went from meeting real-time requirements to missing most deadlines due to a change that was expected to have no impact on system performance."

National Workshop on High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS) Arlington, VA November 30 – December 1, 2006

For a brief optimistic instant, *transactions* looked like they might save us…

"TM is not as easy as it looks (even to explain)"

Michael L. Scott, invited keynote, (EC)2 Workshop, Princeton, NJ, July 2008

So, the answer must be message passing, right?

Not quite…

More discipline is needed that what is provided by today's message passing libraries.

A Model of Threads

Binary digits: $B = \{0, 1\}$
State space: B^*
Instruction (atomic action): $a : B^* \to B^*$
Instruction (action) set: $A \subset [B^* \to B^*]$
Thread (non-terminating): $t : N \to A$
Thread (terminating): $t : \{0, \dots, n\} \to A, \quad n \in N$

A thread is a sequence of atomic actions, a member of A^{**}

Programs

A program is a finite representation of a family of threads (one for each initial state b_0).
Machine control flow: $c : B^* \to N$ (e.g. program counter) where $c(b) = 0$ is interpreted as a "stop" command.

Let m be the program length. Then a program is:

$$p : \{1, \dots, m\} \to A$$

A program is an ordered sequence of m instructions, a member of A^*

Execution (Operational Semantics)

Given initial state $b_0 \in B^*$, then execution is:

$$b_1 = p(c(b_0))(b_0) \quad = t(1)(b_0)$$
$$b_2 = p(c(b_1))(b_1) \quad = t(2)(b_1)$$
$$\ldots$$
$$b_n = p(c(b_{n-1}))(b_{n-1}) \quad = t(n)(b_{n-1})$$
$$c(b_n) = 0$$

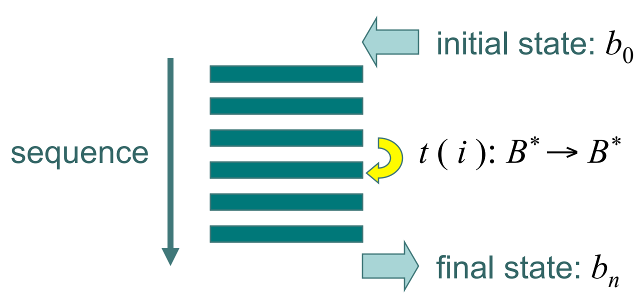
Execution defines a *partial function* (defined on a subset of the domain) from the initial state to final state:

$$e_p : B^* \to B^*$$

This function is undefined if the thread does not terminate.

Lee 05: 39

Threads as Sequences of State Changes



- Time is irrelevant
- All actions are ordered
- The thread sequence depends on the program and the state

Lee 05: 40

Expressiveness

Given a finite action set: $A \subset [B^* \to B^*]$

Execution: $e_p \in [B^* \to B^*]$

Can all functions in $[B^* \to B^*]$ be defined by a program?

Compare the cardinality of the two sets:

set of functions: $[B^* \to B^*]$

set of programs: $[\{1, \ldots, m\} \to A,\ m \in N] = A^*$

Programs Cannot Define All Functions

Cardinality of this set: $[\{1, \ldots, m\} \to A,\ m \in N]$ is the same as the cardinality of the set of integers (put the elements of the set into a one-to-one correspondence with the integers). The set is countable.

This set is larger: $[B^* \to B^*]$.

Proof: Consider the subset of total functions. Isomorphic (there exists a bijection) to $[N \to N]$ using binary encoding of the integers. This set is not countable (use Cantor's diagonal argument to show this).

Taxonomy of Functions

Functions from initial state to final state:

$$F = [\, N \to N \,]$$

Partial recursive functions:

$$PR \subset [\, N \to N \,] \quad \text{(partial functions)}$$

(Those functions for which there is a program that terminates for zero or more initial states (arguments). The domain of the function is the set on which it terminates).

Total recursive functions:

$$TR \subset P \subset [\, N \to N \,]$$

(There is a program that terminates for all initial states).

Lee 05: 43

Church-Turing Thesis

Every function that is computable by any practical computer is in PR.

There are many “good” choices of finite action sets that yield an isomorphic definition of the set PR

Evidence that this set is fundamental is that Turing machines, lambda calculus, PCF (a basic recursive programming language), and all practical computer instruction sets yield isomorphic sets PR.

Lee 05: 44

Key Results in Computation

Turing: Instruction set with 7 instructions is enough to write programs for all partial recursive functions.

- A program using this instruction set is called a Turing machine
- A *universal Turing machine* is a Turing machine that can execute a binary encoding of any Turing machine.

Church: Instructions are a small set of transformation rules on strings called the lambda calculus.

- Equivalent to Turing machines.

Turing Completeness

A *Turing complete* instruction set is a finite subset of PR (and probably of TR) whose transitive closure is PR.

Many choices of underlying instruction sets $A \subset [N \to N]$ are Turing complete and hence equivalent.

Equivalence

Any two programs that implement the same partial recursive function are equivalent.

- Terminate for the same initial states.
- End up in the same final states.

NOTE: Big problem for embedded software:

- All non-terminating programs are equivalent.
- All programs that terminate in the same "exception" state are equivalent.

Limitations of the 20-th Century Theory of Computation

- Only terminating computations are handled.

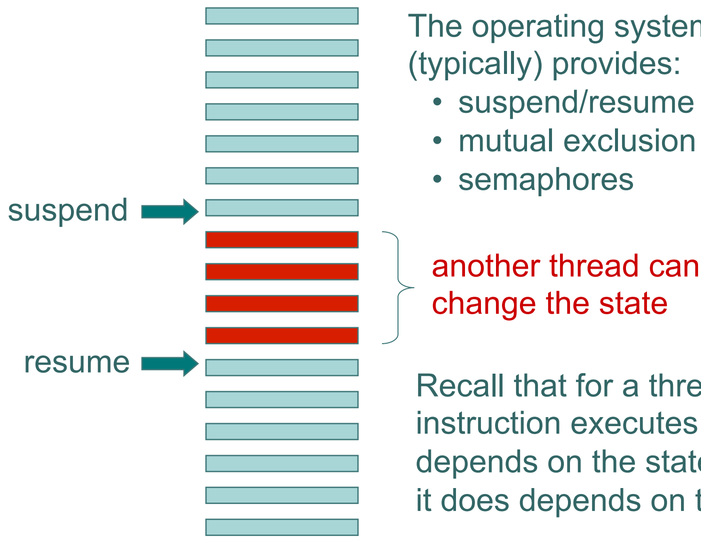
This is not very useful…
But it gets even worse:

- There is no concurrency.

Concurrency: Interactions Between Threads

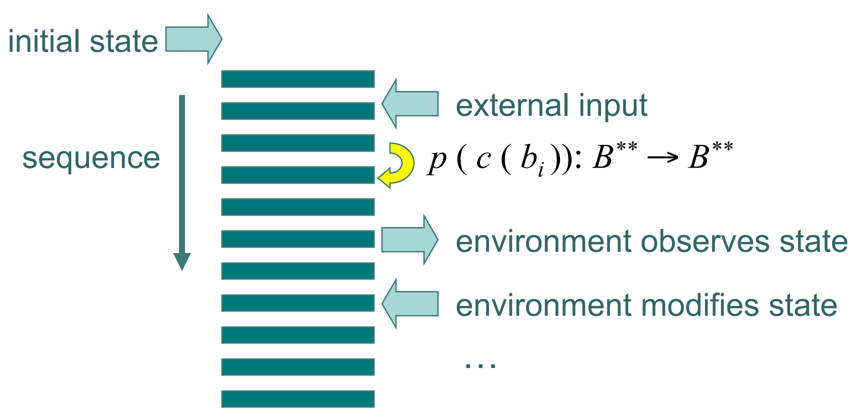
The operating system (typically) provides:

- suspend/resume
- mutual exclusion
- semaphores



Recall that for a thread, which instruction executes next depends on the state, and what it does depends on the state.

Nonterminating and/or Interacting Threads: Allow State to be Observed and Modified



Recall Execution of a Program

Given initial state $b_0 \in B^*$, then execution is:

$b_1 = p\,(\,c\,(\,b_0\,))(\,b_0\,) \quad = t\,(1)(\,b_0\,)$

$b_2 = p\,(\,c\,(\,b_1\,))(\,b_1\,) \quad = t\,(2)(\,b_1\,)$

$\ldots$

$b_n = p\,(\,c\,(\,b_{n\text{-}1}\,))(\,b_{n\text{-}1}\,) \quad = t\,(n)(\,b_{n\text{-}1}\,)$

$c\,(\,b_n\,) = 0$

When a thread executes alone, execution is a composition of functions:

$t\,(n) \circ \ldots \circ t\,(2) \circ t\,(1)$

Interleaved Threads

Consider two threads with functions:

$t_1(1),\, t_1\,(2),\, \ldots\,,\, t_1\,(n)$

$t_2\,(1),\, t_2\,(2),\, \ldots\,,\, t_2\,(m)$

These functions are arbitrarily interleaved.

Worse: The i-th action executed by the machine, if it comes from program $c\,(\,b_{i\text{-}1})$, is:

$t\,(i) = p\,(\,c\,(\,b_{i\text{-}1}))$

which depends on the state, which may be affected by the other thread.

Equivalence of Pairs of Programs

For concurrent programs p_1 and p_2 to be equivalent under threaded execution to programs p_1' and p_2', we need for each arbitrary interleaving of the thread functions produced by that interleaving to terminate and to compose to the same function as all other interleavings for both programs.

This is hopeless, except for trivial concurrent programs!

Equivalence of Individual Programs

If program p_1 is to be executed in a threaded environment, then without knowing what other programs will execute with it, there is no way to determine whether it is equivalent to program p_1' except to require the programs to be identical.

This makes threading nearly useless, since it makes it impossible to reason about programs.

Determinacy

For concurrent programs p_1 and p_2 to be *determinate* under threaded execution we need for each arbitrary interleaving of the thread functions produced by that interleaving to terminate and to compose to the same function as all other interleavings.

This is again hopeless, except for trivial concurrent programs!

Moreover, without knowing what other programs will execute with it, we cannot determine whether a given program is determinate.

Manifestations of Problems

- Race conditions
  - Two threads modify the same portion of the state. Which one gets there first?
- Consistency
  - A data structure with interdependent data is updated in multiple atomic actions. Between these actions, the state is inconsistent.
- Deadlock
  - Fixes to the above two problems result in threads waiting for each other to complete an action that they will never complete.

Improving the Utility of the Thread Model

Brute force methods for making threads useful:

- Segmented memory (processes)
  - Pipes and file systems provide mechanisms for sharing data.
  - Implementation of these requires a thread model, but this implementation is done by operating system expert, not by application programmers.
- Functions (no side effects)
  - Disciplined programming design pattern, or…
  - Functional languages (like Concurrent ML)
- Single assignment of variables
  - Avoids race conditions

Mechanisms for Achieving Determinacy

Less brute force (but also weaker):

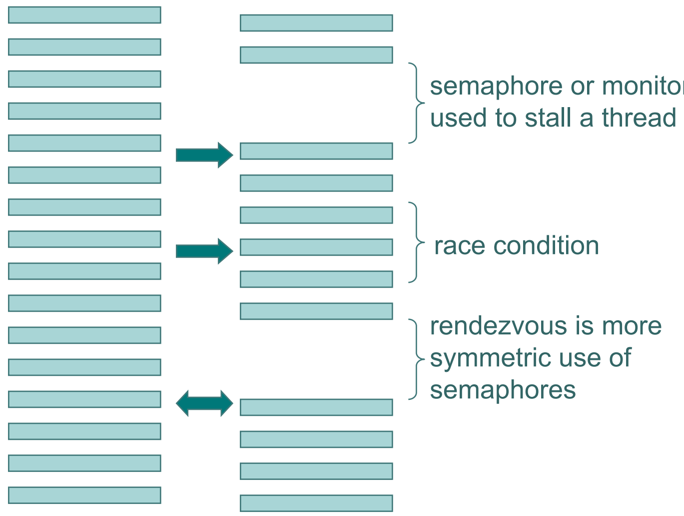
- Semaphores
- Mutual exclusion locks (*mutexes*, *monitors*)
- Rendezvous

All require an atomic test-and-set operation, which is not in the Turing machine instruction set.

Mechanisms for Interacting Threads

Potential for race conditions, inconsistency, and deadlock severely compromise software reliability.

These methods date back to the 1960's (Dijkstra).



Deadlock

"Acquire lock *x*" means the following atomic action:

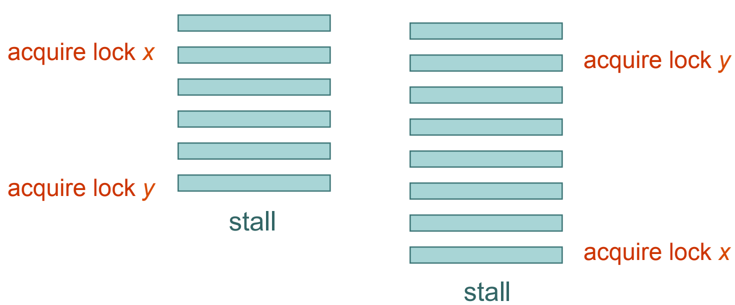
if *x* is false, set it to true,

else stall until it is false.

where *x* is Boolean variable (a "semaphore").

"Release lock *x*" means:

set *x* to false.



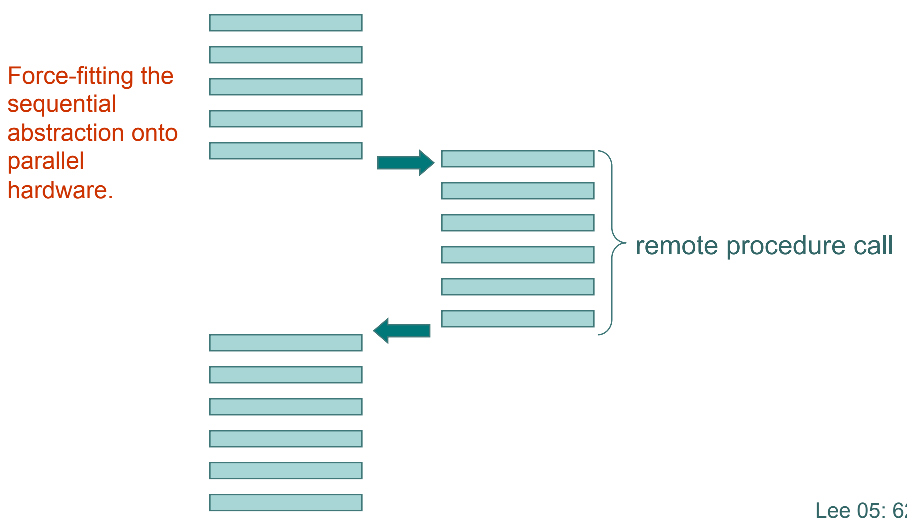
Simple Rule for Avoiding Deadlock [Lea]

"Always acquire locks in the same order."

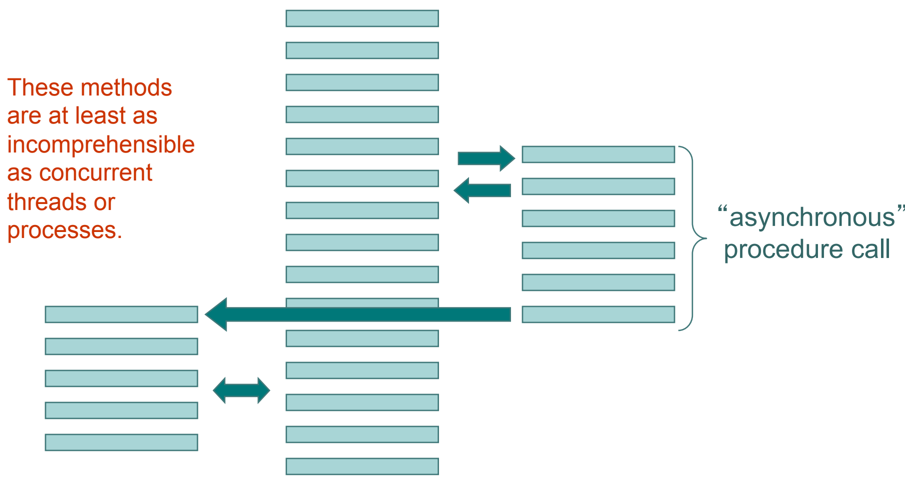
However, this is very difficult to apply in practice:

- Method signatures do not indicate what locks they grab (so you need access to all the source code of methods you use).
- Symmetric accesses (where either thread can initiate an interaction) become more difficult.

Distributed Computing: In Practice, Often Based on Remote Procedure Calls (RPC)



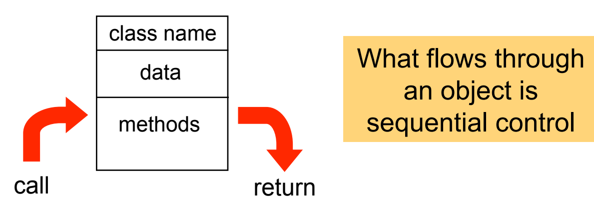
Combining Processes and RPC –
Split-Phase Execution, Futures,
Asynchronous Method Calls, Callbacks, …



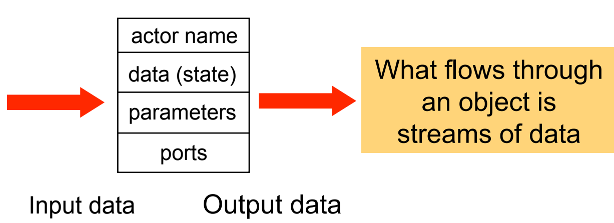
Lee 05: 63

What is an *Actor-Oriented* MoC?

Traditional component interactions:



Actor oriented:



Lee 05: 64

Models of Computation Implemented in Ptolemy II

CI – Push/pull component interaction
Click – Push/pull with method invocation
CSP – concurrent threads with rendezvous
CT – continuous-time modeling
DE – discrete-event systems
DDE – distributed discrete events
FSM – finite state machines
DT – discrete time (cycle driven)
Giotto – synchronous periodic
GR – 2-D and 3-D graphics
PN – process networks
DPN – distributed process networks
SDF – synchronous dataflow
SR – synchronous/reactive
TM – timed multitasking

Most of these are actor oriented.

Do we have a sound foundation for concurrent programming?

If the foundation is bad, then we either tolerate *brittle designs* that are difficult to make work, or we have to rebuild from the foundations.

Note that this whole enterprise is held up by threads

Summary

- Theory of computation supports well only
  - terminating
  - non-concurrent

  computation

- Threads are a poor concurrent model of computation
  - weak formal reasoning possibilities
  - incomprehensibility
  - race conditions
  - inconsistent state conditions
  - deadlock risk

Lee 05: 67