Event-based Concurrency Control

Tom Van Cutsem

Software Languages Lab
Vrije Universiteit Brussel

Copyright is held by the author/owner(s).
OOPSLA'09, October 25–29, 2009, Orlando, FL, USA.
ACM 09/10.

Goals

- Composing concurrent tasks
- Overview of existing models, their benefits and drawbacks
- Propose events as an alternative to the predominant model of multithreading
- Show that event-driven programming can be generalized to exploit multiple CPUs/cores

Agenda

- Before break:
  - Threads
  - Actors
- After break:
  - Event-driven programming
  - (Communicating) Event Loops

Why concurrency?

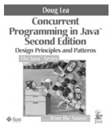
- to express independent tasks
- to deal effectively with I/O: Files, Sockets, ...
- for interactiveness (GUI, Games)
- distributed systems are inherently concurrent
- for efficiency (Scientific apps, web servers)

Parallel vs Concurrent Programming

- Parallel programming: efficiency
  - Matrix multiplication, FFT, search, solving PDEs, monte carlo, ...
- Concurrent programming: architectural reasons
  - UI, I/O, ensuring responsiveness, distributed computing, etc.

Threads (& Locks)

Why threads are a bad idea (for most purposes)
John Ousterhout
Invited Talk at the 1996 USENIX Technical Conference



Concurrent Programming in Java: Design Principles and Patterns
Doug Lea

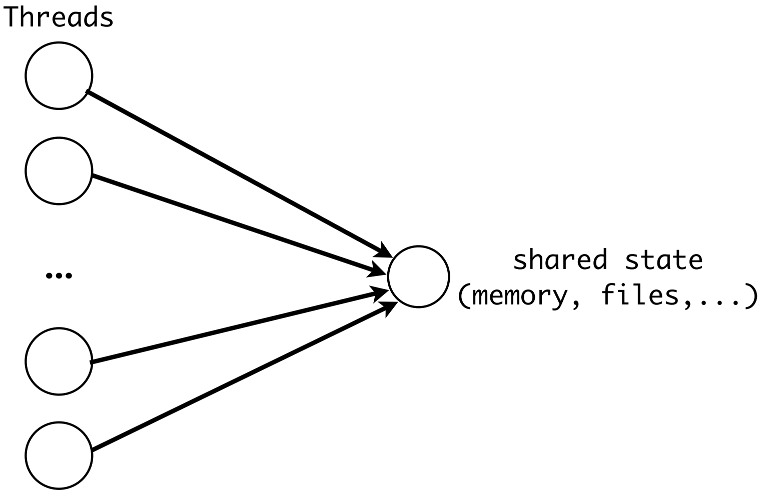
Threads

- Multiple independent control flows
- Scheduler determines interleaving (implicit)
- Communicate by synchronously reading & writing shared data
- Synchronization via locks and condition variables

Preemptive Scheduling

- A thread:
  - may be preempted by any other thread at any time => inconsistent state, non-determinism
  - must never explicitly yield control to another thread => automatic context switching

Threads

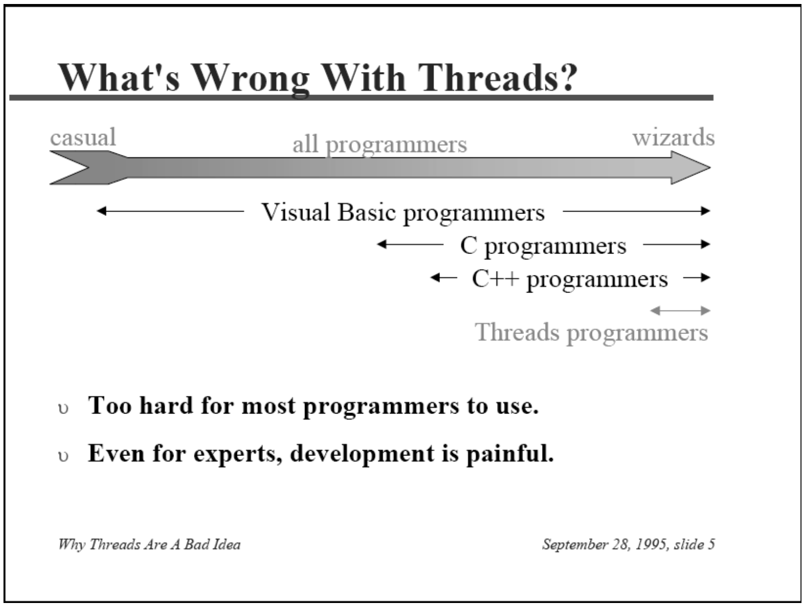


Threads

...

shared state
(memory, files,...)

Threads are for Wizards



What's Wrong With Threads?

- Too hard for most programmers to use.
- Even for experts, development is painful.

Why Threads Are A Bad Idea *September 28, 1995, slide 5*

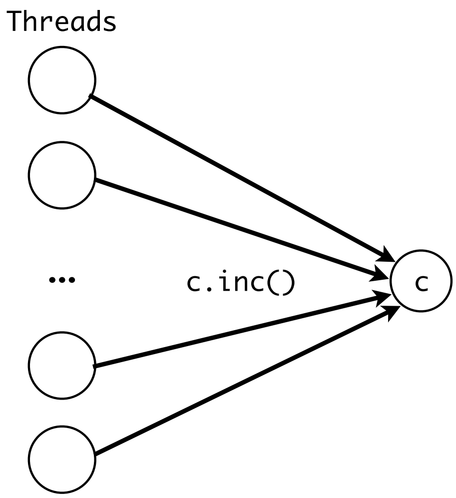
(Ousterhout, 1995)

The Problem with Threads

- seemingly straightforward adaptation of sequential programming model
- but: huge amount of non-determinism
- programmer's job is to prune unwanted non-determinism

The Problem with Threads
Edward Lee
IEEE Computer, Vol. 39, No. 5, pp. 33-42, May 2006

Example: concurrent increments



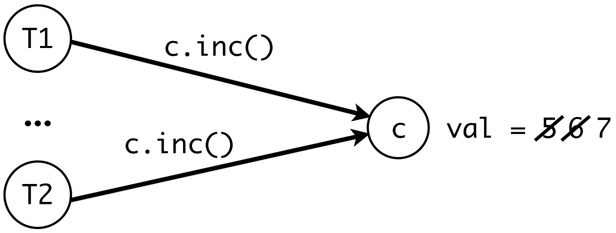
Unsynchronized Counter

```
final Counter c = new Counter();

Thread[] threads = new Thread[MAX_THREADS];
for (int i = 0; i < MAX_THREADS; i++) {
   threads[i] = new Thread(new Runnable() {
      public void run() {
         for (int j = 0; j < NUM_INCS; j++) {
            c.inc();
         }
      }
   });
   threads[i].start();
}

// wait for all threads to finish
for (int j = 0; j < threads.length; j++) {
   threads[j].join();
}
```

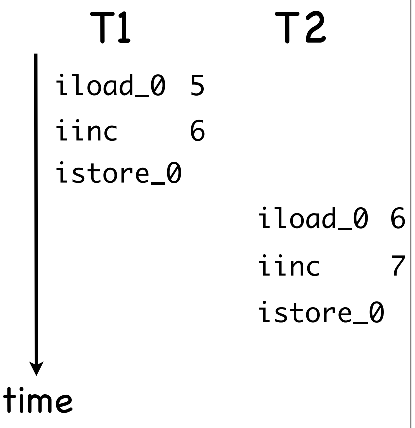
```
class Counter {
  private int val = 0;
  public void inc() {
    val = val + 1;
  }
}
```

MAX_THREADS = 10
NUM_INCS = 100.000
inc() -> 1.000.000x
c.val -> 827.674
time -> 16 millisec

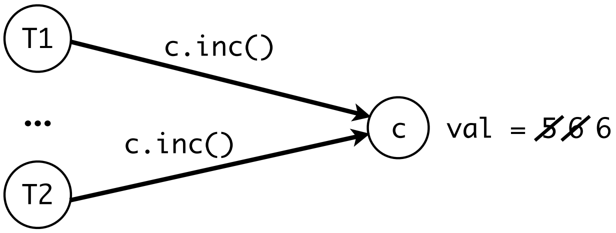
Runtime view



```
inc() { val = val + 1; }

      iload_0
      iinc
      istore_0
```

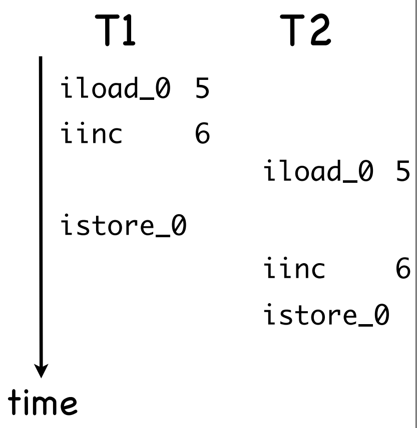


Race Conditions



val = ~~5~~ ~~6~~ 6

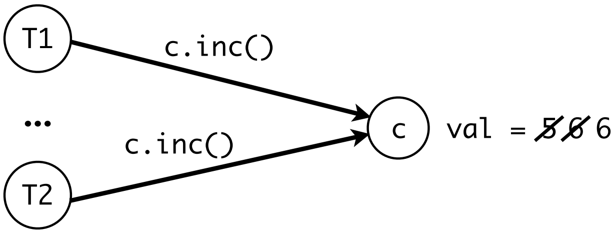
```
inc() { val = val + 1; }

      iload_0
      iinc
      istore_0
```



```
T1              T2
iload_0  5
iinc     6
                iload_0  5
istore_0
                iinc     6
                istore_0
```

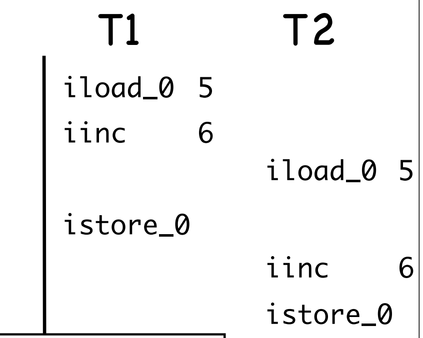
time

Race Conditions



val = ~~5~~ ~~6~~ 6

```
inc() { val = val + 1; }

      iload_0
```



```
T1              T2
iload_0  5
iinc     6
                iload_0  5
istore_0
                iinc     6
                istore_0
```

Outcome depends on thread scheduler!

Race Conditions

- When program output depends unexpectedly upon the arbitrary ordering of concurrent activities

---

Synchronized Counter

```
final Counter c = new Counter();

Thread[] threads = new Thread[MAX_THREADS];
for (int i = 0; i < MAX_THREADS; i++) {
   threads[i] = new Thread(new Runnable() {
      public void run() {
         for (int j = 0; j < NUM_INCS; j++) {
            c.inc();
         }
      }
   });
   threads[i].start();
}

// wait for all threads to finish
for (int j = 0; j < threads.length; j++) {
   threads[j].join();
}
```

```
class Counter {
  private int val = 0;
  public void inc() {
    val = val + 1;
  }
}
```

MAX_THREADS = 10
NUM_INCS = 100.000
inc() -> 1.000.000x
c.val -> 827.674
time -> 16 millisec

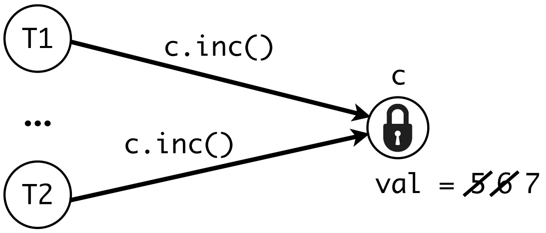
Synchronized Counter

```
final Counter c = new Counter();

Thread[] threads = new Thread[MAX_THREADS];
for (int i = 0; i < MAX_THREADS; i++) {
   threads[i] = new Thread(new Runnable() {
      public void run() {
         for (int j = 0; j < NUM_INCS; j++) {
            synchronized (c) { c.inc(); }
         }
      }
   });
   threads[i].start();
}

// wait for all threads to finish
for (int j = 0; j < threads.length; j++) {
   threads[j].join();
}
```

```
class Counter {
  private int val = 0;
  public void inc() {
    val = val + 1;
  }
}
```

MAX_THREADS = 10
NUM_INCS = 100.000
inc() -> 1.000.000x
c.val -> 1.000.000
time -> 159 millisec

Locking



```
synchronized(c) {
  val = val + 1;
}
```

```
monitorenter
iload_0
iinc
istore_0
monitorexit
```

T1	T2
monitorenter	
iload_0 5	
iinc 6	
	monitorenter
istore_0	
monitorexit	
	monitorenter
	iload_0 6
	iinc 7
	istore_0
	monitorexit

time

Locking Requires Cooperation

- All involved threads must acquire the lock!
- A single thread that forgets to take the lock may concurrently enter the critical section
- Locking protocols

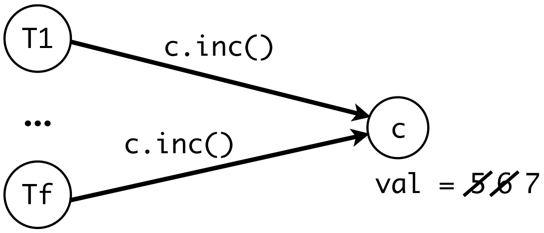
One Forgetful Thread

```
final Counter c = new Counter();

Thread[] threads = new Thread[MAX_THREADS-1];
for (int i = 0; i < threads.length; i++) {
   threads[i] = new Thread(new Runnable() {
      public void run() {
         for (int j = 0; j < NUM_INCS; j++) {
            synchronized (c) { c.inc(); }
         }
      }
   });
   threads[i].start();
}

Thread forgetful = new Thread(new Runnable() {
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      c.inc();
    }
  }
});
forgetful.start();
```

```
class Counter {
  private int val = 0;
  public void inc() {
    val = val + 1;
  }
}
```

MAX_THREADS = 10
NUM_INCS = 100.000
inc() -> 1.000.000x
c.val -> 985.724
time -> 242 millisec

One Forgetful Thread



```
synchronized(c) {
  val = val + 1;
}
```

```
monitorenter
iload_0
iinc
istore_0
monitorexit
```

```
val = val + 1;
```

```
iload_0
iinc
istore_0
```

T1	Tf
monitorenter	
iload_0 5	
iinc 6	
	iload_0 5
istore_0	
monitorexit	
	iinc 6
	istore_0

time

Enforcing synchronization

```
final Counter c = new Counter();

Thread[] threads = new Thread[MAX_THREADS-1];
for (int i = 0; i < threads.length; i++) {
   threads[i] = new Thread(new Runnable() {
      public void run() {
         for (int j = 0; j < NUM_INCS; j++) {
            synchronized (c) { c.inc(); }
         }
      }
   });
   threads[i].start();
}

Thread forgetful = new Thread(new Runnable() {
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      c.inc();
    }
  }
});
forgetful.start();
```

```
class Counter {
  private int val = 0;
  public synchronized void inc() {
    val = val + 1;
  }
}
```

Enforcing synchronization

```
final Counter c = new Counter();

Thread evenIncT = new Thread(new Runnable() {
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      c.inc();
      c.inc();
    }
  }
});
evenIncT.start();
```

```
class Counter {
  private int val = 0;
  public synchronized void inc() {
    val = val + 1;
  }
  public synchronized int count() {
    return val;
  }
}
```

```
Thread inspectorT = new Thread(new Runnable() {
  boolean sawOdd = false;
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      sawOdd = sawOdd | (c.count() % 2 == 1);
    }
  }
});
inspectorT.start();
```

sawOdd = true

Enforcing synchronization

```
final Counter c = new Counter();
Thread evenIncT = new Thread(new Runnable() {
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      synchronized (c) {
        c.inc();
        c.inc();
      }
    }
  }
});
evenIncT.start();
```

```
class Counter {
  private int val = 0;
  public synchronized void inc() {
    val = val + 1;
  }
  public synchronized int count() {
    return val;
  }
}
```

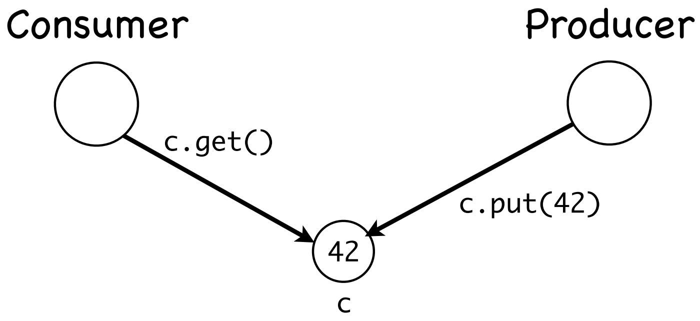
```
Thread inspectorT = new Thread(new Runnable() {
  boolean sawOdd = false;
  public void run() {
    for (int j = 0; j < NUM_INCS; j++) {
      sawOdd = sawOdd | (c.count() % 2 == 1);
    }
  }
});
inspectorT.start();
```

sawOdd = false

Condition Variables

- Make threads wait for each other (without "busy waiting")
- In Java: all objects are condition variables
  - `wait`: suspend thread until notified
  - `notify`: wake up arbitrary waiting thread
  - `notifyAll`: wake up all waiting threads

A cell object



Consumer

Producer

`c.get()`

`c.put(42)`

42

c

A cell object

```
class Cell {
  private int content = 0;
  private boolean isEmpty = true;
  public synchronized void put(int v) {
    while (!isEmpty) {
      try {
        this.wait();
      } catch (InterruptedException e) { }
    }
    isEmpty = false;
    this.notifyAll();
    content = v;
  }
  ...
```

```
  ...
  public synchronized int get() {
    while (isEmpty) {
      try {
        this.wait();
      } catch (InterruptedException e) { }
    }
    isEmpty = true;
    this.notifyAll();
    return content;
  }
}
```

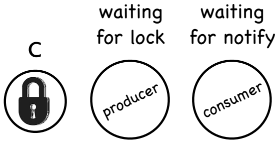
Producers & Consumers

```
final Cell c = new Cell();
```

```
Thread producer = new Thread(new Runnable() {
  public void run() {
    for (int i = 0; i < n; i++) {
      c.put(produce(i));
    }
  }
});
```

```
Thread consumer = new Thread(new Runnable() {
  public void run() {
    for (int i = 0; i < n; i++) {
      consume(c.get());
    }
  }
});
```

Trace



```
class Cell {
  private int content = ~~0~~; 42
  private boolean isEmpty = ~~true~~; ~~false~~; true

  public synchronized void put(int v) {
    while (!isEmpty) { this.wait(); }
    isEmpty = false;
    this.notifyAll();
    content = v;
  }

  public synchronized int get() {
    while (isEmpty) { this.wait(); }
    isEmpty = true;
    this.notifyAll();
    return content;
  }
}
```

producer	consumer
	c.get()
	lock(c)
c.put(42)	
lock(c)	
	isEmpty?
	this.wait()
lock(c)	
!isEmpty?	
isEmpty = false	
this.notifyAll()	
	lock(c)
content = 42	
unlock(c)	
	lock(c)
	isEmpty?
	isEmpty = true

time

Deadlocks

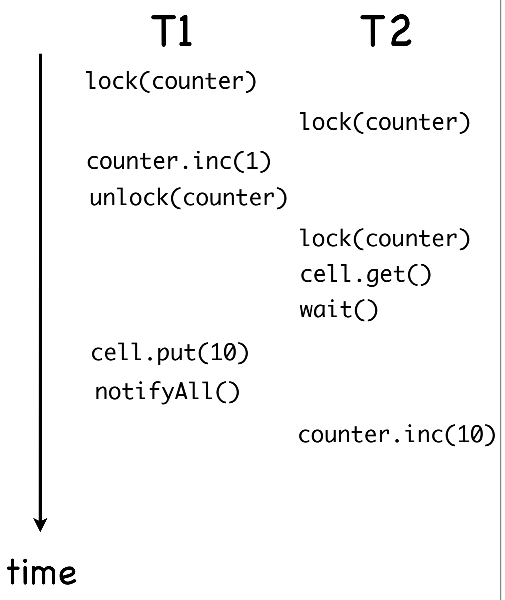
```
class Counter {
  private int val = 0;
  public void inc(n) {
    val = val + n;
  }
}
final Counter c = new Counter();
final Cell cell = new Cell();
```

```
t1 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(1);
    }
    cell.put(10);
  }
})
```

```
t2 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(cell.get());
    }
  }
})
```

Deadlocks

```
t1 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(1);
    }
    cell.put(10);
  }
})


t2 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(cell.get());
    }
  }
})
```

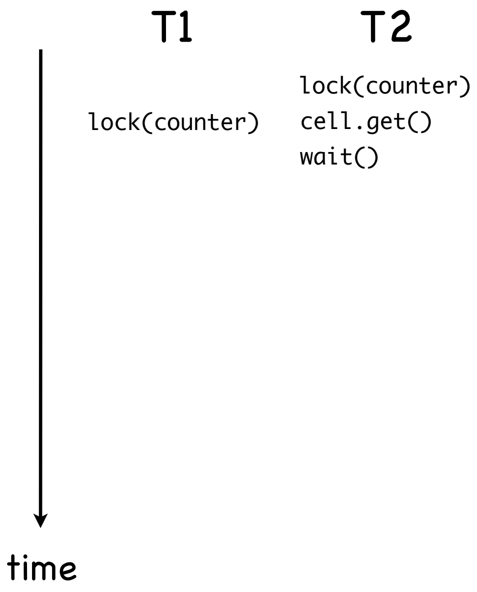


T1	T2
lock(counter)	
	lock(counter)
counter.inc(1)	
unlock(counter)	
	lock(counter)
	cell.get()
	wait()
cell.put(10)	
notifyAll()	
	counter.inc(10)

time

Deadlocks

```
t1 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(1);
    }
    cell.put(10);
  }
})


t2 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(cell.get());
    }
  }
})
```

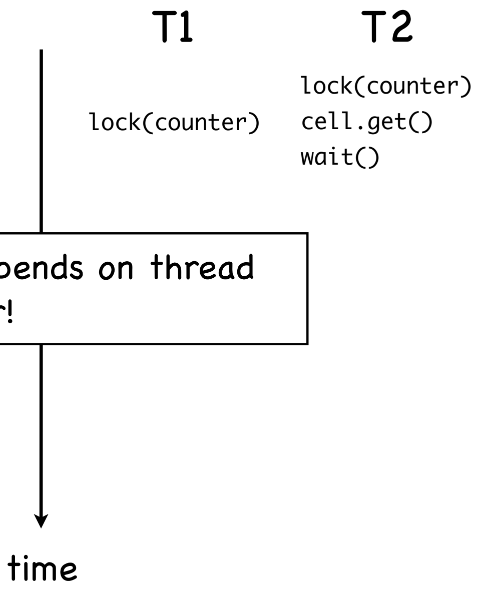


T1	T2
	lock(counter)
lock(counter)	cell.get()
	wait()

time

Deadlocks

```
t1 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(1);
    }
    cell.put(10);
  }
})

t2 = new Thread(new Runnable() {
  public void run() {
    synchronized (counter) {
      counter.inc(cell.get());
    }
  }
})
```



Deadlock occurrence depends on thread scheduler!

Beware! Here be ~~dragons~~ threads

- Preemption: unit of concurrent interleaving is (bytecode) instruction or even smaller => not visible in the code
- Locking protocol requires cooperation from all threads => scattered throughout code
- Locking too little => race conditions
- Locking too much => deadlocks

Some advantages

- Synchronous communication does not disrupt sequential control flow
- Can exploit true multiprocessor concurrency (one thread per physical CPU/core)
- OS Support (but often heavyweight and platform-dependent)

… and some more disadvantages

- Not easily distributable: shared-memory assumption
- Limited scalability: context switch for preemptively scheduled threads is heavyweight
- Overhead of managing thread state on stack
- But…

Why events are a bad idea (for high-concurrency servers)
von Behren, Condit and Brewer
Proceedings of the 9th USENIX Conference on
Hot Topics in Operating Systems, 2003

Best Practices

- Keep critical sections as small as possible
- Reduce shared state to a minimum
- Avoid calls to unknown code while holding locks
- Confine conditional synchronization to high-level abstractions (e.g. a bounded buffer)
- Instead of spawning a large number of threads, better to use an event loop (e.g. managing client socket connections)

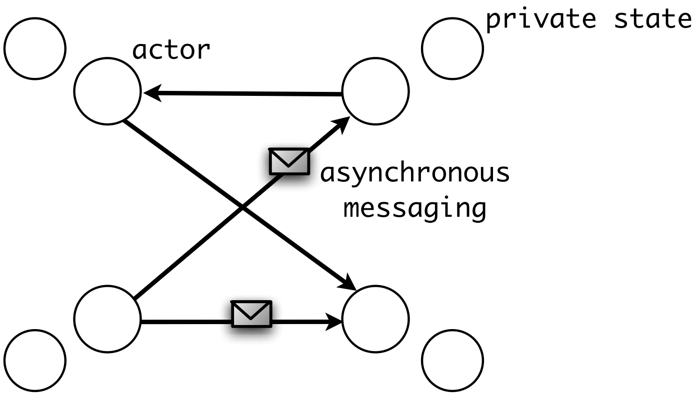
Actors

Concurrent Object-oriented Programming
Gul Agha
In Communications of the ACM, Vol 33 (9), p. 125, 1990

The Actor Model

- Hewitt, Baker, Clinger, Agha, ... (MIT, late 1970s)
  - (formed direct motivation to build Scheme!)
- Fundamental model of concurrent computation
- Designed for open distributed systems
- Functional and stateful (imperative) variants

Actors

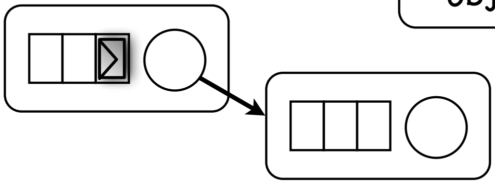


Functional Actors

- An actor has:
  - A mailbox: buffer of incoming messages
  - A behaviour: a script to process incoming messages
  - Acquaintances: references to other actors

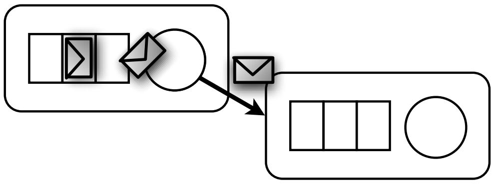
“object + methods”

“object references”



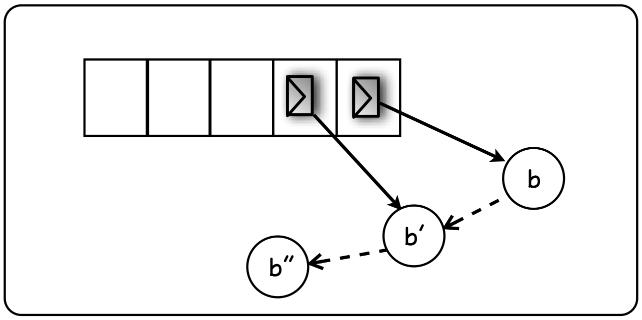
Functional Actors

- In response to a message, an actor can:
  - create new actors
  - send messages (asynchronously)
  - become a new behavior



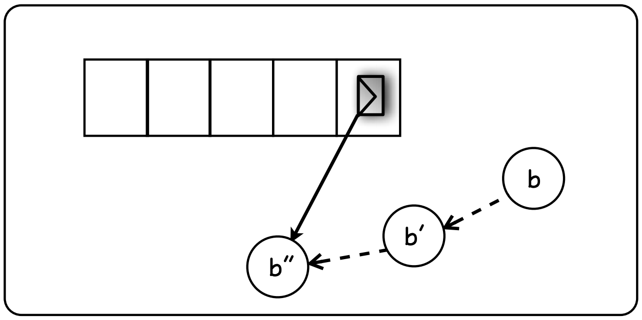
Functional Actors

- become: specify replacement behaviour
- original and replacement behaviour process messages in parallel (pipelining!)



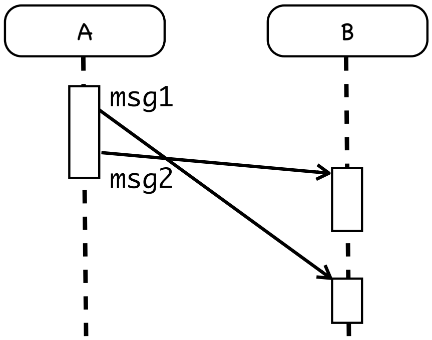
Functional Actors

- become: specify replacement behaviour
- original and replacement behaviour process messages in parallel



(Weak) Guarantees

- Messages not necessarily received in order of sending time
- Every message is eventually delivered



Example: a counter actor

Functional

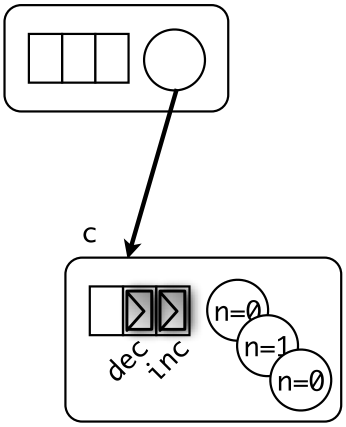
```
def makeCounter(n) {
  behaviour {
    def inc() { become makeCounter(n+1) }
    def dec() { become makeCounter(n-1) }
    def read(customer) {
      customer<-readResult(n)
    }
  }
}

def c = actor makeCounter(0)
c<-inc()
c<-dec()
```

customer = callback

no return value

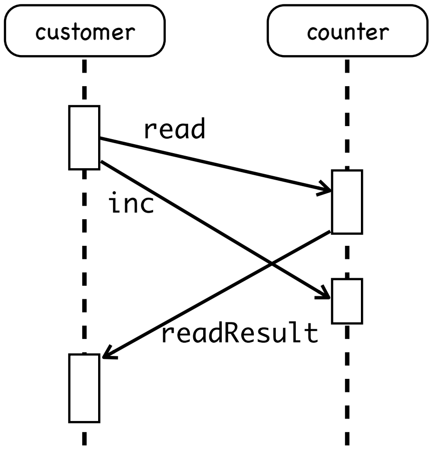
Example: a counter actor

```
def makeCounter(n) {
  behaviour {
    def inc() { become makeCounter(n+1) }
    def dec() { become makeCounter(n-1) }
    def read(customer) {
      customer<-readResult(n)
    }
  }
}

def c = actor makeCounter(0)
c<-inc()
c<-dec()
```



Asynchronous Communication

```
def makeCustomer(counter) {
  behaviour {
    def act() {
      counter<-read(thisActor);
      counter<-inc();
    }
    def readResult(val) { ... }
  }
}

def counter = actor makeCounter(0)
def customer = actor makeCustomer(counter)
customer<-act()
```



Explicit Continuations

- Pure actor model requires continuation passing style (all message sends are asynchronous)
- Has been addressed in many ways:
  - Mixing actors with sequential programming
  - Futures (e.g. ABCL, now also in Java)
  - Token-passing continuations (e.g. SALSA)

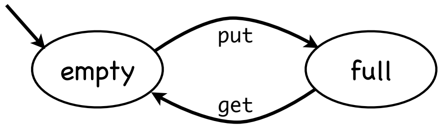
Conditional Synchronization

- Messages that cannot be processed by a behaviour remain in the mailbox
- Allows to postpone processing of a message until the actor is in a suitable state

Example: a cell actor

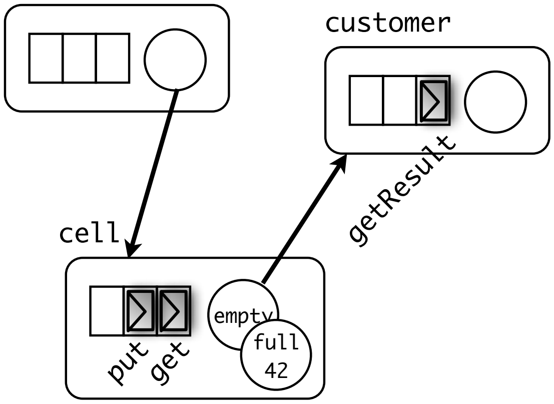
```
def emptyCell = behaviour {
  def put(value) { become makeFullCell(value) }
}
def makeFullCell(val) {
  behaviour {
    def get(customer) {
      become emptyCell
      customer<-getResult(val)
    }
  }
}
def cell = actor emptyCell
c<-get(aCustomer)
c<-put(42)
```

state changes



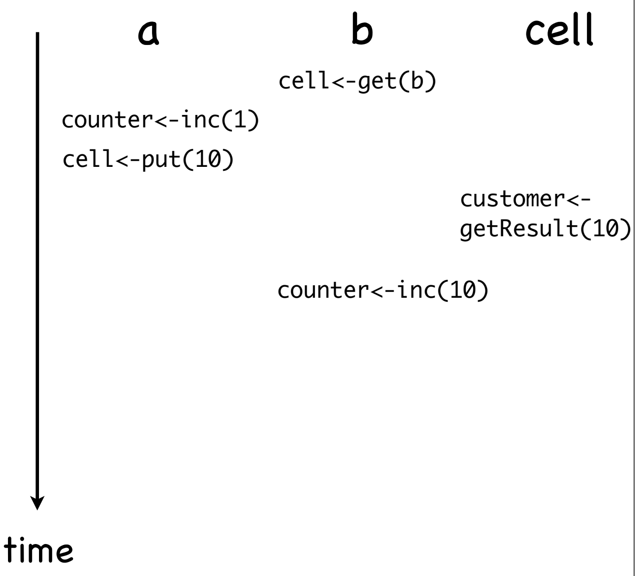
Example: a cell actor

```
def emptyCell = behaviour {
  def put(value) {
    become makeFullCell(value)
  }
}
def makeFullCell(val) {
  behaviour {
    def get(customer) {
      become emptyCell
      customer<-getResult(val)
    }
  }
}

def cell = actor emptyCell
c<-get(aCustomer)
c<-put(42)
```



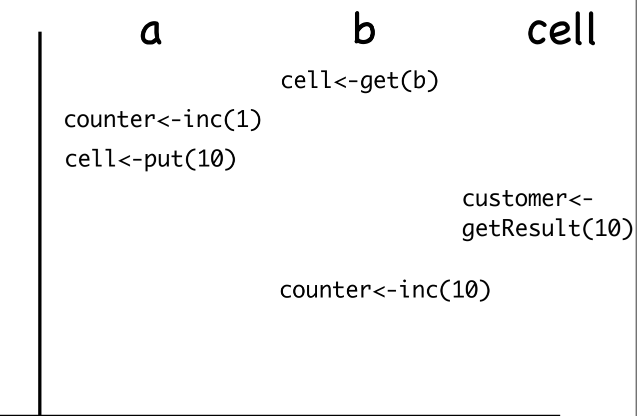
Actors and Deadlock

```
def cell = actor emptyCell;
def counter = actor makeCounter(0);

def a = behaviour {
  def act() {
    counter<-inc(1)
    cell<-put(10)
  }
}
def b = behaviour {
  def act() {
    cell<-get(thisActor)
  }
  def getResult(val) {
    counter<-inc(val)
  }
}

(actor a) <- act()
(actor b) <- act()
```



Actors and Deadlock

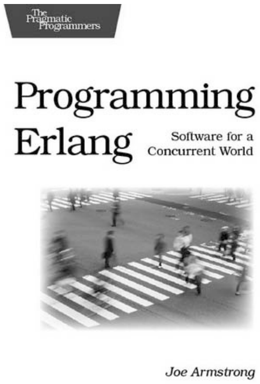
```
def cell = actor emptyCell;
def counter = actor makeCounter(0);

def a = behaviour {
  def act() {
    counter<-inc(1)
    cell<-put(10)
  }
}
def b = behaviour {
  def act() {
    cell<-get(thisActor)
  }
  def getResult(val) {
    counter<-inc(val)
  }
}

(ac
(ac
```



Actors do not deadlock
(but beware of lost progress bugs)

Functional actors in the real world: Erlang

- Joe Armstrong, 1980s
- Developed at Ericsson
- Telephone switches
- New book in 2007



Counter actor in Erlang

```
def makeCounter(n) {
  behaviour {
    def inc() { become makeCounter(n+1) }
    def dec() { become makeCounter(n-1) }
    def read(customer) {
      customer<-readResult(n)
    }
  }
}

def c = actor makeCounter(0)
c<-inc()
c<-dec()
```

```
counterLoop(N) ->
  receive
    inc -> counterLoop(N+1);
    dec -> counterLoop(N-1);
    read(Customer) ->
      Customer ! readResult(N),
      counterLoop(N);
  end.

c = spawn(counterLoop, [0]),
c ! inc,
c ! dec
```

“become” => tail-recursive function
+ explicit receive statement

Erlang Behaviours

- Large Erlang programs abstract from the low-level message passing primitives
- High-level behaviours: servers, finite state machines, event dispatchers

```
% API
make_counter() -> server:start().
inc(C) -> server:cast(C, inc).
dec(C) -> server:cast(C, dec).
read(C) -> server:call(C, read).

% Server implementation
init() -> 0.
handle_cast(inc, N) -> N + 1.
handle_cast(dec, N) -> N - 1.
handle_call(read, N) -> {N, N}.
```

Erlang Behaviours

- Large Erlang programs abstract from the low-level message passing primitives
- High-level behaviours: servers, finite state machines, event dispatchers

```
% API
make_counter() -> server:start().
inc(C) -> server:cast(C, inc).
dec(C) -> server:cast(C, dec).
read(C) -> server:call(C, read).

% Server implementation
init() -> 0.
handle_cast(inc, N) -> N + 1.
handle_cast(dec, N) -> N - 1.
handle_call(read, N) -> {N, N}.
```

current state

reply

new state

Stateful Actors

- May perform assignment on strictly private state
- Execute messages one at a time (serially)
- If no replacement behaviour specified, behaviour remains unchanged

---

Active Objects

- A stateful actor as a combination of:
  - An object representing the behaviour
  - A mailbox to buffer incoming messages
  - A thread of control to process the messages

Example: SALSA

- A stateful actor extension to Java

```
behavior Counter {
  private int count;
  public Counter(int val) { count = val; }
  public void inc() { count = count + 1; }
  public void dec() { count = count - 1; }
}

Counter c = new Counter(0);
c<-inc();
c<-dec();
```

Programming dynamically reconfigurable open systems with SALSA
Varela and Agha
SIGPLAN Not. 36, 12 (Dec. 2001)

Synchronization

```
def makePoint(x,y) {
  behaviour {
    def moveX(dx) { become makePoint(x+dx,y) }
    def moveY(dy) { become makePoint(x,y+dy) }
    def scale(f) { become makePoint(x*f, y*f) }
  }
}

def p = actor makePoint(0,0)

def a = actor {
  def act() {
    p<-moveX(2);
    p<-moveY(4);
  }
}

def b = actor {
  def act() {
    p<-scale(0.5)
  }
}
```

```
p<-moveX(2)
p<-moveY(4)
p<-scale(0.5)
      => (1,2)

p<-moveX(2)
p<-scale(0.5)
p<-moveY(4)
      => (1,4)
```

Synchronization

```
def makePoint(x,y) {
  behaviour {
    def moveX(dx) { become makePoint(x+dx,y) }
    def moveY(dy) { become makePoint(x,y+dy) }
    def scale(f) { become makePoint(x*f, y*f) }
  }
}

def p = actor makePoint(0,0)

def a = actor {
  def act() {
    p<-moveX(2);
    p<-moveY(4);
  }
}

de

}
```

```
p<-moveX(2)
p<-moveY(4)
p<-scale(0.5)

    => (1,2)
```

```
p<-moveX(2)
p<-scale(0.5)
p<-moveY(4)

    => (1,4)
```

$(y+dy)*f \neq (y*f)+dy$

Resulting point depends on execution schedule of messages

Client-side synchronization

- Locks not required only as long as a message can be processed entirely sequentially

~~synchronized (p) {~~
~~p<-moveX(2);~~
~~p<-moveY(4);~~
~~}~~

```
p<-move(2,4);
```

```
def makePoint(x,y) {
  behaviour {
    def move(dx,dy) {
      become makePoint(x+dx,y+dy)
    }
  }
}
```

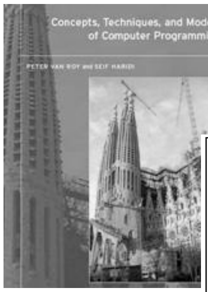
Actors: Advantages

- Message-passing based concurrency: no synchronous access to shared state
    - no locking or race conditions on state
    - easily distributable
- Asynchronous: no deadlocks
- High-level conditional synchronization via behaviour replacement
- Supports multiprocessor concurrency

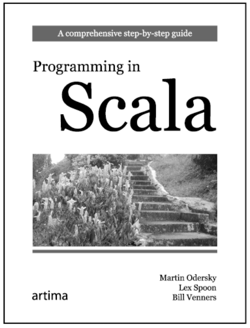
Actors: Disadvantages

- Asynchrony puts constraints on program structure:
    - No return values -> requires callbacks
    - Continuation passing style is unwieldy
- Beware of the ordering of messages
- Impossible for clients to specify additional synchronization conditions

Alive and Kicking

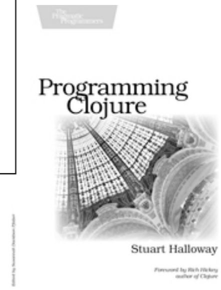


Mozart/Oz: “ports” (2004)



Scala: Erlang-style actors (2008)

Clojure: “agents” (2009)

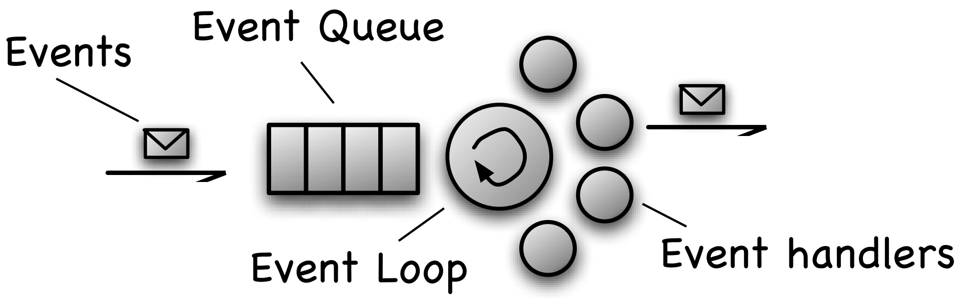


Event-driven Programming

Programming without a call stack
Gregor Hohpe, 2006
Available online: www.enterpriseintegrationpatterns.com

Concurrency among Strangers
Miller, Tribble and Shapiro
In Symposium on Trustworthy global computing, LNCS Vol 3705, pp. 195-229, 2005

Event Loop Model


Events

Event Queue

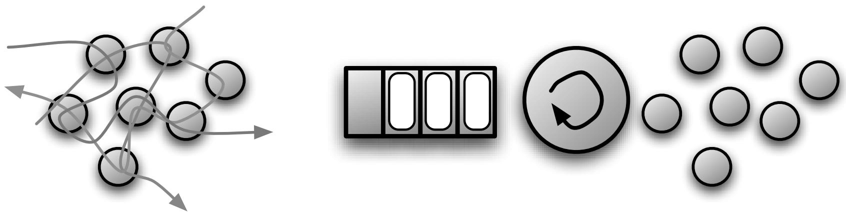
Event Loop

Event handlers

```
while (true) {
  Event e = eventQueue.next();
  switch (e.type) {
    ...
  }
}
```

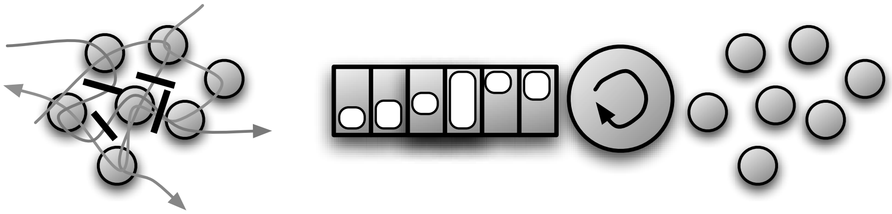
```
void onKeyPressed(KeyEvent e) {
  // process the event
}
```

Event-loop Concurrency



- ○ Let tasks be executed by a single thread
- ○ But what if a single task takes too long?

Event-loop Concurrency



- Split single task into independent fragments
- No locking! => avoids race conditions & deadlocks

Success Stories

- GUIs: events are mouse clicks, button presses, etc.
  - separate event loop keeps GUI responsive
- Distributed systems: events are incoming requests (e.g. read from a socket)
  - asynchronous I/O to exploit I/O overlap

Cooperative Scheduling

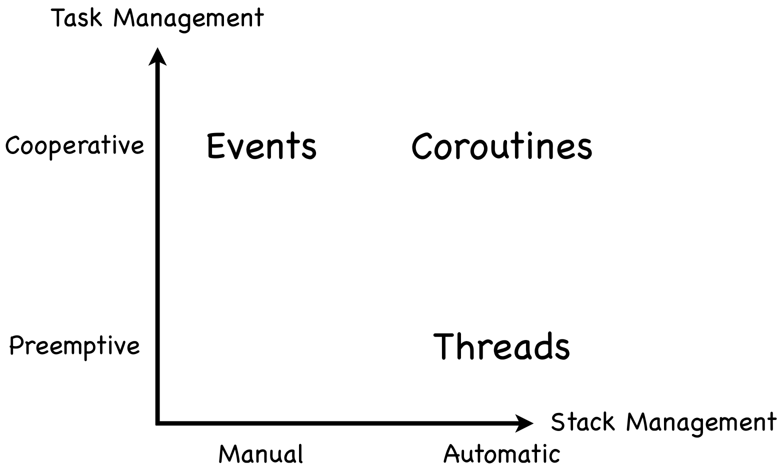
- An event handler:
  - runs without preemption by other event handlers => no race conditions within handler
  - must eventually yield control by returning to the main event loop ("inversion of control") => manual stack management, lightweight tasks

---

Inversion of Control

- Control flow determined by external events
- Program != sequence of instructions (proactive)
- Program = series of event handlers (reactive)
- Flexibility, lightweight tasks, loose coupling
- Fragmented code, cflow becomes obscured

Task vs Stack Management



Task Management

Cooperative Events Coroutines

Preemptive Threads

Stack Management

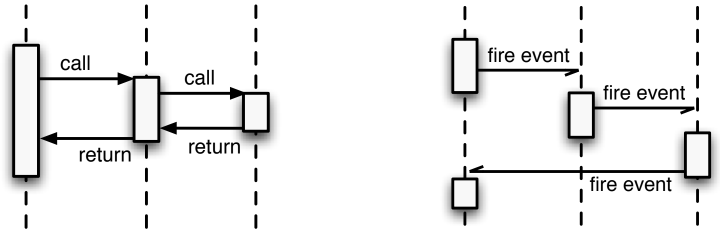
Manual Automatic

Cooperative Task Management without Manual Stack Management
Adya et al.
Proceedings 2002 USENIX Technical Conference

Event-driven programming = programming without a call stack

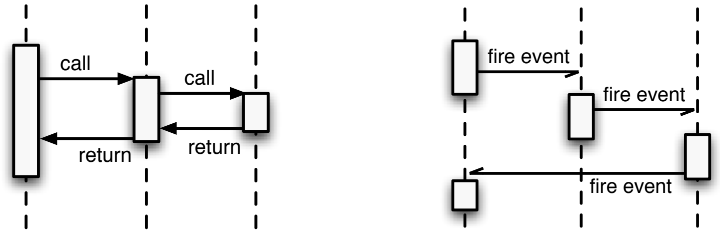
Programming without a call stack
Gregor Hohpe
Available online: www.enterpriseintegrationpatterns.com

Call versus Event



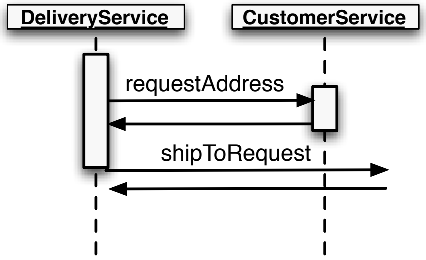
- Programming without a call stack
  - Much more flexible interactions
  - But... free synchronization & context are gone

Call versus Event



- Call stack provides:
  - Coordination: caller waits for callee
  - Continuation: callee returns value to caller
  - Context: upon return, local variables are still available to the caller

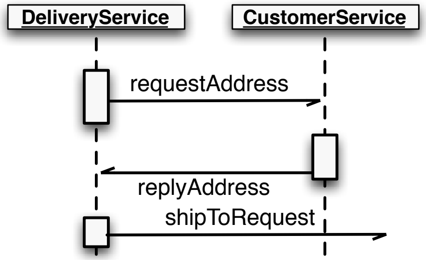
Return values



```
void processDelivery(Order o) {
  // request customer’s address
  Address a = customerService.requestAddress(o.customerId));
  courier.shipToRequest(o, a);
}
```

Callbacks

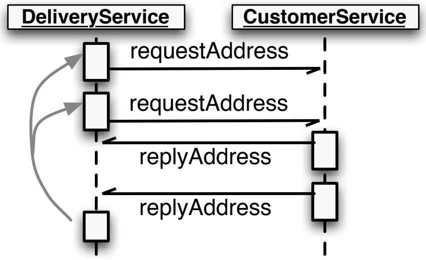
- Reintroduce synchronisation and “return values”



```
void processDelivery(Order o) {
  // store order to retrieve it later
  orders.add(o);
  // request customer’s address
  customerService.receive(
    new RequestAddress(o.orderId, o.customerId));
}

void replyAddress(AddressReply reply) {
  // retrieve order again
  Order o = orders.get(reply.orderId);
  Address a = reply.address;
  courier.receive(new ShipToRequest(o, a));
}
```

Issues with Callbacks

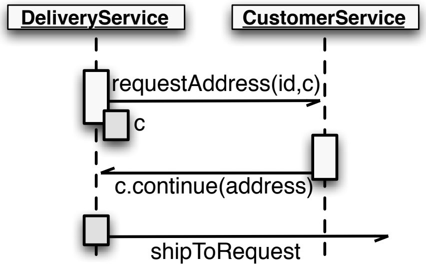
- Fragmented Code: stack ripping
- Callback is out of context:
  - what is its originating call?
  - what was the state (e.g. local variables) when call was made?



Cooperative Task Management without Manual Stack Management
Adya et al.
Proceedings 2002 USENIX Technical Conference

Continuations

- Continuation bundles state where handler left off + function encoding what remains to be done



```
void processDelivery(Order o) {
  customerService.receive(
    new RequestAddress(o.customerId),
    new Continuation() {
      void continue(Result r) {
        Address adr = (Address) r;
        courier.receive(new ShipToRequest(o, adr));
      }
    });
}
```

Continuations

- Continuation can process result in context
- Beware: context may have changed between call and callback

```
void processDelivery(Order o) {
  customerService.receive(
    new RequestAddress(o.customerId),
    new Continuation() {
      void continue(Result r) {
        Address adr = (Address) r;
        courier.receive(new ShipToRequest(o, adr));
      }
    });
}
```

Continuations

- Significant overhead in languages without closures

```
void processDelivery(Order *o) {
  Object args[] = { o };
  Continuation *c = new Continuation(&deliveryCallback, args);
  customerService->receive(new RequestAddress(o->customerId, c));
}

void deliveryCallback(Continuation *cont) {
  // recover local variables
  Order* o = (Order) (cont->args)[0];
  Address* adr = (Address) cont->returnValue;
  courier->receive(new ShipToRequest(o, adr));
  delete cont;
}
```

Event Loop best practices

- Event handlers should be short-lived and return control to the event loop quickly.
- Split up long-running computations by recursively scheduling continuation events.
- Avoid blocking I/O within an event handler. Event loops work best with async. I/O.
- Check whether all handlers are eventually invoked. If not: “lost progress” bug

---

Summary so far

- Event-driven programming = programming without a call stack
- Lightweight, explicit task management
- More flexibility, but more responsibility (inversion of control)
- What does the added flexibility buy us?

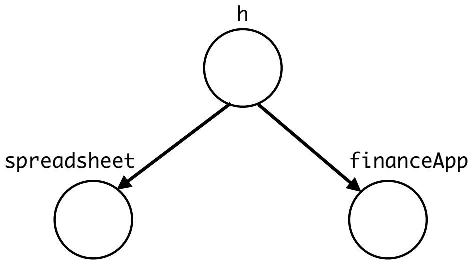
Modularity

- Synchronous (call/return) communication introduces strong temporal coupling
- May lead to interference between independent tasks
- Event loops can make tasks more composable

Concurrency among Strangers
Miller, Tribble and Shapiro
In Symposium on Trustworthy global computing, LNCS Vol 3705, pp. 195-229, 2005

Example: Listeners

```
StatusHolder h = new StatusHolder(state);
h.addListener(spreadsheet);
h.addListener(financeApp);
```



```
void statusChanged(Object s) {
  // update cell
}
```

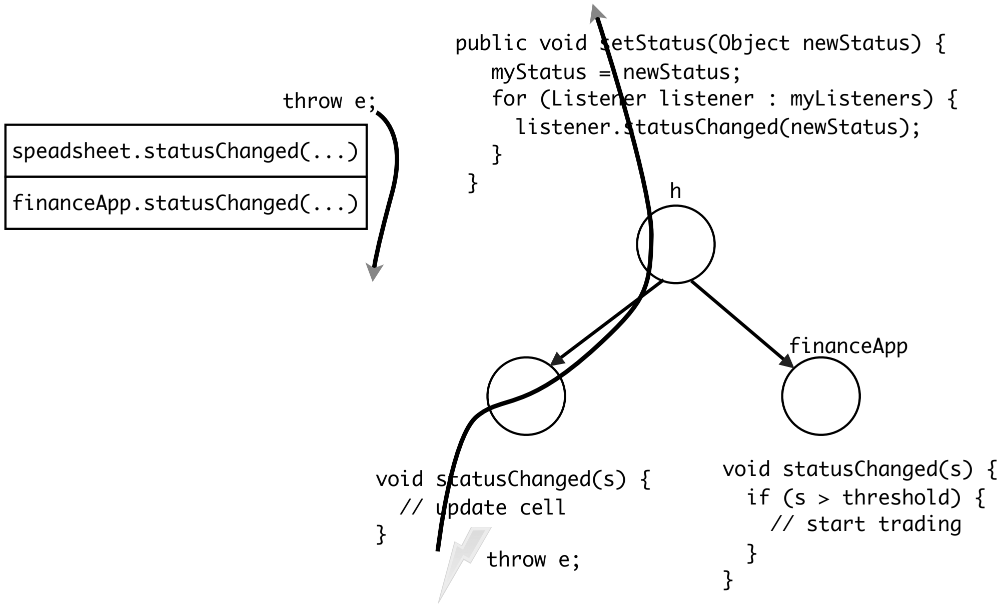
```
void statusChanged(Object s) {
  if (s > threshold) {
    // start trading
  }
}
```

Sequential Example

```
public class StatusHolder {
  private Object myStatus;
  private final ArrayList<Listener> myListeners = new ArrayList();

  public StatusHolder(Object status) {
    myStatus = status;
  }

  public void addListener(Listener newListener) {
    myListeners.add(newListener);
  }

  public Object getStatus() { return myStatus; }
  public void setStatus(Object newStatus) {
    myStatus = newStatus;
    for (Listener listener : myListeners) {
      listener.statusChanged(newStatus);
    }
  }
}
```

Sequential updating of listeners

Aborting the Wrong Task



```
public void setStatus(Object newStatus) {
   myStatus = newStatus;
   for (Listener listener : myListeners) {
     listener.statusChanged(newStatus);
   }
 }
```

```
throw e;
```

speadsheet.statusChanged(...)
financeApp.statusChanged(...)

```
void statusChanged(s) {
  // update cell
}
```

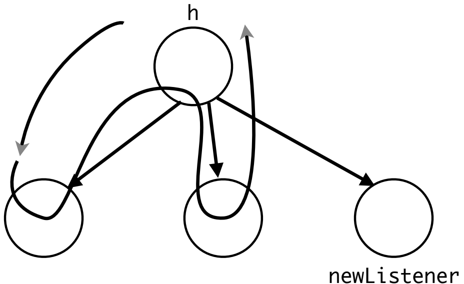
```
throw e;
```

```
void statusChanged(s) {
  if (s > threshold) {
    // start trading
  }
}
```

Nested subscription

```
public void addListener(Listener newListener) {
  myListeners.add(newListener);
}
public void setStatus(Object newStatus) {
  myStatus = newStatus;
  for (Listener listener : myListeners) {
    listener.statusChanged(newStatus);
  }
}
```

myListeners.add(newListener)
speadsheet.statusChanged(...)
financeApp.statusChanged(...)

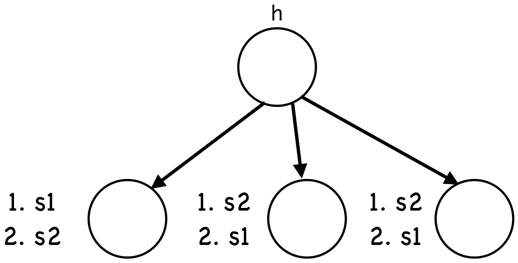


```
void statusChanged(s) {
  ...
  h.addListener(newListener)
}
```

Nested publication

```
public void setStatus(Object newStatus) {
  myStatus = newStatus;
  for (Listener listener : myListeners) {
    listener.statusChanged(newStatus);
  }
}
```

l1.statusChanged(s2)
l2.statusChanged(s2)
l3.statusChanged(s2)
h.setStatus(s2)
l1.statusChanged(s1)
l2.statusChanged(s1)
l3.statusChanged(s1)



```
void statusChanged(s1) {
  ...
  h.setStatus(s2)
}
```

Concurrent StatusHolder

```
public class StatusHolder {
  private Object myStatus;
  private final ArrayList<Listener> myListeners = new ArrayList();

  public StatusHolder(Object status) {
    myStatus = status;
  }

  public void addListener(Listener newListener) {
    myListeners.add(newListener);
  }
  public Object getStatus() { return myStatus; }
  public void setStatus(Object newStatus) {
    myStatus = newStatus;
    for (Listener listener : myListeners) {
      listener.statusChanged(newStatus);
    }
  }
}
```

Concurrent updates to instance variables: inconsistency

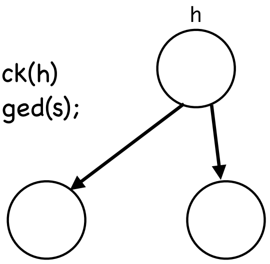
Synchronized StatusHolder

```
public class StatusHolder {
  private Object myStatus;
  private final ArrayList<Listener> myListeners = new ArrayList();

  public StatusHolder(Object status) {
    myStatus = status;
  }

  public synchronized void addListener(Listener newListener) {
    myListeners.add(newListener);
  }
  public synchronized Object getStatus() { return myStatus; }
  public synchronized void setStatus(Object newStatus) {
    myStatus = newStatus;
    for (Listener listener : myListeners) {
      listener.statusChanged(newStatus);
    }
  }
}
```

Synchronized StatusHolder

```
public synchronized void setStatus(Object newStatus) {
  myStatus = newStatus;
  for (Listener listener : myListeners) {
    listener.statusChanged(newStatus);
  }
}
```

publisher: h.setStatus(s); //lock(h)
publisher: listener.statusChanged(s);
pubisher: wait on o's lock



subscriber: has locked o
subscriber: h.addListener(l);
subscriber: wait on h's lock

```
void statusChanged(s) {
  synchronized(o) {
    ...
  }
}
```

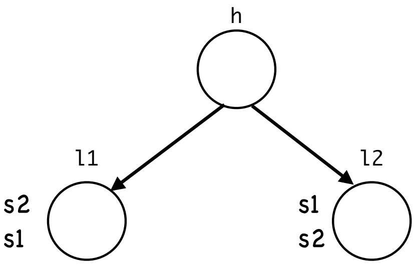
Deadlock

"Improved" Synchronized StatusHolder

```
public class StatusHolder {
  ...

  public void setStatus(Object newStatus) {
    ArrayList<Listener> listeners;
    synchronized(this) {
      myStatus = newStatus;
      listeners = (ArrayList<Listener>) myListeners.clone();
    }
    for (Listener listener : listeners) {
      listener.statusChanged(newStatus);
    }
  }
}
```

May still deadlock, still race conditions

```
public void setStatus(Object newStatus) {
  ArrayList<Listener> listeners;
  synchronized(this) {
    myStatus = newStatus;
    listeners = (ArrayList<Listener>) myListeners.clone();
  }
  for (Listener listener : listeners) {
    listener.statusChanged(newStatus);
  }
}
```

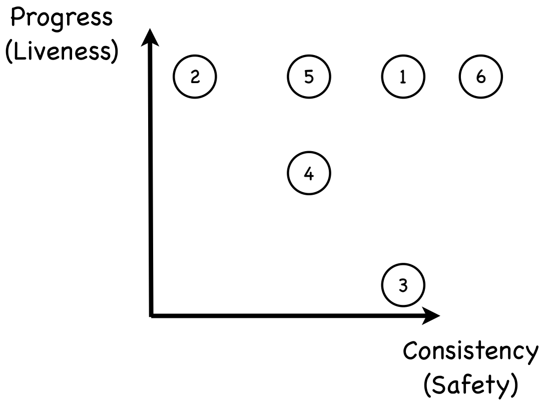


T1: h.setStatus(s1);
T2: h.setStatus(s2);
T1: lock h, s = s1
T2: wait on h
T1: unlock h
T2: lock h, s = s2
T2: unlock h
T2: l1.statusChanged(s2)
T1: l1.statusChanged(s1)
T1: l2.statusChanged(s1)
T2: l2.statusChanged(s2)

No deadlock, same race conditions

```
public void setStatus(Object newStatus) {
  ArrayList<Listener> listeners;
  synchronized(this) {
    myStatus = newStatus;
    listeners = (ArrayList<Listener>) myListeners.clone();
  }
  for (Listener listener : listeners) {
    new Thread(new Runnable() {
      public void run() {
        listener.statusChanged(newStatus);
      }
    }).start();
  }
}
```

Liveness vs Safety



1. Sequential StatusHolder
2. Sequential StatusHolder in concurrent world
3. Fully serialized StatusHolder
4. synchronized outside for-loop
5. new thread per listener
6. event loops

Two ways to execute tasks

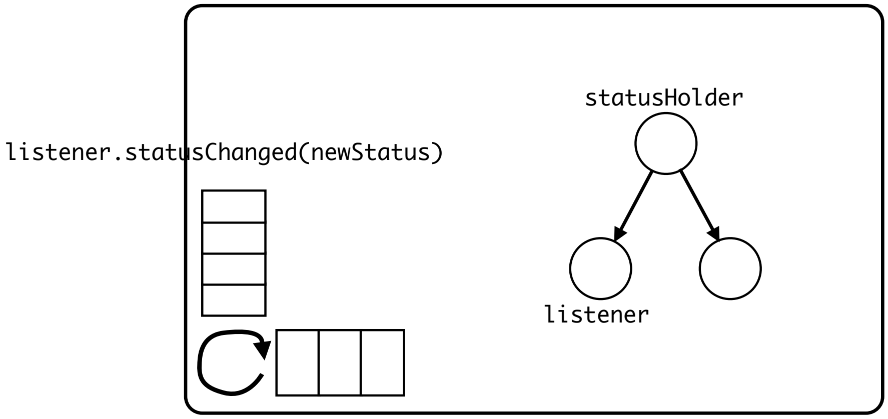
- Given a task X that needs to execute a task Y. Perform Y:
  - Immediately: stop X, do Y, continue with X
  - Eventually: put Y on TODO list, finish X, then start on Y
- Both compositions are easy in an event loop

Event Loop StatusHolder

```
public class StatusHolder {
  private Object myStatus;
  private final ArrayList<Listener> myListeners = new ArrayList();

  public StatusHolder(Object status) {
    myStatus = status;
  }
  public void addListener(Listener newListener) {
    myListeners.add(newListener);
  }
  public Object getStatus() { return myStatus; }

  public void setStatus(final Object newStatus) {
    myStatus = newStatus;
    for (final Listener listener : myListeners) {
      listener.getEventLoop().enqueue(new Runnable() {
        public void run() {
          listener.statusChanged(newStatus);
        }
      });
    }
  }
}
```

Deferred update

Event Loop

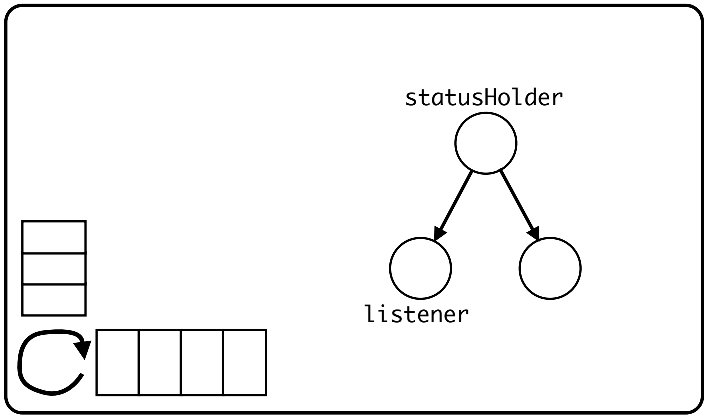


listener.statusChanged(newStatus)

statusHolder

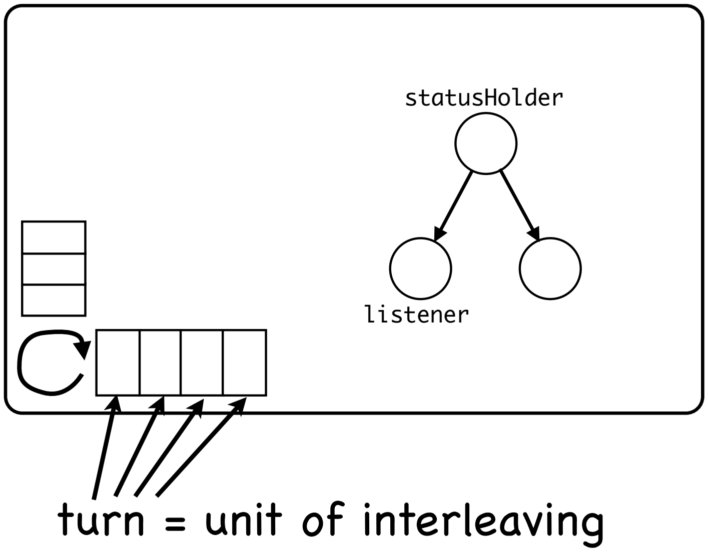
listener

Event Loop



```
listener.getEventLoop().enqueue(new Runnable() {
  public void run() {
    listener.statusChanged(newStatus);
  }
});
```

Turns



turn = unit of interleaving

Temporal Isolation

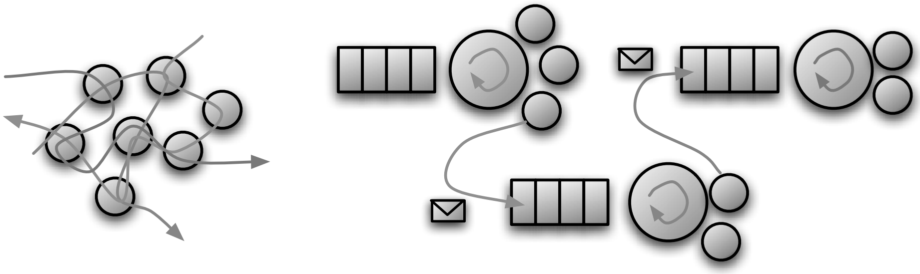
- Exceptions: do not abort later turns
- Nested subscriptions and publications: happen in later turns, after all current subscribers have been notified
- Scales to a concurrent environment without changes

Communicating Event Loops

From Event Loops to Communicating Event Loops

- Single Event Loop:
  - No true CPU concurrency
  - Not distributable
- Communicating Event Loops:
  - Exploit true CPU concurrency
  - Distributable

Communicating Event Loops



Safety Properties of Communicating Event Loops

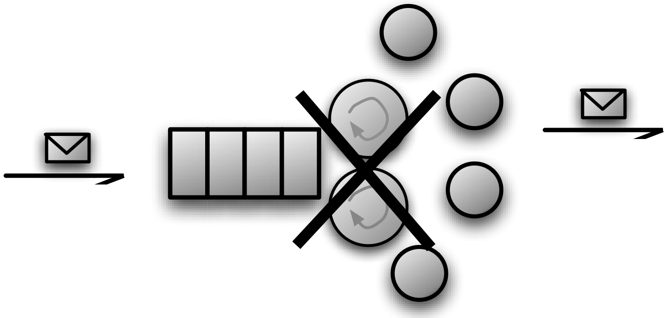
- Serial Execution: prevent race conditions within a single event loop
- Non-blocking Communication: ensures responsiveness, prevents deadlock
- Exclusive State Access: prevent race conditions between different event loops

---

Property #1: Serial Execution

- An event loop processes incoming events from its event queue one at a time (i.e. serially)
- Consequence: events are handled in mutual exclusion. An event handler cannot be preempted, so its state cannot become corrupted by interleaving actions.

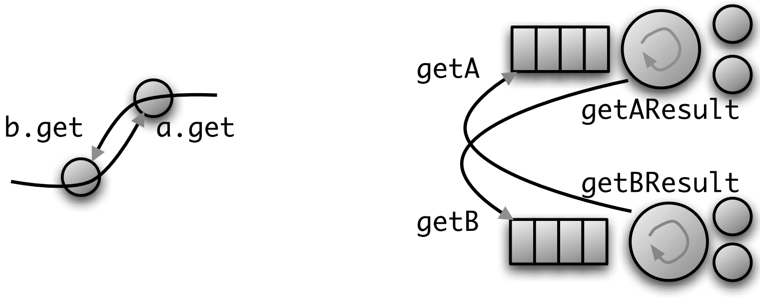
Property #1: Serial Execution



Property #2: Non-blocking Communication

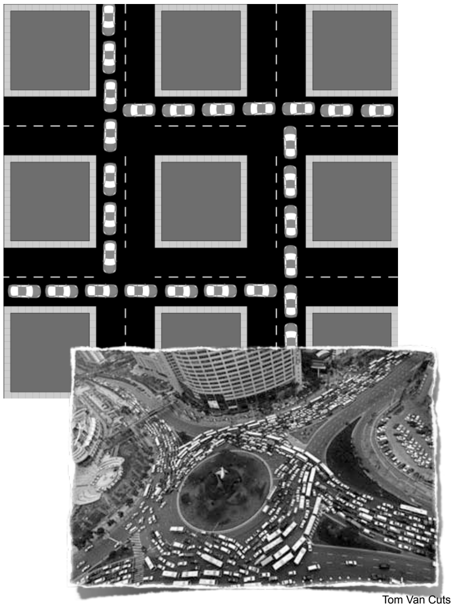
- An event loop never suspends its execution to wait for another event loop to finish a computation. Communication between event loops occurs strictly by means of asynchronous event notifications.
- Consequence: events loop cannot deadlock one another.
- Note: still prone to lost progress (e.g. if a certain event is never triggered)

Property #2: Non-blocking Communication



Gridlock

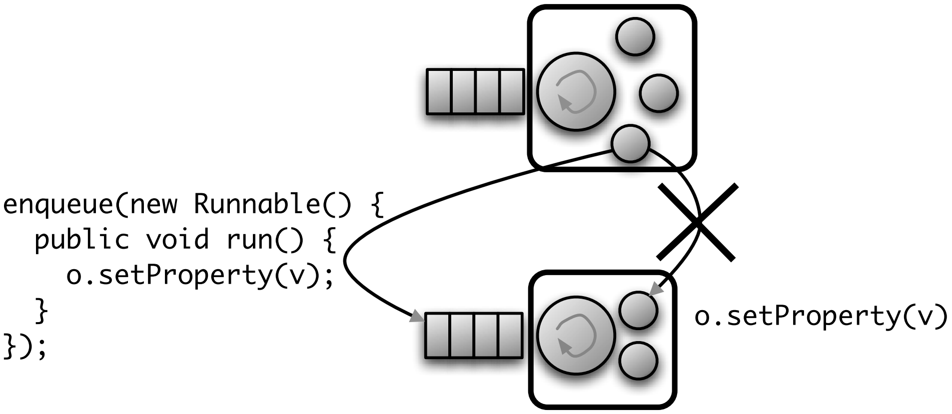
- When buffers are bounded, they can all become full
- An event loop may block on a full buffer => violates non-blocking communication



Property #3: Exclusive State Access

- Event loops never share synchronously accessible mutable state. An event loop has exclusive access to its mutable state.
- Consequence: no locking required, no race conditions on the mutable state
- Note: race conditions still possible at the event level (e.g. interleaving of 'read' and 'write' events)

Property #3: Exclusive State Access



```
enqueue(new Runnable() {
  public void run() {
    o.setProperty(v);
  }
});
```

```
o.setProperty(v)
```

Hidden forms of sharing

- Beware of implicit shared state:
  - Files
  - System calls
- Dedicated programming languages can enforce the properties

Race conditions

```
enqueue(new Runnable() {
  public void run() {
    point.setX(10);
  }
});
enqueue(new Runnable() {
  public void run() {
    point.setY(20);
  }
});
```

≠

```
enqueue(new Runnable() {
  public void run() {
    point.setX(10);
    point.setY(20);
  }
});
```

Return values

No return value needed:

```
listener.getEventLoop().enqueue(new Runnable() {
  public void run() {
    listener.statusChanged(newStatus);
  }
});
```

Schedule callable & use futures:

```
final String customerId = ...;
Future<Address> addressFuture = eventLoop.enqueue(
  new Callable<Address>() {
    public Address call() {
      return customerService.requestAddress(customerId);
    }
  });
```

Futures

- Recall: placeholder for a value to be computed in the future
- Traditionally blocking:

  ~~`Address a = addressFuture.get();`~~

- Violates non-blocking safety property
- Deadlock if the future should be resolved by the same event loop

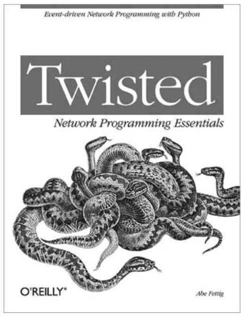
Non-blocking Futures

- Access value by registering an explicit continuation as a listener on the future
- Avoids deadlocks, ensures responsiveness

```
NBFuture<Address> addressFuture = el.schedule(callable);
addressFuture.register(new Resolver<Address>() {
  public void resolved(Address a) {
    // the future is now resolved to “a”
    ...
  }
});
```

always executed in a later turn

Communicating Event loops in the wild



Twisted Python ("reactors")

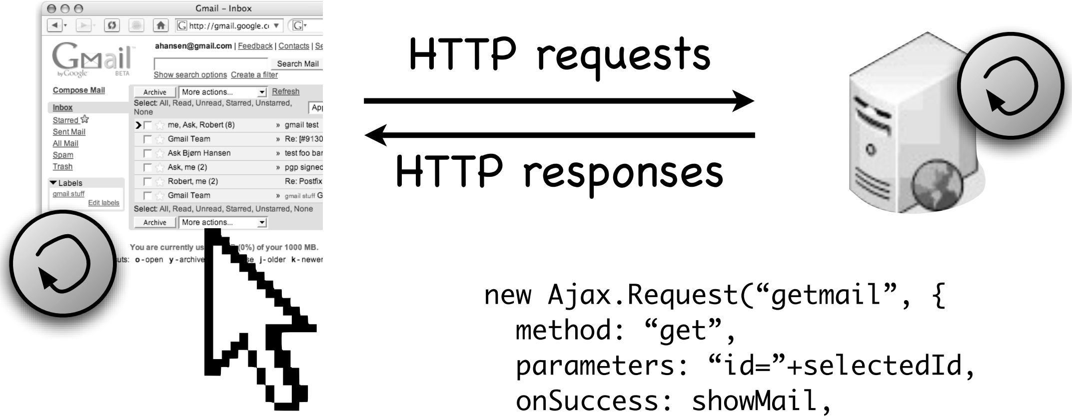
Waterken (web server)

Croquet ("islands")

E ("vats")

Javascript

Web 2.0 = communicating event loops



HTTP requests

HTTP responses

GUI events

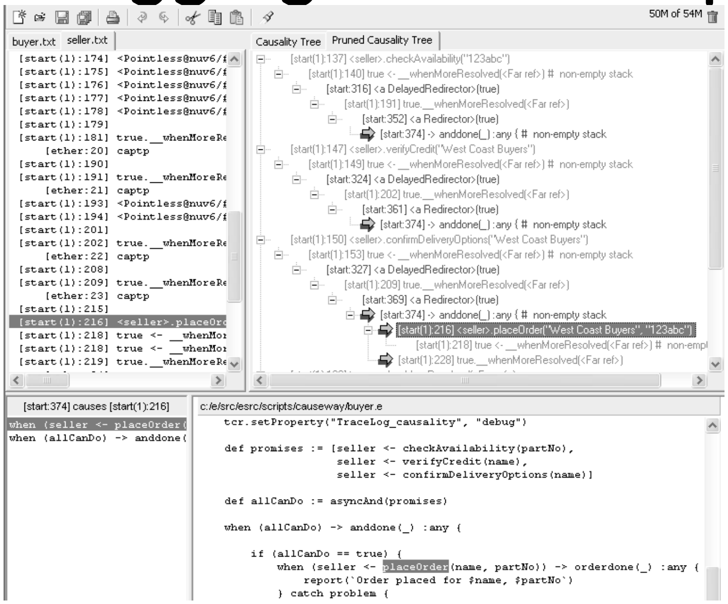
```
new Ajax.Request("getmail", {
  method: "get",
  parameters: "id="+selectedId,
  onSuccess: showMail,
  onFailure: showError
});
```

AJAX

Debugging Event Loops

- Causeway: post-mortem distributed debugger
- Event loops generate trace logs
- Visual inspection of trace logs
- Support for debugging a distributed conversation (tracing causality of messages)

Debugging Event Loops



```
[start(1):174] <Pointless@nuv6/
[start(1):175] <Pointless@nuv6/
[start(1):176] <Pointless@nuv6/
[start(1):177] <Pointless@nuv6/
[start(1):178] <Pointless@nuv6/
[start(1):179]
[start(1):181] true.__whenMoreRe
    [ether:20] captp
[start(1):190]
[start(1):191] true.__whenMoreRe
    [ether:21] captp
[start(1):193] <Pointless@nuv6/
[start(1):194] <Pointless@nuv6/
[start(1):201]
[start(1):202] true.__whenMoreRe
    [ether:22] captp
[start(1):208]
[start(1):209] true.__whenMoreRe
    [ether:23] captp
[start(1):215]
[start(1):216] <seller>.placeOrc
[start(1):218] true <- __whenMor
[start(1):218] true <- __whenMor
[start(1):219] true.__whenMoreRe
```

```
[start(1):137] <seller>.checkAvailability("123abc")
  [start(1):140] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
    [start:316] <a DelayedRedirector>(true)
      [start(1):191] true.__whenMoreResolved(<Far ref>)
        [start:352] <a Redirector>(true)
          [start:374] -> anddone(_) :any { #  non-empty stack
[start(1):147] <seller>.verifyCredit("West Coast Buyers")
  [start(1):149] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
    [start:324] <a DelayedRedirector>(true)
      [start(1):202] true.__whenMoreResolved(<Far ref>)
        [start:361] <a Redirector>(true)
          [start:374] -> anddone(_) :any { #  non-empty stack
[start(1):150] <seller>.confirmDeliveryOptions("West Coast Buyers")
  [start(1):153] true <- __whenMoreResolved(<Far ref>) #  non-empty stack
    [start:327] <a DelayedRedirector>(true)
      [start(1):209] true.__whenMoreResolved(<Far ref>)
        [start:369] <a Redirector>(true)
          [start:374] -> anddone(_) :any { #  non-empty stack
            [start(1):216] <seller>.placeOrder("West Coast Buyers", "123abc")
              [start(1):218] true <- __whenMoreResolved(<Far ref>) #  non-empl
            [start(1):228] true.__whenMoreResolved(<Far ref>)
```

[start:374] causes [start(1):216]

```
when (seller <- placeOrder(
when (allCanDo) -> anddone(
```

c:/e/src/esrc/scripts/causeway/buyer.e

```
    tcr.setProperty("TraceLog_causality", "debug")

    def promises := [seller <- checkAvailability(partNo),
                     seller <- verifyCredit(name),
                     seller <- confirmDeliveryOptions(name)]

    def allCanDo := asyncAnd(promises)

    when (allCanDo) -> anddone(_) :any {

        if (allCanDo == true) {
            when (seller <- placeOrder(name, partNo)) -> orderdone(_) :any {
                report(`Order placed for $name, $partNo`)
            } catch problem {
```

http://www.erights.org/elang/tools/causeway

Communicating Event Loops: advantages

- Event handlers run without preemption (i.e. in a single turn)
- No synchronously accessible shared state => no race conditions on mutable state
- Strictly asynchronous communication => no deadlocks

Communicating Event Loops: disadvantages

- Still race conditions across turns
- Future listener is still an explicit form of continuation => stack ripping
- Conditional synchronization is cumbersome (future resolution must be postponed manually)

Summary

- Explicit unit of interleaving (turn) => tasks within event loops are composable
- Explicit ownership of state (objects) => event loops themselves are composable
- Model scales to a distributed setting (asynchrony hides latency)

Communicating Event Loops
+
Objects

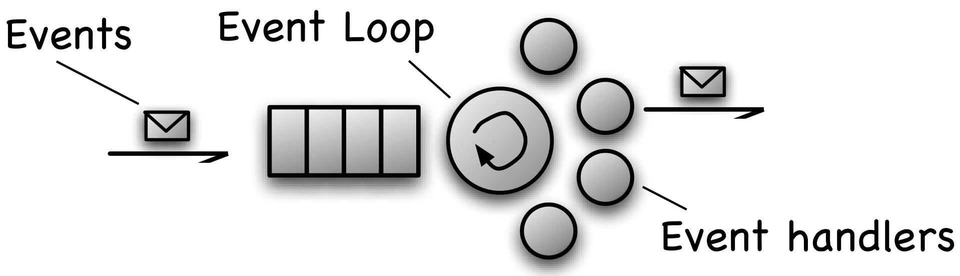
Communicating Event Loops + OOP

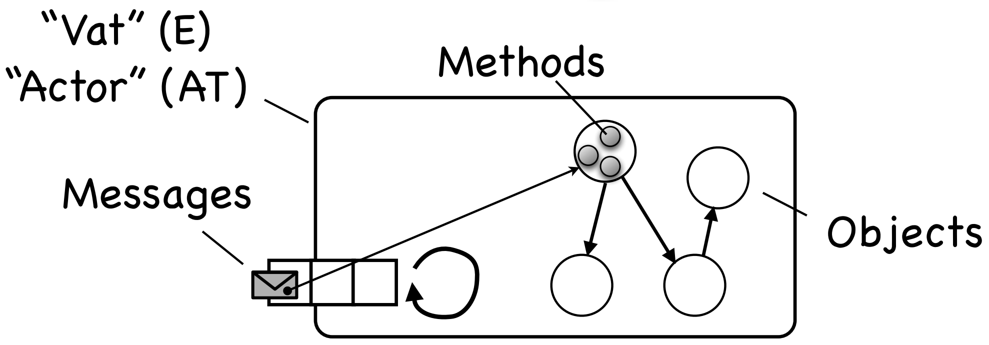
- Getting rid of the boilerplate code
- Event handler = method (or function)
- Event = (asynchronously sent) message
- Event sources and sinks = objects
- Same properties as before

Event Loop Languages

- E (Miller et al., 1998)
- AmbientTalk (Van Cutsem et al., 2006)
- AsyncObjects Framework for Java (Plotnikov, 2007)
- Newspeak (?) (Bracha, 2007)

Event Loops + OOP



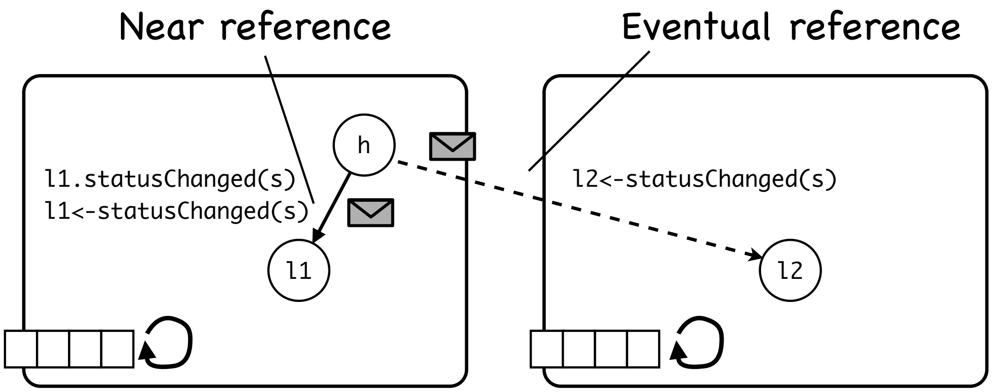


StatusHolder in AmbientTalk

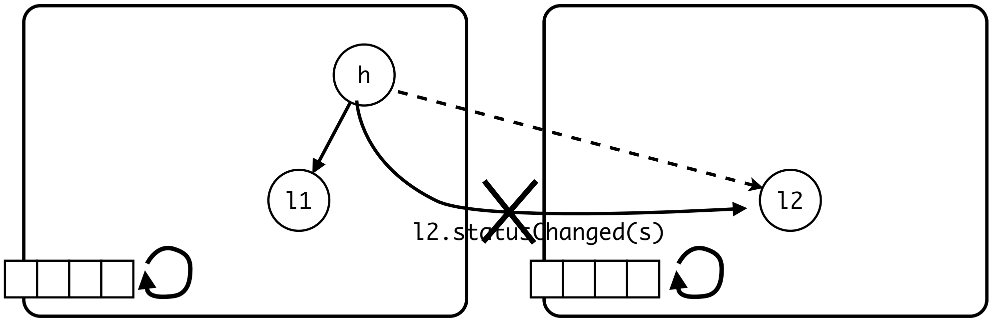
```
def makeStatusHolder(myStatus) {
  def myListeners := [];
  object: {
    def addListener(newListener) {
      myListeners.append(newListener);
    };
    def getStatus() { myStatus };
    def setStatus(newStatus) {
      myStatus := newStatus;
      myListeners.each: { |listener|
        listener<-statusChanged(newStatus)
      }
    };
  }
}
```

Eventual
(asynchronous) send

Communicating Event Loops + OOP



Communicating Event Loops + OOP



No client-side synchronization

- Instead: invoked methods define synchronization boundaries (executed in a single turn)

not synchronized

```
point<-setX(10);
point<-setY(20);
```

⇒

synchronized

```
point<-move(10,20);

// in point
def move(dx,dy) {
  self.setX(dx);
  self.setY(dy);
}
```

Return values

- Eventual sends return non-blocking futures
- Synonyms: promises (E), deferreds (Twisted)
- "when" statement to access a future's value:

```
def processDelivery(order) {
  def f := customerService<-requestAddress(order.customerId);
  when: f becomes: { |address|
    courier<-ship(order, address)
  }
}
```

Return values

- Eventual sends return non-blocking futures
- Synonyms: promises (E), deferreds (Twisted)
- "when" statement to access a future's value:

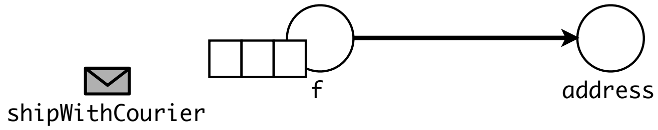
```
def processDelivery(order) {
  def f := customerService<-requestAddress(order.customerId);
  when: f becomes: { |address|
    courier<-ship(order, address)
  }
}
```

always executed in a later turn

Data Flow Synchronization

- May send eventual messages to a future
- Messages are buffered and forwarded later

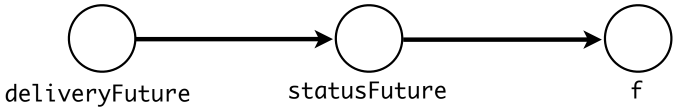
```
def processDelivery(order) {
  def f := customerService<-requestAddress(order.customerId);
  f<-shipWithCourier(order, courier);
}

// in Address
def shipWithCourier(order, courier) {
  courier<-ship(order, self);
}
```



Data Flow Synchronization

- Resolving a future with another future creates a dependency link

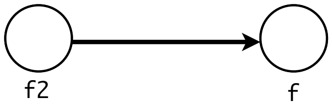
```
def processDelivery(order) {
  def f := customerService<-requestAddress(order.customerId);
  f<-shipWithCourier(order, courier); // returns statusFuture
}

pendingDeliveries.add(order);
def deliveryFuture := deliveryService<-processDelivery(order);
when: deliveryFuture becomes: { |status|
  pendingDeliveries.update(order, status);
}
```



Ruining Futures

- Separate ‘errback’ for exceptions
- Future ruining is “contageous”

```
def processDelivery(order) {
  def f := customerService<-requestAddress(order.customerId);
  when: f becomes: { |address|
     courier<-ship(order, address)
  } catch: AddressNotFound using: { |e|
    // deal with unknown address
  }
}
```



In Practice

- Programming Languages: E, AmbientTalk
- Roll your own event loop framework using threads, queues & proxies
- Or use existing libraries:
  - ActiveObjects (Java)
  - Twisted (Python)

AsyncObjects

http://asyncobjects.sourceforge.net

- Objects assigned to Vats
- Vat events executed by VatRunners
- Only proxies for objects may cross vat boundaries
- Proxies dispatch async calls to Vats

Asynchronous Components

```
public class StatusHolder extends AsyncUnicastServer<AStatusHolder>
                          implements AStatusHolder {
   private Object myStatus;
   private final ArrayList<AListener> myListeners = new ArrayList();

   public StatusHolder(Object status) { myStatus = status; }

   public void addListener(AListener newListener) {
      myListeners.add(newListener);
   }

   public Promise<Object> getStatus() { return new Promise<Object>(myStatus); }

   public void setStatus(Object newStatus) {
      myStatus = newStatus;
      for (AListener listener : myListeners) {
         listener.statusChanged(newStatus);
      }
   }
}
```

asynchronous interfaces

read: listener<-statusChanged(...)

cf. Java RMI

Asynchronous Interfaces

```
public interface AStatusHolder extends AsyncObject {
   public void setStatus(Object status);
   public Promise<Object> getStatus();
   public void addListener(AListener l);
}


public interface AListener extends AsyncObject {
   public void statusChanged(Object status);
}
```

return type =
void | Promise<T>

Asynchronous Interfaces

in vat A

```
StatusHolder h = new StatusHolder(init);
AStatusHolder proxy = h.export();
```

returns “eventual reference”
as a proxy

in vat B

```
Application a = new Application();
AListener l = a.export();
proxy.addListener(l);
```

read: proxy<-addListener(l)

Creating Vats

in vat A

```
StatusHolder h = new StatusHolder(init);
AStatusHolder proxy = h.export();
```

Creating Vats

```
VatRunner r = new SingleThreadRunner();
Vat vatA = r.newVat(“Vat A”);
vatA.enqueue(new Runnable() {
  public void run() {
    StatusHolder h = new StatusHolder(init);
    AStatusHolder proxy = h.export();
  }
});
```

Pitfalls

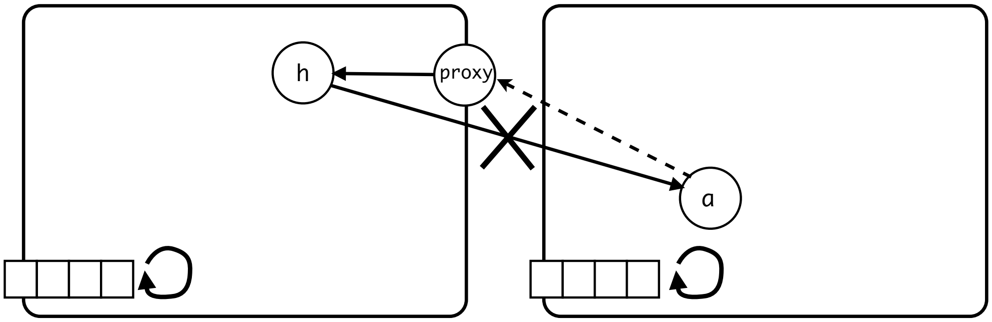
- Libraries usually cannot strictly enforce the event loop properties that ensure safety!
    - Exclusive State Access: not enforced that a vat-local object is always accessed via its 'eventual' proxy
    - PL implementation automatically creates proxy when object crosses vat boundary

---

Pitfalls

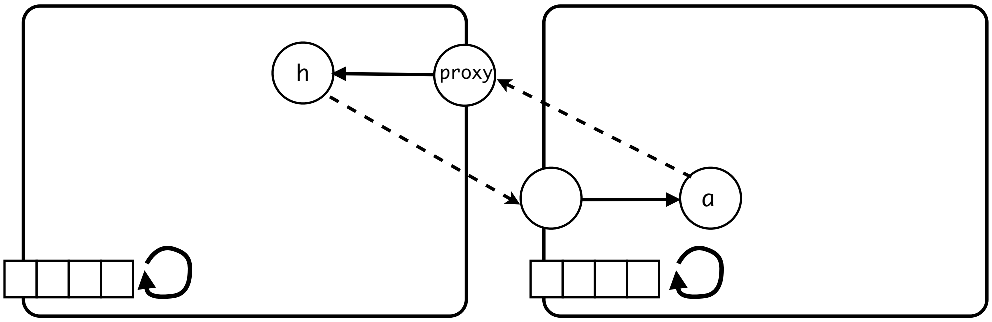
```
Application a = new Application();
proxy.addListener(a);
```

Bug!



Pitfalls

```
Application a = new Application();
proxy.addListener(a.export());
```



Inconveniences

- Lack of "<-" message passing operator makes asynchronous calls implicit
- Lots of closures: boilerplate code
- Very dependent on host language

```
when: calc<-add(a,b) becomes: { |result|
  println(result);
}
```

```
(new AsyncAction<Void>() {
  public Promise<Void> run() {
    new When<int,Void>(calc.add(a,b)) {
      public Void resolved(int result) {
        System.out.println(result);
        return null;
      }
    }
  }
}).startInCurrentThread();
```

Summary

- Event Loops & OOP go hand in hand
  - event = asynchronous message
- Language can enforce safety properties (especially ownership boundaries of event loops)
- Stack ripping manageable thanks to closures
- No client-side synchronization (to achieve atomic changes across turns)

Concluding Remarks

Characterizing Concurrency Control

- Communication via shared state
  - Threads
- Communication via message passing
  - Actors
  - Event Loops

Characterizing Concurrency Control

- Serializability: what is the smallest unit of non-interleaved operation?
  - Threads: memory access/single low-level instruction
  - Events: event handlers
  - Databases and STM: transactions

Characterizing Concurrency Control

- Mutual exclusion: what mechanisms are provided to eliminate unwanted interleavings?
  - Threads: locks, condition variables
  - Events: explicit yield points, futures
  - Databases and STM: conflict detection, rollback & retry

Threads do not compose

- No explicit unit of interleaving: threads can be preempted at any point in time
- No explicit ownership of state: any thread can freely modify any mutable data it can access

Event loops compose

- Explicit unit of interleaving (‘turn’): event handlers are never preempted
- Explicit ownership of state: state is owned by a single event loop (but can still be shared!)

Bibliography

Concurrent Object-oriented Programming (Gul Agha)
In Communications of the ACM, Vol 33 (9), p. 125, 1990

Why threads are a bad idea (for most purposes) (John Ousterhout)
Invited Talk at the 1996 USENIX Technical Conference

Cooperative Task Management without Manual Stack Management (Adya et al.)
Proceedings 2002 USENIX Technical Conference

Why events are a bad idea (for high-concurrency servers) (von Behren, Condit and Brewer)
Proceedings of the 9th USENIX Conference on Hot Topics in Operating Systems, 2003

Concurrency among Strangers (Miller, Tribble and Shapiro)
In Symposium on Trustworthy global computing, LNCS Vol 3705, pp. 195-229, 2005

Programming without a call stack (Gregor Hohpe)
Available online: www.enterpriseintegrationpatterns.com 2006

The Problem with Threads (Edward Lee)
IEEE Computer, Vol. 39, No. 5, pp. 33-42, May 2006