Concurrent Programming

Prof. O. Nierstrasz

Wintersemester 1999/2000

Table of Contents

1. Concurrent Programming

Lecturer:	Prof. Oscar Nierstrasz Schützenmattstr. 14/103, Tel. 631.4618, `oscar@iam.unibe.ch`
Assistants:	Dr. Markus Lumpe, Franz Achermann
WWW:	`www.iam.unibe.ch/~scg/Lectures/`

Text:

- ❑ D. Lea, Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley, 1996

Other Sources:

- ❑ D. Lea, Online Supplement to Concurrent Programming in Java, `http://gee.cs.oswego.edu/dl/cpj/index.html`
- ❑ N. Carriero, D. Gelernter, How to Write Parallel Programs: a First Course, MIT Press, Cambridge, 1990.
- ❑ A. Burns, G. Davies, Concurrent Programming, Addison-Wesley, 1993
- ❑ J. Magee, J. Kramer, Concurrency: State Models & Java Programs, Wiley, 1999

Schedule

1.	01.11	Introduction — Concurrency and Java
2.	08.11	Safety
3.	15.11	Liveness
4.	22.11	Lab session
5.	29.11	Asynchronous Methods
6.	06.12	Fine-grained Synchronization
7.	13.12	Lab session
8.	20.12	Architectural Styles for Concurrency
9.	10.01	Concurrent Programming Approaches
10.	17.01	Petri Nets
11.	24.01	Pi Calculus (I)
12.	31.01	Pi Calculus (II)
13.	07.02	Piccola — a language for scripting concurrent components

Introduction

Overview

- ❑ Concurrency and Parallelism
- ❑ Applications of Concurrency
- ❑ Limitations
  - ☞ safety, liveness, non-determinism ...
- ❑ Approach
  - ☞ idioms, patterns and architectural styles
- ❑ Java and concurrency

Concurrency and Parallelism

"A sequential program specifies sequential execution of a list of statements; its execution is called a process.

A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes."

A concurrent program can be executed by:

1.	Multiprogramming:	processes share one or more processors
2.	Multiprocessing:	each process runs on its own processor but with shared memory
3.	Distributed processing:	each process runs on its own processor connected by a network to others

Assume only that all processes make positive finite progress.

Applications of Concurrency

There are many good reasons to build concurrent programs:

- ❑ Reactive programming
  - ☞ minimize response delay; maximize throughput
- ❑ Real-time programming
  - ☞ process control applications
- ❑ Simulation
  - ☞ modelling real-world concurrency
- ❑ Parallelism
  - ☞ exploit multiple CPUs for number-crunching; exploit parallel algorithms
- ❑ Distribution
  - ☞ coordinate distributed services

Limitations

But concurrent applications introduce complexity:

- ❑ Safety
  - ☞ synchronization mechanisms are needed to maintain consistency
- ❑ Liveness
  - ☞ special techniques may be needed to guarantee progress
- ❑ Non-determinism
  - ☞ debugging is harder because results may depend on “race conditions”
- ❑ Communication complexity
  - ☞ communicating with a thread is more complex than a method call
- ❑ Run-time overhead
  - ☞ thread construction, context switching and synchronization take time

Atomicity

Programs P1 and P2 execute concurrently:

```
            { x = 0 }
P1:         x := x+1
P2:         x := x+2
            { x = ? }
```

What are possible values of x after P1 and P2 complete?

What is the intended final value of x?

Synchronization mechanisms are needed to restrict the possible interleavings of processes so that sets of actions can be seen as atomic.

Mutual exclusion ensures that statements within a critical section are treated atomically.

Expressing Concurrency

Notations for expressing concurrent computation must address:

1. **Process Creation:** how is concurrent execution specified?
2. **Communication:** how do processes communicate?
3. **Synchronization:** how is consistency maintained?

Safety and Liveness

There are two principal difficulties in implementing concurrent programs:

Safety — ensuring consistency:

- ☞ Mutual exclusion — shared resources must be updated atomically
- ☞ Condition synchronization — operations may need to be delayed if shared resources are not in an appropriate state (e.g., read from empty buffer)

i.e., “Nothing bad happens”

Liveness — ensuring progress:

- ☞ No Deadlock — some process can always access a shared resource
- ☞ No Starvation — all processes can eventually access shared resources

i.e., “Something good happens”

Idioms, Patterns and Architectural Styles

Idioms, patterns and architectural styles express best practice in resolving common design problems.

- ❑ Idioms
  - ➪ “a low-level pattern specific to a programming language”
    — or more generally: “an implementation technique”
- ❑ Design patterns
  - ➪ “a commonly-recurring structure of communicating components that solves a general design problem within a particular context”
- ❑ Architectural patterns (styles)
  - ➪ “a fundamental structural organization schema for software systems”

— cf. Buschmann et al., Pattern-Oriented Software Architecture, pp. 12-14

Java

Language design influenced by existing OO languages (C++, Smalltalk ...):

- Strongly-typed, concurrent, pure object-oriented language
- Syntax, type model influenced by C++
- Single-inheritance but multiple subtyping
- Garbage collection

Innovation in support for network applications:

- Standard API for language features, basic GUI, IO, concurrency, network
- Compiled to bytecode; interpreted by portable abstract machine
- Support for native methods
- Classes can be dynamically loaded over network
- Security model protects clients from malicious objects

Java applications do not have to be installed and maintained by users

Threads

A Java Thread has a `run` method defining its behaviour:

```
class SimpleThread extends Thread {
  public SimpleThread(String str) {
    super(str);                      // Call Thread constructor
  }
  public void run() {                // What the thread does
    for (int i = 0; i < 10; i++) {
      System.out.println(i + " " + getName());
      try {
        sleep((int)(Math.random() * 1000));
      } catch (InterruptedException e) { }
    }
    System.out.println("DONE! " + getName());
  }
}
```

Threads ...

A Thread’s run method is never called directly but is executed when the Thread is started:

```
class TwoThreadsTest {
  public static void main (String[] args) {
    // Instantiate a Thread, then start it:
    new SimpleThread("Jamaica").start();
    new SimpleThread("Fiji").start();
  }
}
```

Running the TwoThreadsTest

```
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
2 Fiji
3 Jamaica
3 Fiji
4 Jamaica
4 Fiji
5 Jamaica
6 Jamaica
5 Fiji
6 Fiji
7 Fiji
7 Jamaica
8 Jamaica
9 Jamaica
8 Fiji
DONE! Jamaica
9 Fiji
DONE! Fiji
```

In this implementation of Java, the execution of the two threads is interleaved.

This is not guaranteed for all implementations!

✎ Why are the output lines never garbled?

E.g.

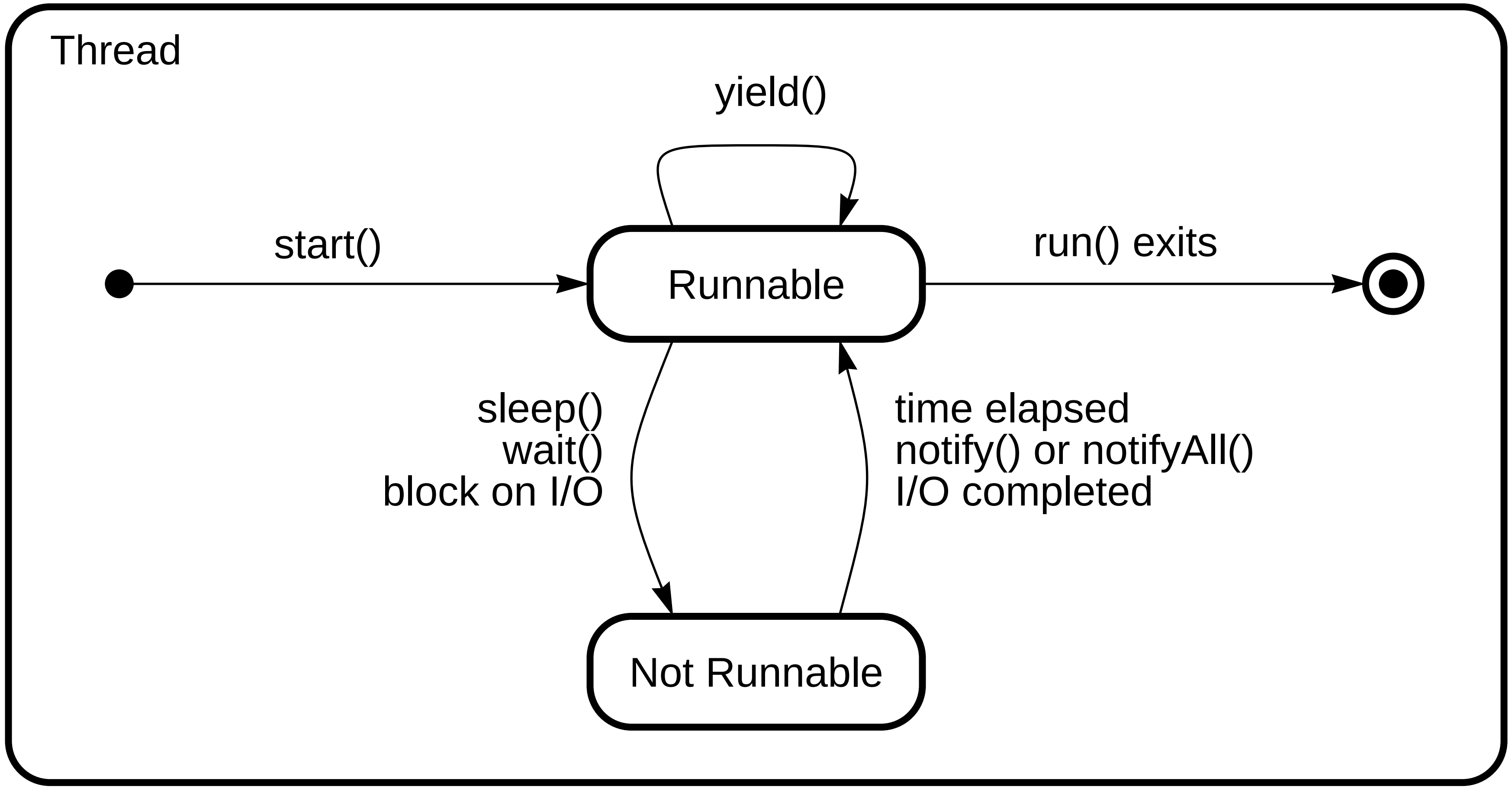
```
00 JaFimajicai
...
```

java.lang.Thread

The Thread class encapsulates all information concerning running threads of control:

```
public class java.lang.Thread
  extends java.lang.Object implements java.lang.Runnable
{
  public Thread();
  public Thread(Runnable target);
  public Thread(Runnable target, String name);
  public Thread(String name);
...
  public static void sleep(long millis)
                           throws InterruptedException;
  public static void yield();
...
  public final String getName();
  public void run();
  public synchronized void start();
...
}
```

Transitions between Thread States



java.lang.Runnable

Since multiple inheritance is not supported, it is not possible to inherit from both `Thread` and from another class providing useful behaviour (like `Applet`).

In these cases it is sufficient to define a class that implements the Runnable interface, and to call the Thread constructor with an instance of that class as a parameter:

```
public interface java.lang.Runnable
{
  public abstract void run();
}
```

Creating Threads

A `Clock` object updates the time as an `Applet` with its own `Thread`:

```
import java.awt.Graphics;
import java.util.Date;
public class Clock    extends java.applet.Applet
                      implements Runnable
{
  Thread clockThread = null;
  public void start() {
    if (clockThread == null) {
      clockThread = new Thread(this, "Clock");
                 // NB: creates its own thread
      clockThread.start();
    }
  }
  ...
```

Creating Threads ...

```
  ...
  public void run() {
    // terminates when clockThread is set to null in stop()
    while (Thread.currentThread() == clockThread) {
      repaint();
      try { clockThread.sleep(1000); }
      catch (InterruptedException e){ }
    }
  }
  public void paint(Graphics g) {
    Date now = new Date();
    g.drawString(now.getHours() + ":" + now.getMinutes()
                          + ":" + now.getSeconds(), 5, 10);
  }
  public void stop() { clockThread = null; }
}
```

Synchronization

Without synchronization, an arbitrary number of threads may run at any time within the methods of an object.

- ➪ Methods cannot assume that the class invariant holds (since another method may be running)
- ➪ There is no way to guarantee that a method will ensure its post-condition

A simple solution is to consider a method to be a critical section which locks access to the object while it is running.

This works as long as methods cooperate in locking and unlocking access!

Synchronization ...

One can either declare an entire method to be synchronized with other synchronized methods of an object:

```
public class PrintStream extends FilterOutputStream {
    ...
    public synchronized void println(String s);
    public synchronized void println(char c);
    ...
}
```

or an individual block within a method may be synchronized with respect to some object:

```
synchronized (resource) { // Lock resource before using it
    ...
}
```

wait and notify

Sometimes threads must be delayed until a resource is in a suitable state:

```
class Slot {                                               // a one-slot buffer
   private Object slotVal_;                                // initially null
   public synchronized void put(Object val) {              // put contents,
      while (slotVal_ != null) {                           // if there is room
         try { wait(); }                                   // otherwise wait
         catch (InterruptedException e) { }
      }
      slotVal_ = val;
      notifyAll();                                         // wake up waiting consumer
      return;
   }
   public synchronized Object get() {                      // return contents,
      Object rval;
      while (slotVal_ == null) {                           // if available
         try { wait(); }                                   // otherwise wait
         catch (InterruptedException e) { }
      }
      rval = slotVal_;
      slotVal_ = null;
      notifyAll();                                         // wake up waiting producer
      return rval;
   }
}
```

java.lang.Object

Unlike `synchronized`, `wait()` and `notify()` are methods rather than keywords:

```
public class java.lang.Object
{
   ...
   public final void wait() throws InterruptedException;

   public final void wait(long timeout)
                                    throws InterruptedException;
   public final void wait(long timeout, int nanos)
                                    throws InterruptedException;

   public final void notify();
   public final void notifyAll();
   ...
}
```

Summary

You Should Know The Answers To These Questions:

- ❑ What is the distinction between “concurrency” and “parallelism”?
- ❑ What are classical applications of concurrent programming?
- ❑ Why are concurrent programs more complex than sequential ones?
- ❑ What are “safety” and “liveness”? Give examples.
- ❑ How do you create a new thread in Java?
- ❑ What states can a Java thread be in? How does it change state?
- ❑ When should you declare a method to be `synchronized`?

Can You Answer The Following Questions?

- ✎ What is an example of a “race condition”?
- ✎ When will a concurrent program run faster than an equivalent sequential one? When will it be slower?
- ✎ What is the difference between deadlock and starvation?
- ✎ What happens if you call `wait` or `notify` outside a synchronized method or block?
- ✎ When is it better to use synchronized blocks rather than methods?

2. Safety

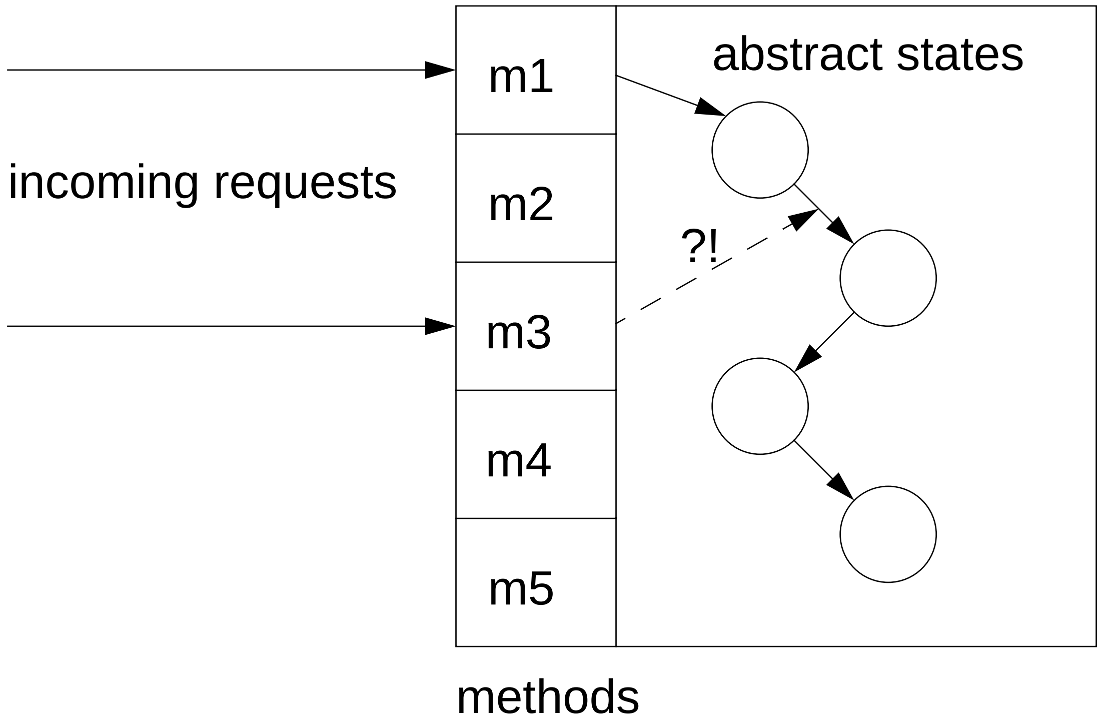
Overview

- ❑ Immutability:
  - ☞ avoid safety problems by avoiding state changes
- ❑ Full Synchronization:
  - ☞ dynamically ensure exclusive access
- ❑ Partial Synchronization:
  - ☞ restrict synchronization to “critical sections”
- ❑ Containment:
  - ☞ structurally ensure exclusive access

Safety problems

Objects must only be accessed when they are in a consistent state

☞ methods must maintain class (state and representation) invariants



Normally each method may assume the class invariant holds when it starts, (i.e., that the object is in a consistent state) and it must ensure it when it is done.

If methods interleave arbitrarily, an inconsistent state may be accessed, and the object may be left in a "dirty" state.

Immutable classes

Intent

Bypass safety issues by not changing an object’s state after creation.

Applicability

- ❑ When objects represent values of simple ADTs
  - ☞ colours (java.awt.Color), numbers (java.lang.Integer) and strings (java.lang.String)
- ❑ When classes can be separated into mutable and immutable versions
  - ☞ java.lang.String vs. java.lang.StringBuffer
- ❑ When updating by copying is cheap
  - ☞ “hello” + “ ” + “world” $\rightarrow$ “hello world”
- ❑ When multiple instances can represent the same value
  - ☞ i.e., two distinct copies of the integer 712 represent the same value

Immutability variants

Variants

- ❑ Stateless methods
  - ☞ methods that do not access an object’s state do not need to be synchronized (such methods can be declared `static`)
  - ☞ any temporary state should be purely local to the method
- ❑ Stateless objects
  - ☞ an object whose “state” is dynamically computed needs no synchronization
- ❑ “Hardening”
  - ☞ object becomes immutable after a mutable phase
  - ☞ be sure that object is exposed to concurrent threads only after hardening

Immutable classes — design steps

- ❑ Declare a class with instance variables that are never changed after construction.

```
class Relay {                // a helper for some Server class
  private final Server server_;

  Relay(Server s) {          // blank finals must be initialized
    server_ = s;             // in all constructors
  }

  void doIt() {
    server_.doIt();
  }
}
```

Design steps ...

- ❑ Especially if the class represents an immutable data abstraction (such as `String`), consider overriding `Object.equals` and `Object.hashCode`.
- ❑ Consider writing methods that generate new objects of this class. (e.g., `String` concatenation)
- ❑ Consider declaring the class as `final`.
- ❑ If only some variables are immutable, use synchronization or other techniques for the methods that are not stateless.

Fully Synchronized Objects

Intent

Maintain consistency by fully synchronizing all methods.At most one method will run at any point in time.

Applicability

- ❑ You want to eliminate all possible read/write and write/write conflicts, regardless of the context in which it the object is used.
- ❑ All methods can run to completion without waits, retries, or infinite loops.
- ❑ You do not need to use instances in a layered design in which other objects control synchronization of this class.
- ❑ You can avoid or deal with liveness failures, for example, by:
  - ☞ Exploiting partial immutability
  - ☞ Removing synchronization for accessors.
  - ☞ Removing synchronization in invocations.
  - ☞ Arranging per-method concurrency.

Full Synchronization — design steps

- ❑ Declare all methods as `synchronized`
  - ☞ Do not allow any direct access to state (i.e, no `public` instance variables; no methods that return references to instance variables).
  - ☞ Constructors cannot be marked as `synchronized` in Java. Use a synchronized block in case a constructor passes `this` to multiple threads.
  - ☞ Methods that access `static` variables must either do so via `static synchronized` methods or within blocks of the form `synchronized(getClass()) { ... }`.

Design steps ...

- ❑ Ensure that every `public` method exits leaving the object in a consistent state, even if it exits via an exception.
- ❑ Keep methods short so they can atomically run to completion. State-dependent actions must rely on balking:
  - ☞ Return failure (i.e., exception) to client if preconditions fail
  - ☞ If the precondition does not depend on state (e.g., just on the arguments), then no need to run check in synchronized code!
  - ☞ Provide public accessor methods so that clients can check conditions before making request!

Example: a BalkingBoundedCounter

A Bounded Counter holds a value between MIN and MAX.
If the preconditions for `inc()` or `dec()` fail, an exception is raised:

```
public class BalkingBoundedCounter {
  protected long count_ = BoundedCounter.MIN;
  public synchronized long value() { return count_; }
  public synchronized void inc()
    throws CannotIncrementException {
    if (count_ >= BoundedCounter.MAX)
      throw new CannotIncrementException();
    else
      ++count_;
  }
  public synchronized void dec() ... { ... }
}
```

✎ What safety problems would arise if this class were not fully synchronized?

Example: an ExpandableArray

This Expandable Array is a simplified variant of java.util.Vector:

```
import java.util.NoSuchElementException;
public class ExpandableArray {
  private Object[] data_;            // the elements
  private int size_;                 // the number of slots used
  public ExpandableArray(int cap) {
    data_ = new Object[cap];         // reserve some space
    size_ = 0;
  }
  public synchronized int size() { return size_; }
  public synchronized Object at(int i) // array indexing
    throws NoSuchElementException {
    if (i < 0 || i >= size_ )
      throw new NoSuchElementException();
    else
      return data_[i];
  }
  ...
```

Example ...

```
  public synchronized void append(Object x) { // add at end
    if (size_ >= data_.length) {     // need a bigger array
      Object[] olddata = data_;      // so increase ~50%
      data_ = new Object[3 * (size_ + 1) / 2];
      for (int i = 0; i < size_; ++i)
        data_[i] = olddata[i];
    }
    data_[size_++] = x;
  }
  public synchronized void removeLast()
    throws NoSuchElementException {
    if (size_ == 0)
      throw new NoSuchElementException();
    else
      data_[--size_] = null;
  }
}
```

✎ What could happen if any of these methods were not synchronized?

Bundling Atomicity

- ❑ Consider adding synchronized methods that perform frequently desired sequences of actions as single atomic action, so that clients do not need to impose extra synchronization or control.

```
public interface Procedure { // apply an operation to an object
  public void apply(Object x);
}

public class ExpandableArrayV2 extends ExpandableArray {
  public ExpandableArrayV2(int cap) { super(cap); }
  public synchronized void applyToAll(Procedure p) {
    for (int i = 0; i < size_; ++i) {
      p.apply(data_[i]);
    } // oops -- SIZE _ and data_ should have been protected!
  }
}
```

✎ What possible liveness problems does this technique introduce?

Using inner classes

Anonymous inner classes (in Java 1.1) are the OO equivalent of lambda expressions:

```
class ExpandableArrayUser {
  public static void main(String[] args) {
    ExpandableArrayV2 a = new ExpandableArrayV2(100);
    for (int i = 0; i < 100; ++i)    // fill it up
      a.append(new Integer(i));
    a.applyToAll(new Procedure () {  // print all elements
        public void apply(Object x) {
          System.out.println(x);
        }
      }
    )
  }
}
```

Any variables shared with the host object must be declared `final` (immutable).

Partial Synchronization

Intent

Reduce overhead by synchronizing only within “critical sections”.

Applicability

- ❑ When objects have both mutable and immutable instance variables.
- ❑ When methods can be split into a “critical section” that deals with mutable state and a part that does not.

Design steps

- ❑ Fully synchronize all methods
- ❑ Remove synchronization for accessors to atomic or immutable values
- ❑ Remove synchronization for methods that access mutable state through a single other, already synchronized method
- ❑ Replace method synchronization by block synchronization for methods where access to mutable state is restricted to a single, critical section

Example: LinkedCells

```
public class LinkedCell {
  protected double value_; // NB: doubles are not atomic!
  protected final LinkedCell next_;  // fixed

  public LinkedCell (double val, LinkedCell next) {
    value_ = val; next_ = next;
  }

  public synchronized double value() { return value_; }
  public synchronized void setValue(double v) { value_ = v; }

  public LinkedCell next() {        // not synched!
    return next_;                   // next_ is immutable
  }
  ...
```

Example ...

```
  ...
  public double sum() {    // add up all element values
    double v = value();    // get via synchronized accessor
    if (next() != null)
      v += next().sum();
    return v;
  }

  public boolean includes(double x) { // search for x
    synchronized(this) {              // synch to access value
      if (value_ == x) return true;
    }
    if (next() == null) return false;
    else return next().includes(x);
  }
}
```

Containment

Intent

Achieve safety by avoiding shared variables. Unsynchronized objects are "contained" inside other objects that have at most one thread active at a time.

Applicability

- There is no need for shared access to the embedded objects.
- The embedded objects can be conceptualized as exclusively held resources
- You can tolerate the additional context dependence for embedded objects.
- Embedded objects must be structured as islands — communication-closed sets of objects ultimately reachable from a single unique reference. They cannot contain methods that reveal their identities to other objects.
- You are willing to hand-check designs for compliance.
- You can deal with or avoid indefinite postponements or deadlocks in cases where host objects must transiently acquire multiple resources.

Contained Objects — design steps

- ❑ Define the interface for the outer host object.
    - ☞ The host could be, e.g., an Adaptor, a Composite, or a Proxy, that provides synchronized access to an existing, unsynchronized class
- ❑ Ensure that the host is either fully synchronized, or is in turn a contained object.
- ❑ Define instances variables that are unique references to the contained objects.
    - ☞ Make sure that these references cannot leak outside the host!
    - ☞ Establish policies and implementations that ensure that acquired references are really unique!
    - ☞ Consider methods to duplicate or clone contained object, to ensure that copies are unique

Managed Ownership

- ❑ Model contained objects as physical resources:
  - ☞ If you have one, then you can do something that you couldn't do otherwise.
  - ☞ If you have one, then no one else has it.
  - ☞ If you give one to someone else, then you no longer have it.
  - ☞ If you destroy one, then no one will ever have it.
- ❑ If contained objects can be passed among hosts, define a transfer protocol.
  - ☞ Hosts should be able to acquire, give, take, exchange and forget resources
  - ☞ Consider using a dedicated class to manage transfer

A minimal transfer protocol class

This class is essentially a one-slot buffer for transferring resources between hosts in separate threads.

```
public class ResourceVariable {
  protected Object ref_;
  public ResourceVariable(Object res) { ref_ = res; }
  public synchronized Object resource() { return ref_; }
  public synchronized Object exchange(Object r) {
    Object old = ref_;
    ref_ = r;
    return old;
  }
}
```

NB: `exchange()` is enough to implement most transfer operations, e.g., `take()` is implemented by `exchange(null)`

Summary

You Should Know The Answers To These Questions:

- ❑ Why are immutable classes inherently safe?
- ❑ Why doesn’t a “relay” need to be synchronized?
- ❑ What is “balking”? When should a method balk?
- ❑ When is partial synchronization better than full synchronization?
- ❑ How does containment avoid the need for synchronization?

Can You Answer The Following Questions?

- ✎ When is it all right to declare only some methods as `synchronized`?
- ✎ When is an inner class better than an explicitly named class?
- ✎ What liveness problems can full synchronization introduce?
- ✎ Why is it a bad idea to have two separate critical sections in a single method?
- ✎ Does it matter if a contained object is synchronized or not?

3. Liveness and State

Overview

- ❑ Liveness and Fairness
  - ☞ The Dining Philosophers problem

- ❑ Guarded Methods
  - ☞ Checking guard conditions
  - ☞ Handling interrupts
  - ☞ Structuring notification
  - ☞ Tracking state
  - ☞ Delegating notifications

Liveness Problems

Liveness properties guarantee that your (concurrent) programs will make progress.

A program may be “safe”, yet suffer from various kinds of liveness problems:

- ❑ Contention:
  - ☞ AKA “starvation” or “indefinite postponement” — the system as a whole makes progress, but some individual processes don’t
- ❑ Dormancy:
  - ☞ A waiting process fails to be woken up
- ❑ Deadlock:
  - ☞ Two or more processes are blocked, waiting for resources held by the others (i.e., in a cycle)
- ❑ Premature termination:
  - ☞ A process is killed before it should be

Achieving Liveness

There are various strategies and techniques to ensure liveness:

- ❑ Start with safe design and selectively remove synchronization
- ❑ Start with live design and selectively add safety
- ❑ Adopt design patterns that limit the need for synchronization
- ❑ Adopt standard architectures that avoid cyclic dependencies

The Dining Philosophers Problem

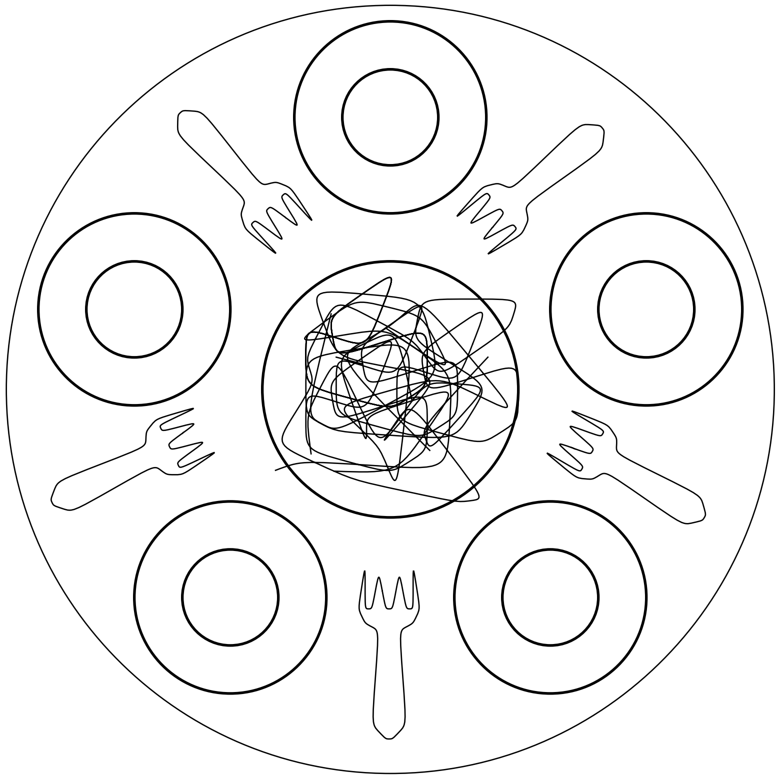
Philosophers alternate between thinking and eating.

A philosopher needs two forks to eat.

No two philosophers may hold the same fork simultaneously.

There should be no deadlock and no starvation.

Want efficient behaviour under absence of contention.



Dining Philosophers, Safety and Liveness

Dining Philosophers illustrates many classical safety and liveness issues:

Mutual Exclusion	Each fork can be used by one philosopher at a time
Condition synchronization	A philosopher needs two forks to eat
Shared variable communication	Philosophers share forks ...
Message-based communication	... or they can pass forks to each other
Busy-waiting	A philosopher can poll for forks ...
Blocked waiting	... or can sleep till woken by a neighbour
Livelock	All philosophers can grab the left fork and busy-wait for the right ...
Deadlock	... or grab the left one and wait (sleep) for the right
Starvation	A philosopher may starve if the left and right neighbours are always faster at grabbing the forks

Dining Philosopher Solutions

There are countless solutions to the Dining Philosophers problem that use various concurrent programming styles and patterns, and offer varying degrees of liveness guarantees:

- ❑ Number the forks;
  philosophers grab the lowest numbered fork first.
- ❑ Have philosophers leave the table while they think;
  allow at most four to sit at a time;
  philosophers queue to sit down.

✎ Is deadlock possible in either case?

✎ What about starvation?

✎ Are these solutions “fair”?

Fairness

There are subtle differences between definitions of fairness:

Weak fairness:

☞ If a process continuously makes a request, eventually it will be granted.

Strong fairness:

☞ If a process makes a request infinitely often, eventually it will be granted.

Linear waiting:

☞ If a process makes a request, it will be granted before any other process is granted the request more than once.

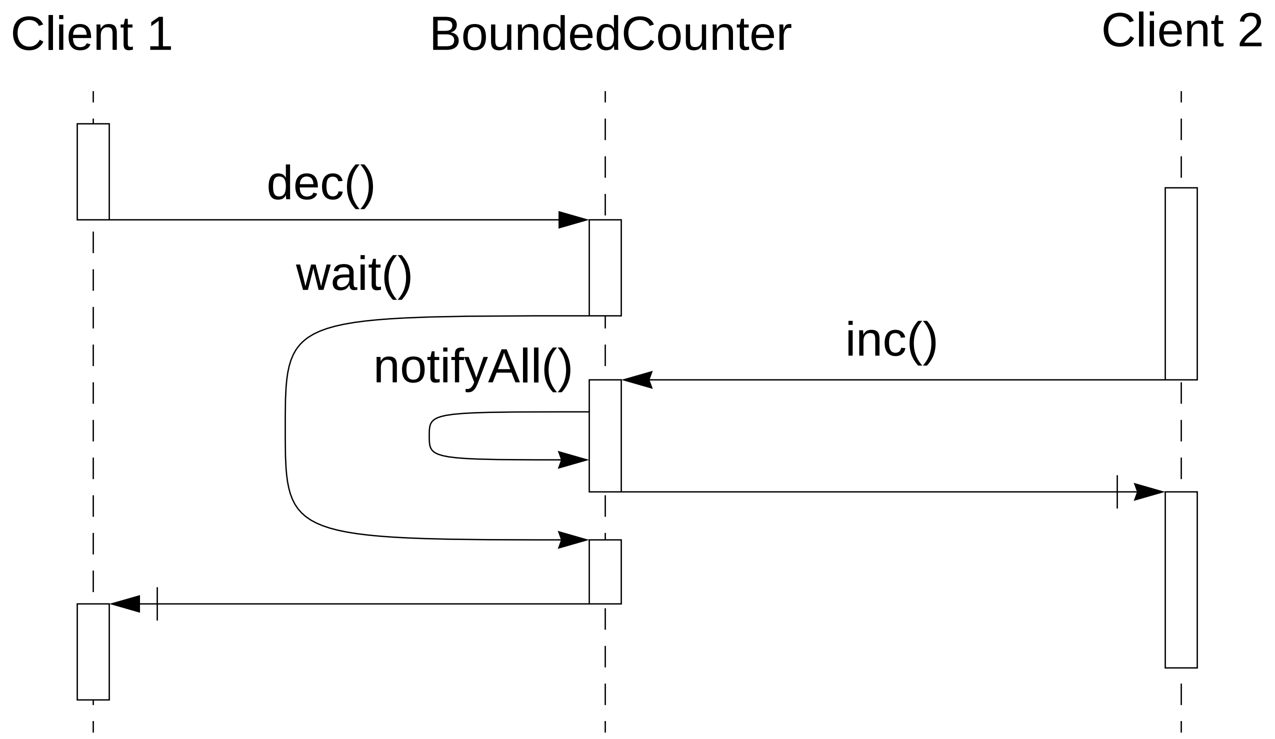
FIFO (first-in first out):

☞ If a process makes a request, it will be granted before that of any process making a later request.

Guarded Methods

Intent

Temporarily suspend an incoming thread when an object is not in the right state to fulfil a request, and wait for the state to change rather than balking (raising an exception).



Guarded Methods — applicability

❑ Clients can tolerate indefinite postponement. (Otherwise, use a balking design.)

❑ You can guarantee that the required states are eventually reached (via other requests), or if not, that it is acceptable to block forever.

❑ You can arrange that notifications occur after all relevant state changes. (Otherwise consider a design based on a busy-wait spin loop.)

❑ You can avoid or cope with liveness problems due to waiting threads retaining all synchronization locks (except for that of the host).

❑ You can construct computable predicates describing the state in which actions will succeed. (Otherwise consider an optimistic design.)

❑ Conditions and actions are managed within a single object. (Otherwise consider a transactional form.)

Guarded Methods — design steps

The basic recipe is to use `wait` in a conditional loop to block until it is safe to proceed, and use `notifyAll` to wake up blocked threads.

```
public synchronized Object service() {
    while (wrong State) {
        try { wait(); }
        catch (InterruptedException e) { }
    }

    // fill request and change state ...

    notifyAll();    // NB: use notify() only if it does not
                    // matter which waiting thread you wake up
    return result;
}
```

Separate interface from policy

- ❑ Define interfaces for the methods, so that classes can implement guarded methods according to different policies.

```
public interface BoundedCounter {
  public static final long MIN = 0;  // minimum allowed value
  public static final long MAX = 10; // maximum allowed value

  public long value();     // invariant: MIN <= value() <= MAX
                           // initial condition: value() == MIN

  public void inc();       // increment only when value() < MAX
  public void dec();       // decrement only when value() > MIN
}
```

Check guard conditions

- ❑ Define a predicate that precisely describes the conditions under which actions may proceed. (This can be encapsulated as a helper method.)

- ❑ Precede the conditional actions with a guarded wait loop of the form:

  ```
  while (!condition)
      try { wait(); }
      catch (InterruptedException ex) { ... }
  ```

  Optionally, encapsulate this code as a helper method.

- ❑ If there is only one possible condition to check in this class (and all plausible subclasses), and notifications are issued only when the condition is true, then there is no need to re-check the condition after returning from `wait()`

- ❑ Ensure that the object is in a consistent state (i.e., the class invariant holds) before entering any `wait` (since wait releases the synchronization lock).

  The easiest way to do this is to perform the guards before taking any actions.

Handle interrupts

- ❑ Establish a policy for dealing with `InterruptedExceptions` (which will also force a return from `wait`). Possible policies are:
  - ☞ Ignore interrupts (i.e., have an empty `catch` clause), which preserves safety at the possible expense of liveness.
  - ☞ Terminate the current thread (via `stop`). This also preserves safety, though brutally! (Not recommended.)
  - ☞ Exit the method, possibly raising an exception. This preserves liveness but may require the caller to take special action to preserve safety.
  - ☞ Take some pre-planned action; such as cleanup and restart.
  - ☞ Ask for user intervention before taking further action.

Interrupts can be useful to signal that the guard can never become true because, for example, the collaborating threads have terminated.

Signal state changes

- ❑ Add notification code to each method of the class that changes state in any way that can affect the value of a guard condition. Some options are:
  - ☞ `notifyAll` wakes up all threads that are blocked in waits for the host object. Calls to `notifyAll` (as well as `notify`) must be enclosed within a synchronized method or block.
  - ☞ `notify` wakes up only one thread (if any exist). This is best treated as an optimization where:
    - ➪ all blocked threads are necessarily waiting for conditions signalled by the same notifications,
    - ➪ only one of them can be enabled by any given notification, and
    - ➪ it does not matter which one of them becomes enabled.
  - ☞ You build your own special-purpose notification methods using `notify` and `notifyAll`. (For example, to selectively notify threads, or to provide certain fairness guarantees.)

Structure notifications

❑ Ensure that each wait is balanced by at least one notification. Options include:

Blanket Notifications	Place a *notification at the end of every method* that can cause any state change (i.e., assigns any instance variable). Simple and reliable, but can cause performance problems ...
Encapsulating Assignment	*Encapsulate assignment* to each variable mentioned in any guard condition *in a helper method* that performs the notification after updating the variable.
Tracking State	Only issue notifications for the *particular state changes* that could actually unblock waiting threads. May improve performance, at the cost of flexibility (i.e., subclassing becomes harder.)
Tracking State Variables	Maintain an *instance variable that represents control state*. Whenever the object changes state, invoke a helper method that re-evaluates the control state and will *issue notifications if guard conditions are affected*.
Delegating Notifications	Use *helper objects to maintain aspects of state* and have these helpers issue the notifications.

Encapsulating assignment

```
public class BoundedCounterV0 implements BoundedCounter {
  protected long count_ = MIN;

  public synchronized long value() { return count_; }

  public synchronized void inc() {
    awaitIncrementable();
    setCount(count_ + 1);
  }

  public synchronized void dec() {
    awaitDecrementable();
    setCount(count_ - 1);
  }
  ...
```

Encapsulating assignment ...

```
  ...
  protected synchronized void setCount(long newValue) {
    count_ = newValue;
    notifyAll(); // wake up any thread depending on new value
  }

  protected synchronized void awaitIncrementable() {
    while (count_ >= MAX)
    try { wait(); }
    catch(InterruptedException ex) {};
  }

  protected synchronized void awaitDecrementable() {
    while (count_ <= MIN)
      try { wait(); }
      catch(InterruptedException ex) { };
  }
}
```

Tracking State

The only transitions that could possibly affect waiting threads in BoundedCounter are those that step away from logical states bottom and top:

```
public class BoundedCounterVST implements BoundedCounter {
  protected long count_ = MIN;
  public synchronized long value() {
    return count_;
  }
  public synchronized void inc() {
    while (count_ == MAX)
      try { wait(); } catch(InterruptedException ex) {};
    if (count_++ == MIN)
      notifyAll(); // signal if previously in bottom state
  }
  public synchronized void dec() { ... } // ditto
}
```

Tracking State Variables

```
public class BoundedCounterVSW implements BoundedCounter {
  static final int BOTTOM= 0;      // logical states
  static final int MIDDLE= 1;
  static final int TOP= 2;

  protected int state_ = BOTTOM;   // the state variable
  protected long count_ = MIN;

  protected synchronized void checkState() {
    int oldState = state_;
    if (count_ == MIN)          state_ = BOTTOM;
    else if (count_ == MAX)     state_ = TOP;
    else                        state_ = MIDDLE;
    if (state_ != oldState          // notify on transition
        && (oldState == TOP || oldState == BOTTOM))
      notifyAll();
  }
  ...
```

Tracking State Variables ...

```
  ...
  public synchronized long value() { return count_; }

  public synchronized void inc() {
    while (state_ == TOP)    // only consult logical state
      try { wait(); }
      catch(InterruptedException ex) {};
    ++count_;                // modify actual state
    checkState();            // re-evaluate logical state
  }

  public synchronized void dec() { ... }
}
```

Delegating notifications

NotifyLong() encapsulates both atomic state changes and notifications:

```
public class NotifyingLong {
  private long value_;
  private Object observer_;
  public NotifyingLong(Object o, long v) {
    observer_ = o;
    value_ = v;
  }
  public synchronized long value() { return value_; }
  public void setValue(long v) {
    synchronized(this) {      // NB: partial synchronization
      value_ = v;
    }
    synchronized(observer_) {
      observer_.notifyAll();  // NB: must be synchronized!
    }
  }
}
```

Delegating notifications ...

Notification is delegated to the helper object:

```
public class BoundedCounterVNL implements BoundedCounter {
  private NotifyingLong c_ = new NotifyingLong(this, MIN);
  public synchronized long value() {
    return c_.value();
  }
  public synchronized void inc() {
    while (c_.value() >= MAX)
      try { wait(); }
      catch(InterruptedException ex) {};
    c_.setValue(c_.value()+1);    // will issue notification
  }
  public synchronized void dec() {... }
}
```

Summary

You Should Know The Answers To These Questions:

- ❑ What kinds of liveness problems can occur in concurrent programs?
- ❑ What is the difference between livelock and deadlock?
- ❑ When should methods recheck guard conditions after waking from a `wait()`?
- ❑ Why should you usually prefer `notifyAll()` to `notify()`?
- ❑ When and where should you issue notification?

Can You Answer The Following Questions?

- ✎ How can you detect deadlock? How can you avoid it?
- ✎ What is the easiest way to guarantee fairness?
- ✎ When are guarded methods better than balking?
- ✎ What is the best way to structure guarded methods for a class if you would like it to be easy for others to define correctly functioning subclasses?
- ✎ Is the complexity of delegating notifications worth it?

4. Lab session

5. Liveness and Asynchrony

Overview

- ❑ Asynchronous invocations
  - ☞ Simple Relays
    - ➪ Direct Invocations
    - ➪ Thread-based messages; Gateways
    - ➪ Command-based messages
  - ☞ Tail calls
  - ☞ Early replies
  - ☞ Futures

Asynchronous Invocations

Intent

Avoid waiting for a request to be serviced by decoupling sending from receiving.

Applicability

- ❑ When a host object can distribute services amongst multiple helper objects.
- ❑ When an object does not need the result of an invocation to continue doing useful work.
- ❑ When invocations that are logically asynchronous, regardless of whether they are coded using threads.
- ❑ During refactoring, when classes and methods are split in order to increase concurrency and reduce liveness problems.

Asynchronous Invocations — form

Generally, asynchronous invocation designs take the following form:

```
class Host {
  public service() {
    pre();              // code to run before invocation
    invokeHelper();     // the invocation
    during();           // code to run in parallel
    post();             // code to run after completion
  }
}
```

Asynchronous Invocations — design steps

Consider the following issues:

Does the Host need to get results back from the Helper?	Not if, e.g., the Helper returns results directly to the Host's caller!
Can the Host process new requests while the Helper is running?	Might depend on the kind of request ...
Does the Host need to do something while the Helper is running?	i.e., in the `during()` code
Does the Host need to do synchronized pre-invocation processing?	i.e., if `service()` is guarded or if `pre()` updates the Host's state
Does the Host need to do synchronized post-invocation processing?	i.e., if `post()` updates the Host's state
Does post-invocation processing only depend on the Helper's result?	... or does the host have to wait for other conditions?
Is the same Helper always used?	Is a new one generated to help with each new service request?

Simple Relays

A relay method is obtains all its functionality by delegating to the helper, without any `pre()`, `during()`, or `post()` actions.

Three common forms:

- ❑ Direct invocations
  - ☞ Invoke the Helper directly, but without synchronization
- ❑ Thread-based messages
  - ☞ Create a new thread to invoke the Helper
- ❑ Command-based messages
  - ☞ Pass the request as a Command object to another object that will run it

Relays are commonly seen in Adaptors.

Direct invocations

Asynchrony is achieved through the absence of synchronization.

The Host is free to accept other requests, while the Host’s caller must wait for the reply.

```
class Host {
  protected Helper helper_ = new Helper();
  public void service() {           // unsynchronized
    invokeHelper();                 // stateless method!
  }
  protected void invokeHelper() {  // unsynchronized
    helper_.help();
  }
}
```

If `helper_` is mutable, it can be protected with an accessor:

```
  protected synchronized Helper helper() { return helper_; }
  public void service() {           // unsynchronized
    helper().help();                // partially synchronized
  }
```

Thread-based messages

The invocation can be performed within a new thread:

```
protected void invokeHelper() {
  new Thread() {                              // An inner class
    final Helper h_ = helper_;                // Must be final!
    public void run() { h_.help() ; }
  }.start();
}
```

The cost of evaluating Helper.help() should outweigh the overhead of creating a thread!

- ☞ If the Helper is a daemon (loops endlessly)
- ☞ If the Helper does I/O
- ☞ Possibly, if multiple helper methods are invoked

Thread-per-message Gateways

Variant: the host may construct a new Helper to service each request.

```
public class FileIO {
  public void writeBytes(String fileName, byte[] data) {
    new Thread (new FileWriter(fileName, data)).start();
  }
  public void readBytes(...) { ... }
}

class FileWriter implements Runnable {
  private String nm_;                       // hold arguments
  private byte[] d_;
  public FileWriter(String name, byte[] data) { ... }
  public void run() { ... } // write bytes in d_ to file nm_ ...
}
```

Command-based messages

The Host can also put a message in a queue for another object that will invoke the Helper:

```
protected EventQueue q_;
protected invokeHelper() {
  q_.put(new HelperMessage(helper_));
}
```

Command-based forms especially useful for:

- ❑ scheduling of helpers
- ❑ undo and replay capabilities
- ❑ transporting messages over networks

Tail calls

Applies when the helper method is the last statement of a method (i.e., there is no `post()` processing). Only `pre()` code is synchronized.

The host is immediately available to accept other messages invoking the helper.

```
class Subject {
  protected Observer obs_ = new ...;
  protected double state_;
  public void updateState(double d) {  // not synched
    doUpdate(d);                       // synched
    sendNotification();                // not synched
  }
  protected synchronized doUpdate(double d) { state_ = d; }
  protected void sendNotification() {
    obs_.changeNotification(this);
  }
}
```

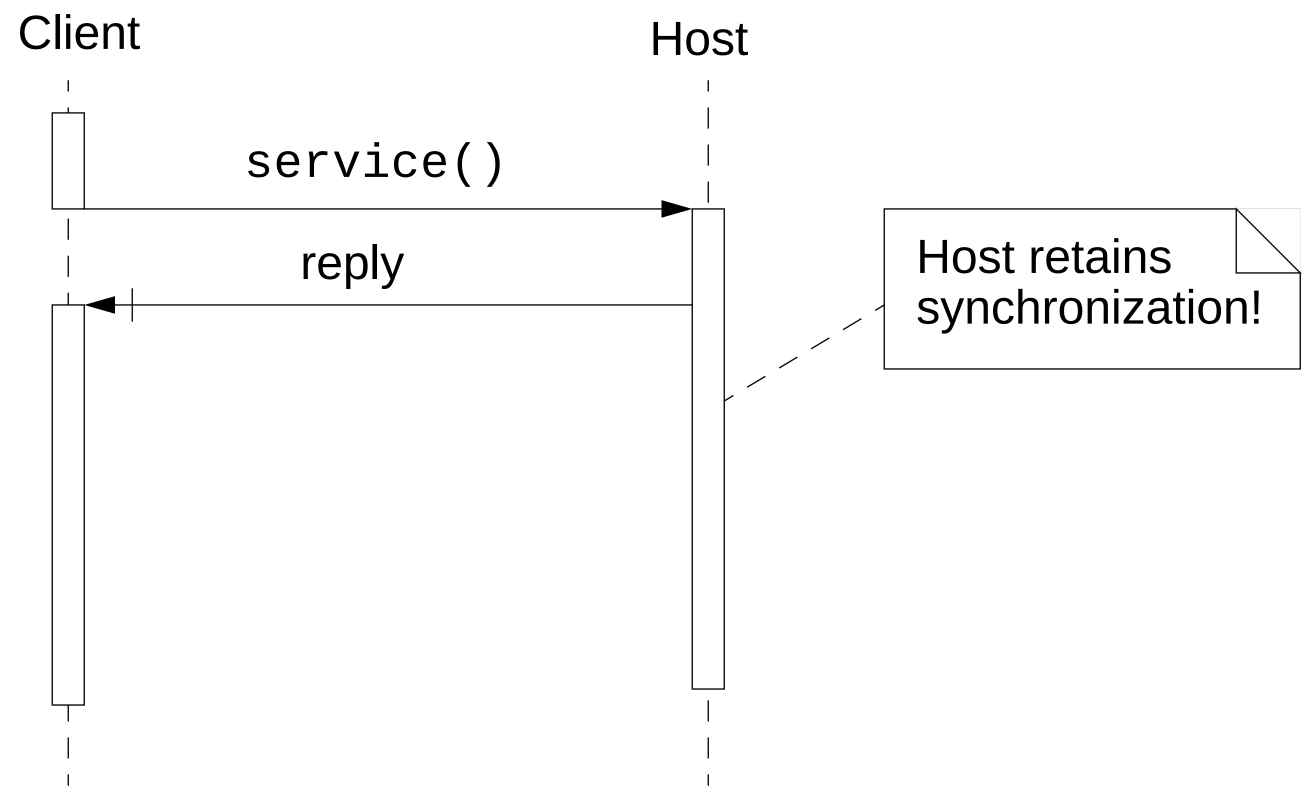
Tail calls with new threads

Alternatively, the tail call may be performed in a separate thread:

```
public synchronized void updateState(double d) {
  state_ = d;
  new Thread(){
    final Observer o_ = obs_;
    public void run() {
      o_.changeNotification(Subject.this);
    }
  }.start();
}
```

Early Reply

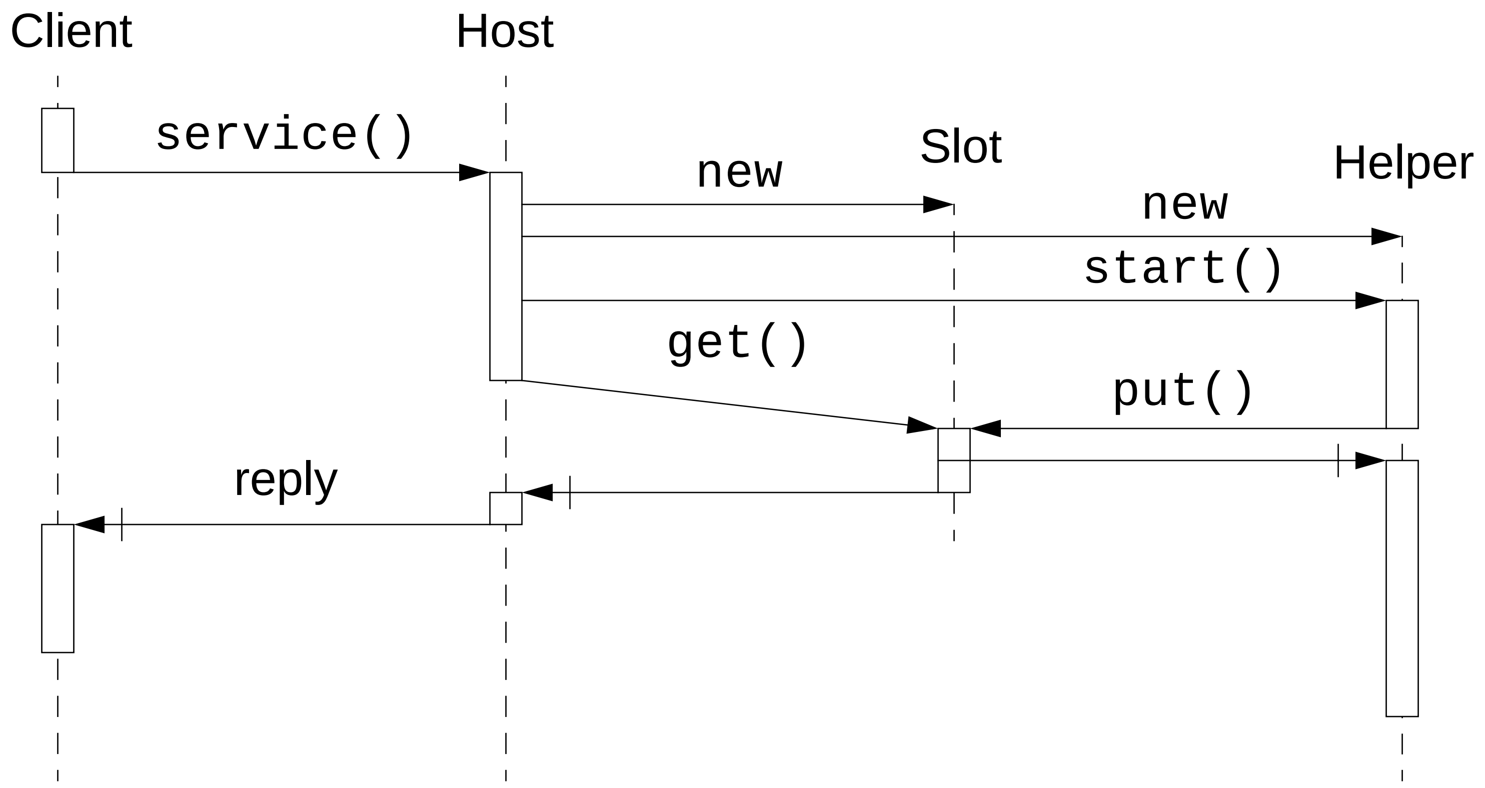
Early reply allows a host to perform useful activities after returning a result to the client:



Early reply is a built-in feature in some programming languages.
It can be easily simulated when it is not a built-in feature.

Simulating Early Reply

A one-slot buffer can be used to pick up the reply from a helper thread:



A one-slot buffer is a simple abstraction that can be used to implement many higher-level concurrency abstractions ...

One-Slot Buffer

```
class Slot {                                        // a one-slot buffer
   private Object slotVal_;                         // initially null
   public synchronized void put(Object val) {       // put contents,
      while (slotVal_ != null) {                    // if there is room
         try { wait(); }                            // otherwise wait
         catch (InterruptedException e) { }
      }
      slotVal_ = val;
      notifyAll();                                  // wake up waiting consumer
      return;
   }
   public synchronized Object get() {               // return contents,
      Object rval;
      while (slotVal_ == null) {                    // if available
         try { wait(); }                            // otherwise wait
         catch (InterruptedException e) { }
      }
      rval = slotVal_;
      slotVal_ = null;
      notifyAll();                                  // wake up waiting producer
      return rval;
   }
}
```

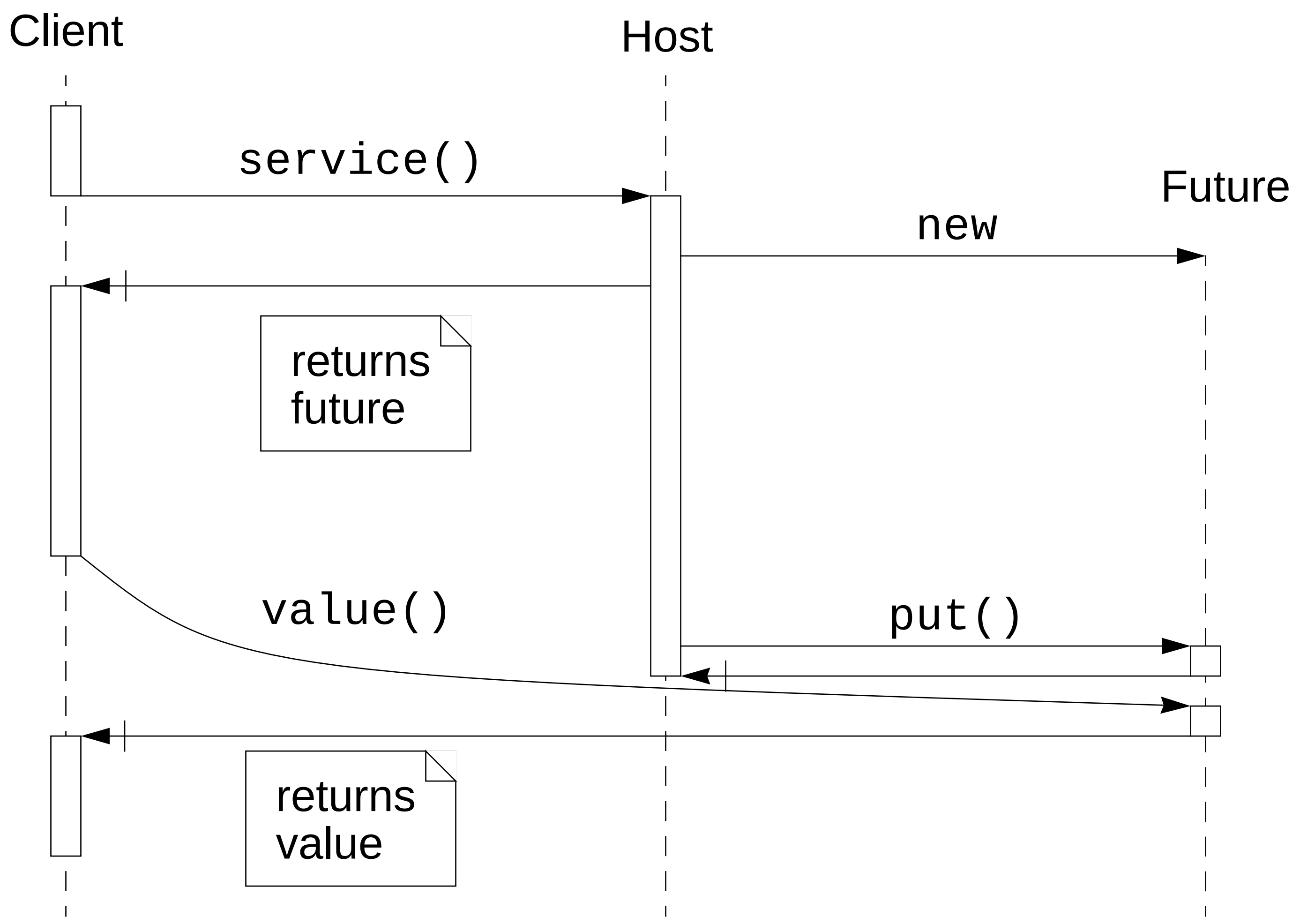
Early Reply in Java

Early reply can be easily implemented using an anonymous inner class:

```
public Stuff service() {           // unsynchronized
  final Slot reply = new Slot();
  new Thread() {
    public void run() {
      Stuff result;
      synchronized (this) {        // retain lock!
        // compute result
        reply.put(result);         // send early reply
        // do cleanup activity
      }
    }
  }.start();
  return (Stuff) reply.get();      // early reply
}
```

Futures

Futures allow a client to continue in parallel with a host until the future value is needed:



A Future Class

Futures can be implemented as a layer of abstraction around a shared Slot:

```
class Future {
  private Object val_;        // initially null
  private Slot slot_;         // shared with some worker
  public Future(Slot slot) {
    slot_ = slot;
  }
  public Object value() {
    if (val_ == null)
      val_ = slot_.get();     // be sure to only get() once!
    return val_;
  }
}
```

Using Futures in Java

WIth futures, the client, rather than the host, proceeds in parallel with a helper thread.

```
public Future service () {   // unsynchronized
  final Slot slot = new Slot();
  new Thread() {
    public void run() {
      slot.put(computeResult());
    }
  }.start();
  return new Future(slot);  // immediately return Future
}

protected synchronized Object computeResult() { ... }
```

Without special language support, futures are less transparent than early replies, since the client must explicitly request a `value()` from the future object.

Summary

You Should Know The Answers To These Questions:

- ❑ What general form does an asynchronous invocation take?
- ❑ When should you consider using asynchronous invocations?
- ❑ In what sense can a direct invocation be “asynchronous”?
- ❑ Why (and how) would you use “inner classes” to implement asynchrony?
- ❑ What is “early reply”, and when would you use it?
- ❑ What are “futures”, and when would you use them?
- ❑ How can implement futures and early replies in Java?

Can You Answer The Following Questions?

- ✎ Why are servers commonly structured as thread-per-message gateways?
- ✎ Which of the concurrency abstractions we have discussed till now can be implemented using one-slot-buffers as the only synchronized objects?
- ✎ When are futures better than early replies? Vice versa?

6. Fine-grained Synchronization

Overview

- ❑ Condition Objects
  - ☞ The “Nested Monitor Problem”
  - ☞ Permits and Semaphores
- ❑ Concurrently available methods
  - ☞ Priority
  - ☞ Interception
  - ☞ Readers and Writers
- ❑ Optimistic methods

Condition Objects

Intent

Condition objects encapsulate the waits and notifications used in guarded methods.

Applicability

- ❑ To simplify class design by off-loading waiting and notification mechanics.
  - ☞ Because of the limitations surrounding the use of condition objects in Java, in some cases the use of condition objects will increase rather than decrease design complexity!
- ❑ As an efficiency manoeuvre.
  - ☞ By isolating conditions, you can often avoid notifying waiting threads that could not possibly proceed given a particular state change.
- ❑ As a means of encapsulating special scheduling policies surrounding notifications, for example to impose fairness or prioritization policies.
- ❑ In the particular cases where conditions take the form of “permits” or “latches.”

A Simple Condition Object

Condition objects implement this interface:

```
public interface Condition {
  public void await();  // wait for some condition
  public void signal(); // signal that some condition holds
}
```

Suppose we tried to encapsulate guard conditions with this class:

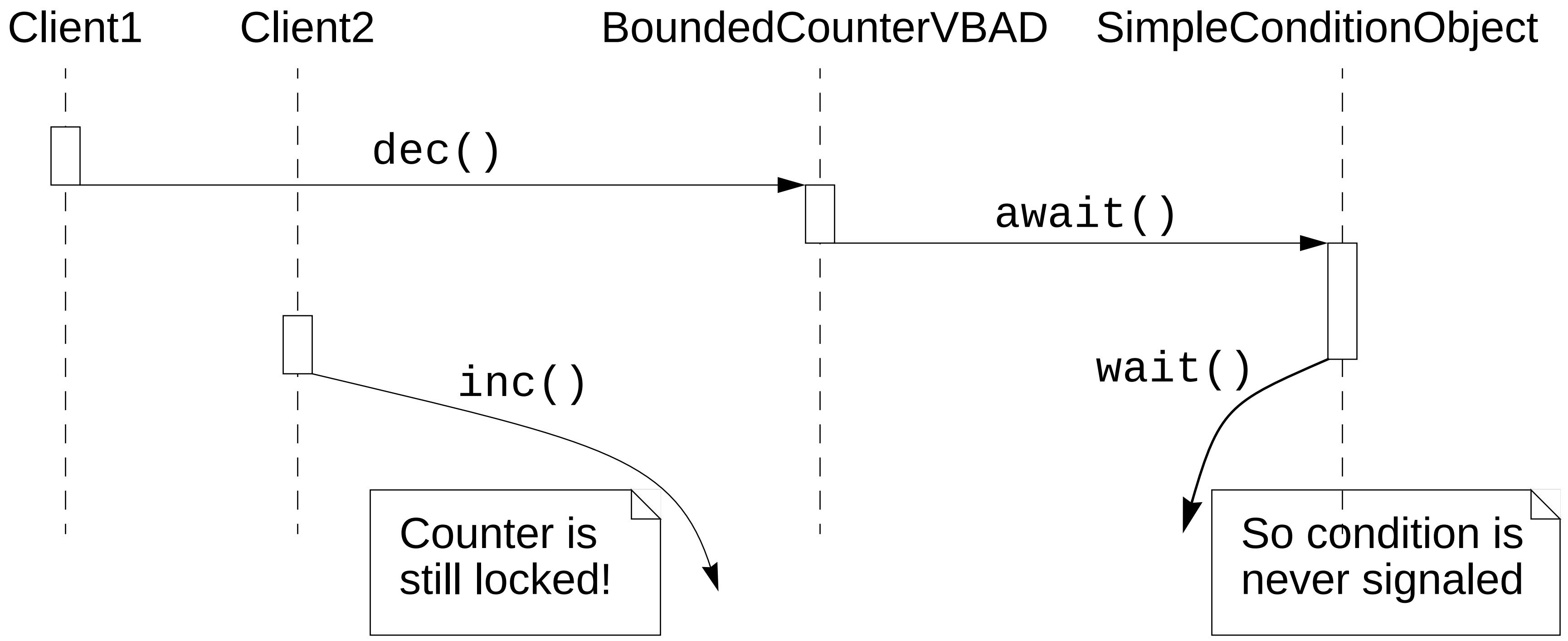
```
public class SimpleConditionObject implements Condition {
  public synchronized void await() {
    try { wait(); }
    catch (InterruptedException ex) {}
  }
  public synchronized void signal() {
    notifyAll();
  }
}
```

Careless use of this class can lead to the “Nested Monitor Problem”

The Nested Monitor problem

```
public class BoundedCounterVBAD implements BoundedCounter {
  protected long count_ = MIN;
  protected Condition notMin_ = new SimpleConditionObject();
  protected Condition notMax_ = new SimpleConditionObject();
  public synchronized long value() { return count_; }
  public synchronized void dec() {
    while (count_ == MIN)
      notMin_.await();      // wait till count not MIN
    if (count_-- == MAX)
      notMax_.signal();
  }
  public synchronized void inc() {   // can't get in!
    while (count_ == MAX)
      notMax_.await();
    if (count_++ == MIN)
      notMin_.signal();     // we never get here!
  }
}
```

The Nested Monitor problem ...



Nested monitor lockouts occur whenever a blocked thread holds the lock for an object containing the method that would otherwise provide a notification to unblock the wait.

Solving the Nested Monitors problem

You must ensure that:

- ❑ Waits do not occur while synchronization is held on the host object.
  - ☞ This leads to a guard loop that reverses the synchronization seen in the faulty version.
- ❑ Notifications are never missed.
  - ☞ The entire guard wait loop should be enclosed within synchronized blocks on the condition object.
- ❑ Notifications do not deadlock.
  - ☞ All notifications should be performed only upon release of all synchronization except of that for the notified condition object.
- ❑ Helper and host state must be consistent.
  - ☞ If the helper object maintains any state, it must always be consistent with that of the host, and if it shares any state with the host, that access is properly synchronized.

Example solution

```
public class BoundedCounterVCV implements BoundedCounter {
   ...
   public void inc() {         // NOT synched!
      boolean wasMin = false; // record notification condition
      synchronized(notMax_) {    // synch on condition object
         for (;;) {                 // the recast guard loop
            synchronized(this) {
               if (count_ < MAX) {   // check and act
                  wasMin = (count_++ == MIN);
                  break;
               }
            }
            notMax_.await();    // release host synch before wait
         }
      }
      if (wasMin) notMin_.signal();  // first release sync!
   }
}
```

Permits and Semaphores

Intent

Bundle synchronization in a condition object when synchronization is mainly concerned with tracking the value of a counter.

Applicability

- ❑ When any given `await` may proceed only if there have been more signals than awaits.
  - ☞ More generally, if there are enough “permits”, where every signal increments and every await decrements the number of permits.
- ❑ You need to guarantee the absence of missed signals.
  - ☞ Unlike simple condition objects, semaphores work even if one thread enters its await after another thread has signalled that it may proceed.
- ❑ The host classes using them can arrange to invoke `Condition` methods outside of synchronized methods or code blocks.

Permits and Semaphores — design steps

- Define a class implementing `Condition` that maintains a permit count,and immediately releases await if there are already enough permits.
  - ☞ e.g., `BoundedCounter`

- As with all kinds of condition objects, the classes using them must avoid invoking await inside of synchronized methods and code blocks.
  - ☞ One way to help ensure this is to use a before/after design of the form:

```
class Host {
  Condition aCondition_;
  Condition anotherCondition_; ...
  public method m1() {
    aCondition_.await();  // not synched
    doM1();               // synched
    for each Condition c enabled by m1()
      c.signal();         // not synched
  }
  protected synchronized doM1() { ... }
}
```

Variants

- Permit Counters (Counting Semaphores)
  - Just keep track of the number of “permits”
  - Can use `notify` instead of `notifyAll` if class is `final`
- Fair Semaphores
  - Maintain FIFO queue of threads waiting on a `SimpleCondition`
- Locks and Latches
  - Locks can be acquired and released in separate methods
  - Keep track of thread holding the lock so locks can be reentrant!
  - A latch is set to true by `signal`, and always stays true

See the On-line supplement for details.

Concurrently Available Methods

Intent

Non-interfering methods comprising a service an be made concurrently available by splitting them into different objects or aspects of the same object, while tracking state and execution conditions to enable and disable the methods according to a given concurrency control policy.

Applicability

- ❑ Host objects are typically accessed across many different threads.
- ❑ Host services are not completely interdependent, so need not be performed under mutual exclusion.
- ❑ You need better throughput for one or more of the services provided by the object, and need to eliminate nonessential blocking on synchronization locks.
- ❑ You want to prevent various accidental or malicious denial of service attacks in which synchronized methods on a host block because some client forever holds its lock.
- ❑ Use of full synchronization would needlessly make host objects prone to deadlock or other liveness problems.

Concurrent Methods — design steps

Layer concurrency control policy over mechanism by:

- ❑ Policy Definition:
  - ☞ When may methods run concurrently?
  - ☞ What happens when a disabled method is invoked?
  - ☞ What priority is assigned to waiting tasks?
- ❑ Instrumentation:
  - ☞ Define state variables that can detect and enforce policy.
- ❑ Interception:
  - ☞ Have the host object intercept public messages and then relay them under the appropriate conditions to the methods that actually perform the actions.

Priority

- Priority may depend on any of:
  - Intrinsic attributes of the tasks (class and instance variable values).
  - Representations of task priority, cost, price, or urgency.
  - The number of tasks waiting for some condition.
  - The time at which each task is added to a queue.
  - Fairness — guarantees that each waiting task will eventually run.
  - The expected duration or time to completion of each task.
  - The desired completion time of each task.
  - Termination dependencies among tasks.
  - The number of tasks that have completed.
  - The current time.

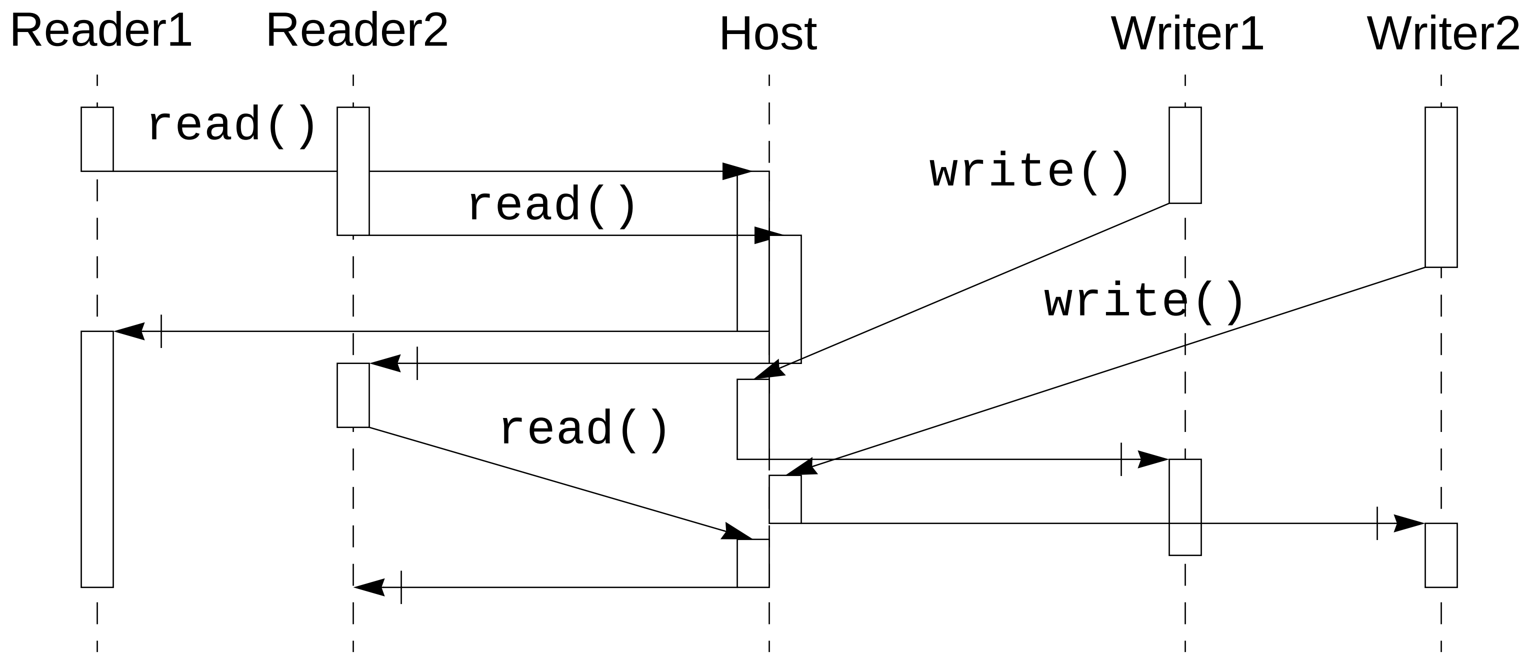
Interception

Interception strategies include:

- ❑ Pass-Throughs
  - ☞ The host maintains a set of immutable references to helper objects and simply relays all messages to them within unsynchronized methods.
- ❑ Lock-Splitting
  - ☞ Instead of splitting the class, split the synchronization locks associated with subsets of functionality
- ❑ Before/After methods
  - ☞ Public methods contain before/after processing surrounding calls to non-public methods in the host that perform the services.

Concurrent Reader and Writers

“Readers and Writers” are a family of concurrency control designs that provide various policies governing concurrent invocation of non-mutating accessors (“Readers”) and mutative, state-changing operations (“Writers”).



The basic idea is to let any number of readers to concurrently execute as long as there are no writers, but writers have exclusive access.

Readers and Writers Policies

Individual policies must address:

- ❑ Can new Readers join already active Readers even if a Writer is waiting?
  - ☞ If yes, Writers may starve; if not, the throughput of Readers decreases.
- ❑ If both Readers and Writers are waiting for a Writer to finish, which should you let in first?
  - ☞ Readers? A Writer? Earliest first? Random? Alternate?
  - ☞ Similar choices are available after termination of Readers.
- ❑ Can Readers upgrade to Writers without having to give up access?

A typical set of choices:

- ❑ Block incoming Readers if there are waiting Writers.
- ❑ “Randomly” choose among incoming threads (i.e., let the scheduler choose).
- ❑ No upgrade mechanisms.

Before/after methods are the simplest way to implement Readers and Writers policies.

Readers and Writers example

```
public abstract class RWVT {
  protected int activeReaders_ = 0;     // zero or more
  protected int activeWriters_ = 0;     // always zero or one
  protected int waitingReaders_ = 0;
  protected int waitingWriters_ = 0;
  protected abstract void read_();      // defined by subclass
  protected abstract void write_();
  public void read() {                  // unsynchronized
    beforeRead();                       // obtain access
    read_();
    afterRead();                        // release access
  }
  public void write() {
    beforeWrite();
    write_();
    afterWrite();
  }
...
```

Readers and Writers ...

```
...
  protected synchronized void beforeRead() {
    ++waitingReaders_;                    // available to subclasses
    while (!allowReader())
      try { wait(); }
      catch (InterruptedException ex) {}
    --waitingReaders_;
    ++activeReaders_;
  }
  protected boolean allowReader() {         // default policy
    return waitingWriters_ == 0 && activeWriters_ == 0;
  }
  protected synchronized void afterRead() {
    --activeReaders_;
    notifyAll();
  }
...
```

Readers and Writers ...

```
...
  protected synchronized void beforeWrite() {
    ++waitingWriters_;
    while (!allowWriter())
      try { wait(); }
      catch (InterruptedException ex) {}
    --waitingWriters_;
    ++activeWriters_;
  }
  protected boolean allowWriter() {       // default policy
    return activeReaders_ == 0 && activeWriters_ == 0;
  }
  protected synchronized void afterWrite() {
    --activeWriters_;
    notifyAll();
  }
}
```

Optimistic Methods

Intent

Optimistic methods attempt actions, but rollback state if the actions could have been interfered with by the actions of other threads. After rollback, they either throw failure exceptions or retry the actions.

Applicability

- ❑ Clients can tolerate either failure or retries.
  - ☞ If not, consider using guarded methods .
- ❑ You can avoid or cope with livelock.
- ❑ You have a way to deal with actions occurring before failure detection
  - ☞ Provisional action: “pretend” to act, delaying commitment of effects until the possibility of failure has been ruled out.
  - ☞ Rollback/Recovery: undo the effects of each performed action. If messages are sent to other objects, they must be undone with “anti-messages”

Optimistic Methods — design steps

❑ Collect and encapsulate all mutable state so that it can be tracked as a unit.

- ☞ Define an immutable helper class holding values of all instance variables.
- ☞ Define a representation class, but make it mutable (allow instance variables to change), and additionally include a version number (or transaction identifier) field or even a sufficiently precise time stamp.
- ☞ Embed all instance variables, plus a version number, in the host class, but define `commit` to take as arguments all assumed values and all new values of these variables.
- ☞ Maintain a serialized copy of object state.
- ☞ Various mixtures of the above ...

Detect failure ...

- Provide an operation that simultaneously detects version conflicts and performs updates via a method of the form:

```
class Optimistic {                         // generic code sketch

  private State currentState_;             // State is any type

  synchronized boolean commit(State assumed, State next) {
    boolean success = (currentState_ == assumed);
    if (success)
      currentState_ = next;
    return success;
  }
}
```

Detect failure ...

❑ Structure the main actions of each public method as follows:

```
State assumed = currentState();
State next = ...
if (!commit(assumed, next))
  rollback();
else
  otherActionsDependingOnNewStateButNotChangingIt();
```

Handle conflicts ...

- Choose and implement a policy for dealing with commitment failure:
  - ☞ Throw an exception upon commit failure that tells a client that it may retry. (Of course, this kicks the decision back to the caller, which may or may not be in a better position to decide whether to retry.)
  - ☞ Internally retry the action until it succeeds.
  - ☞ Retry some bounded number of times, or until a timeout occurs, finally throwing an exception.
  - ☞ Synchronize the method, precluding commit failure. This can be done even when other methods in the class use exceptions or retries.

- Take precautions to ensure that retries are based upon accurate, current values of instance variables.
  - ☞ If state is maintained in an immutable helper object accessed via a single reference in the class, then this reference should be declared `volatile`. All accessor methods can be left as unsynchronized.

`volatile` specifies that a variable changes asynchronously and the compiler should not attempt optimizations with it (such as using a copy stored in a register).

Ensure progress ...

- ❑ Take precautions to ensure progress in case of internal retries within state-dependent methods.
  - ☞ Optimistic state-dependent methods require use of a busy-wait spin loop in which it is counterproductive to immediately retry the method.
  - ☞ Yielding may not be effective unless all threads have reasonable priorities and the Java scheduler at least approximates fair choice among waiting tasks (which it is not guaranteed to do)!

- ❑ Limit retries.
  - ☞ Unless there is some independent assurance that the method will eventually succeed, retries can result in livelock.

An Optimistic Bounded Counter

```
public class BoundedCounterVOPT implements BoundedCounter {
   protected volatile Long count_ = new Long(MIN);
   protected synchronized boolean commit(Long oldc, Long newc) {
      boolean success = (count_ == oldc);
      if (success) count_ = newc;
      return success;
   }
   public long value() { return count_.longValue(); }
   public void inc() {
      for (;;) {                                  // thinly disguised busy-wait!
         Long c = count_; long v = c.longValue();
         if (v < MAX && commit(c, new Long(v+1))) break;
         Thread.currentThread().yield();          // is there another thread?!
      }
   }
   public void dec() {
      for (;;) {
         Long c = count_; long v = c.longValue();
         if (v > MIN && commit(c, new Long(v-1))) break;
         Thread.currentThread().yield();
      }
   }
}
```

Summary

You Should Know The Answers To These Questions:

- ❑ What are “condition objects”? How can they make your life easier? Harder?
- ❑ What is the “nested monitor problem”? How can you avoid it?
- ❑ What are “permits” and “semaphores”? When is it natural to use them?
- ❑ Why (when) can semaphores use `notify()` instead of `notifyAll()`?
- ❑ When should you consider allowing methods to be concurrently available?
- ❑ What kinds of policies can apply to concurrent Readers and Writers?
- ❑ How do optimistic methods differ from guarded methods?

Can You Answer The Following Questions?

- ✎ What is the easiest way to avoid the nested monitor problem?
- ✎ What assumptions do nested monitors violate?
- ✎ How can the obvious implementation of semaphores (in Java) violate fairness?
- ✎ How does “partial synchronization” differ from “concurrently available methods”?
- ✎ When should you prefer optimistic methods to guarded methods?

7. Lab session

8. Architectural Styles for Concurrency

Overview

- ❑ What is Software Architecture?
- ❑ Three-layered application architecture
- ❑ Flow architectures
- ❑ Blackboard architectures

Sources

- ❑ M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.
- ❑ F. Buschmann, et al., Pattern-Oriented Software Architecture — A System of Patterns, John Wiley, 1996.
- ❑ D. Lea, Concurrent Programming in Java — Design principles and Patterns, The Java Series, Addison-Wesley, 1996.
- ❑ N. Carriero and D. Gelernter, How to Write Parallel Programs: a First Course, MIT Press, Cambridge, 1990.

Software Architecture

A Software Architecture defines a system in terms of computational components and interactions amongst those components.

An Architectural Style defines a family of systems in terms of a pattern of structural organization.

— cf. Shaw & Garlan, Software Architecture, pp. 3, 19

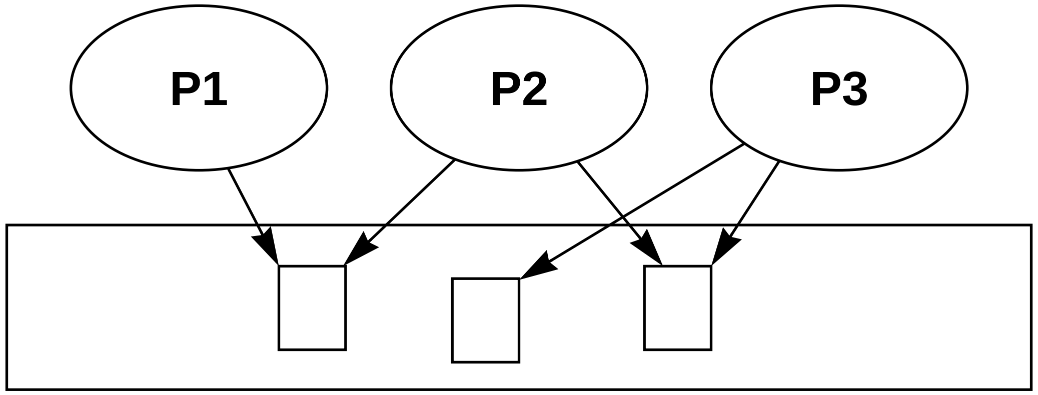
Architectural style

Architectural styles typically entail four kinds of properties:

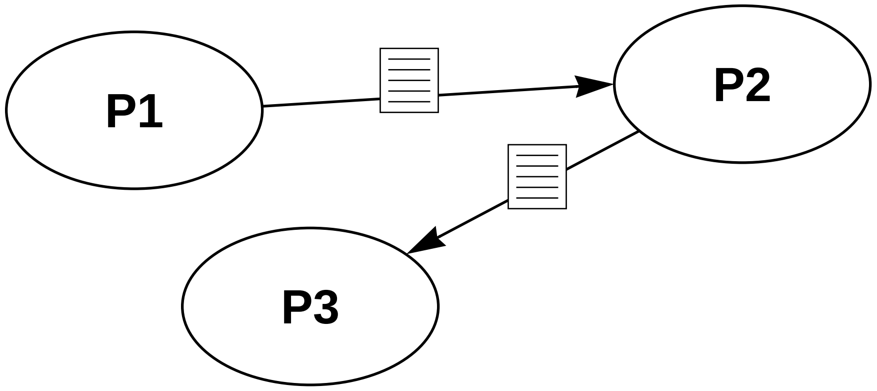
- ❑ A vocabulary of design elements
  - ☞ e.g., “pipes”, “filters”, “sources”, and “sinks”
- ❑ A set of configuration rules that constrain compositions
  - ☞ e.g., pipes and filters must alternate in a linear sequence
- ❑ A semantic interpretation
  - ☞ e.g., each filter reads bytes from its input stream and writes bytes to its output stream
- ❑ A set of analyses that can be performed
  - ☞ e.g., if filters are “well-behaved”, no deadlock can occur, and all filters can progress in tandem

Communication Styles

Shared Variables:

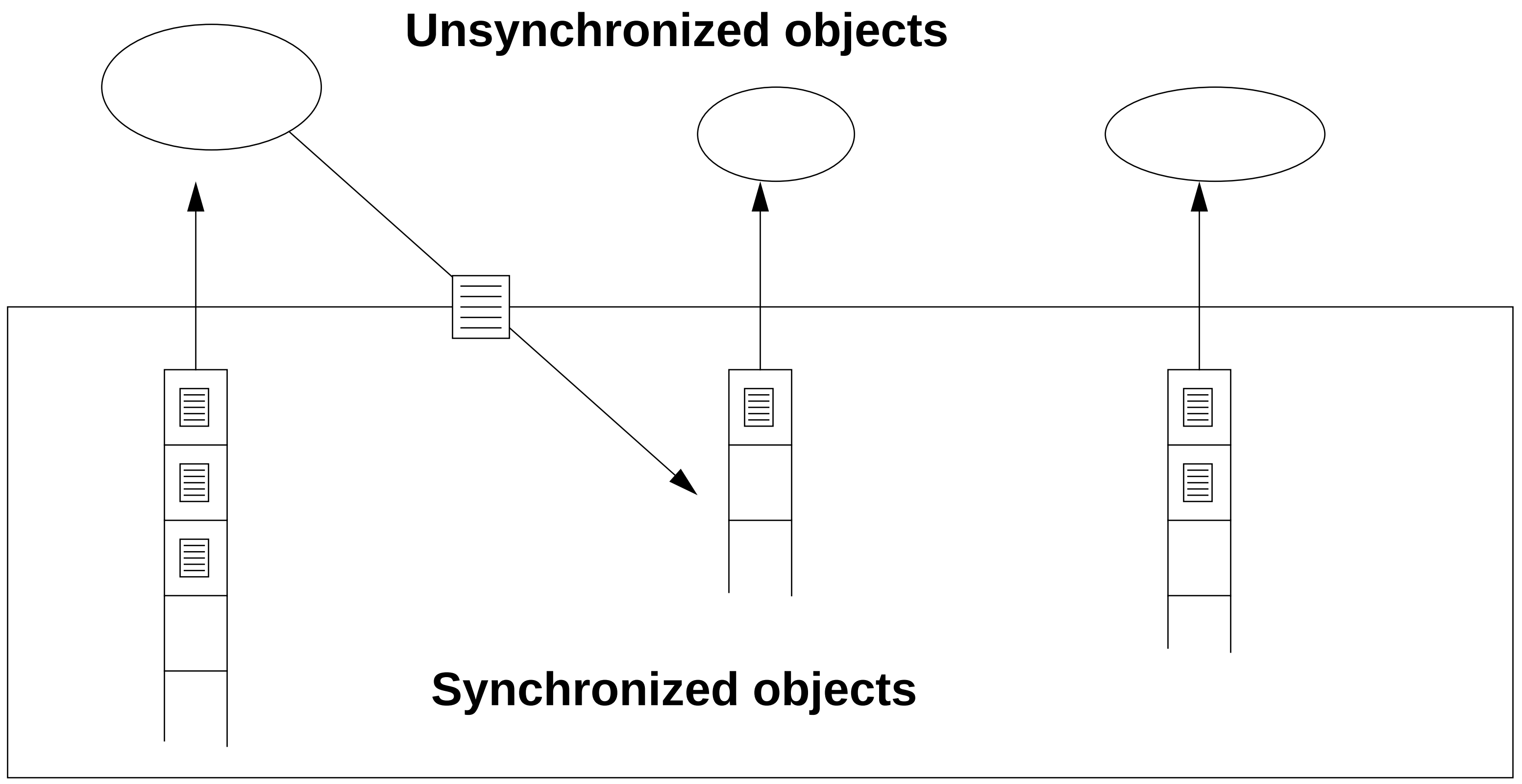


Message-Passing:



Simulated Message-Passing

Most concurrency and communication styles can be simulated by one another:



Message-passing can be modelled by associating message queues to each process.

Three-layered Application Architectures

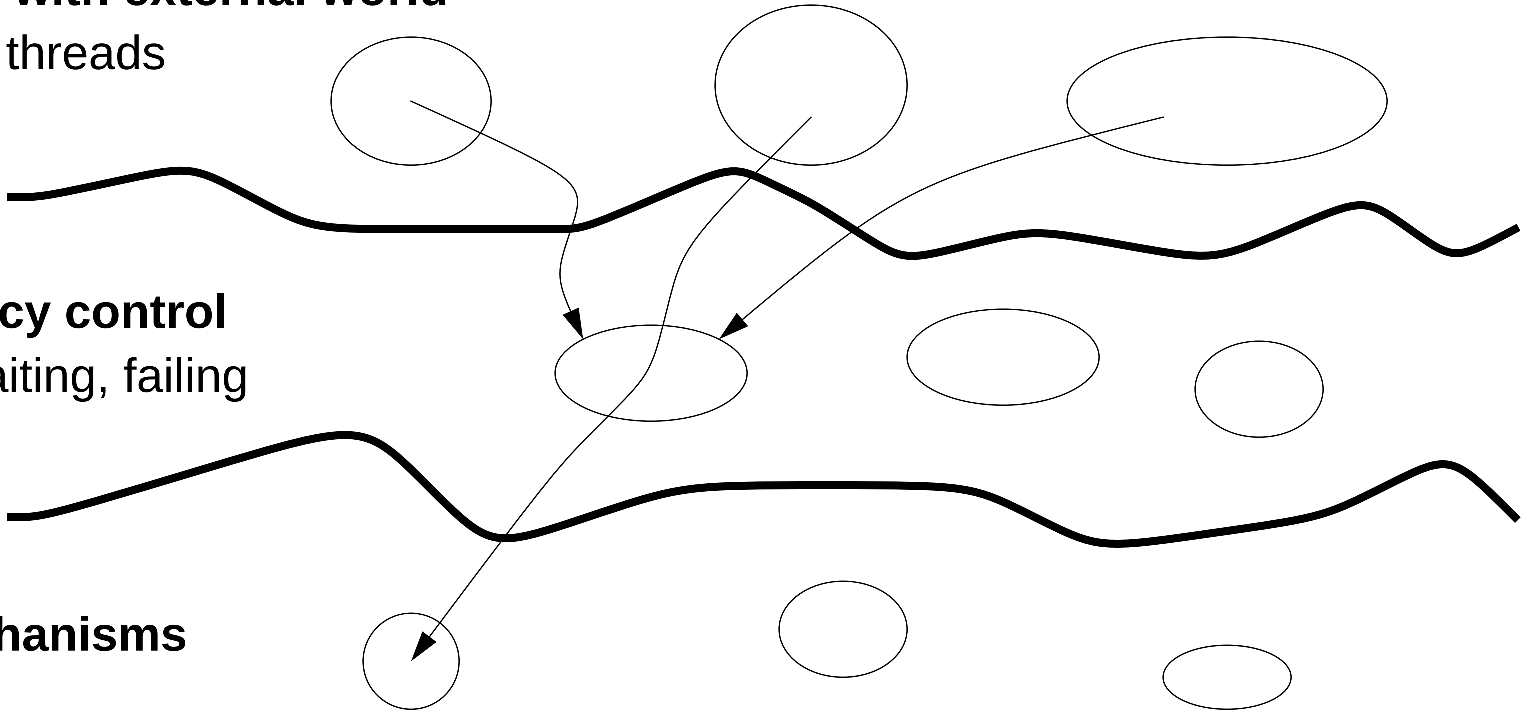
Interaction with external world

Generating threads

Concurrency control

Locking, waiting, failing

Basic mechanisms



This kind of architecture avoids nested monitor problems by restricting concurrency control to a single layer.

Problems with Layered Designs

Hard to extend beyond three layers because:

- ❑ Control is restricted to before/after — not within
- ❑ Control may depend on unavailable information
  - ☞ Because it is not safely accessible
  - ☞ Because it is not represented (e.g., message history)
- ❑ Actions in control code may encounter conflicting policies
  - ☞ E.g., nested monitor lockouts
- ❑ Ground actions may need to know current policy
  - ☞ E.g., blocking vs. failing

Partial solutions:

- ❑ Explicit policy compatibility constraints
- ❑ Explicit nesting constraints
- ❑ Delegated control

Flow Architectures

Many synchronization problems can be avoided by arranging things so that information only flows in one direction from sources to filters to sinks:

- ❑ Unix “pipes and filters”:
  - ☞ Processes are connected in a linear sequence
- ❑ Control systems:
  - ☞ events are picked up by sensors, processed, and generate new events
- ❑ Workflow systems
  - ☞ Electronic documents flow through workflow procedures

Flow Stages

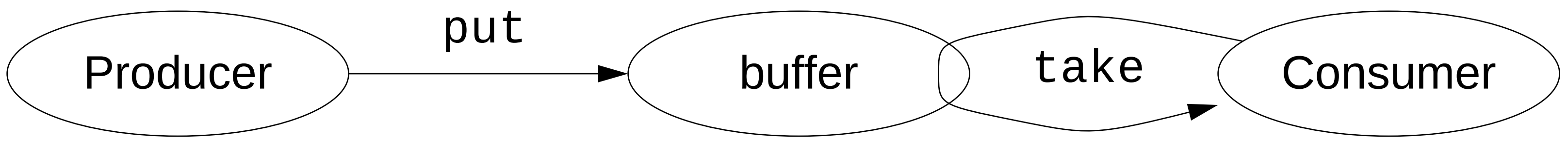
Every flow stage is a producer or consumer or both:

- ❑ Splitters (Multiplexers) have multiple successors
  - ☞ Multicasters clone results to multiple consumers
  - ☞ Routers distribute results amongst consumers
- ❑ Mergers (Demultiplexers) have multiple predecessors
  - ☞ Collectors interleave inputs to a single consumer
  - ☞ Combiners process multiple input to produce a single result
- ❑ Conduits have both multiple predecessors and consumers

Flow Policies

Flow can be pull-based, push-based, or a mixture:

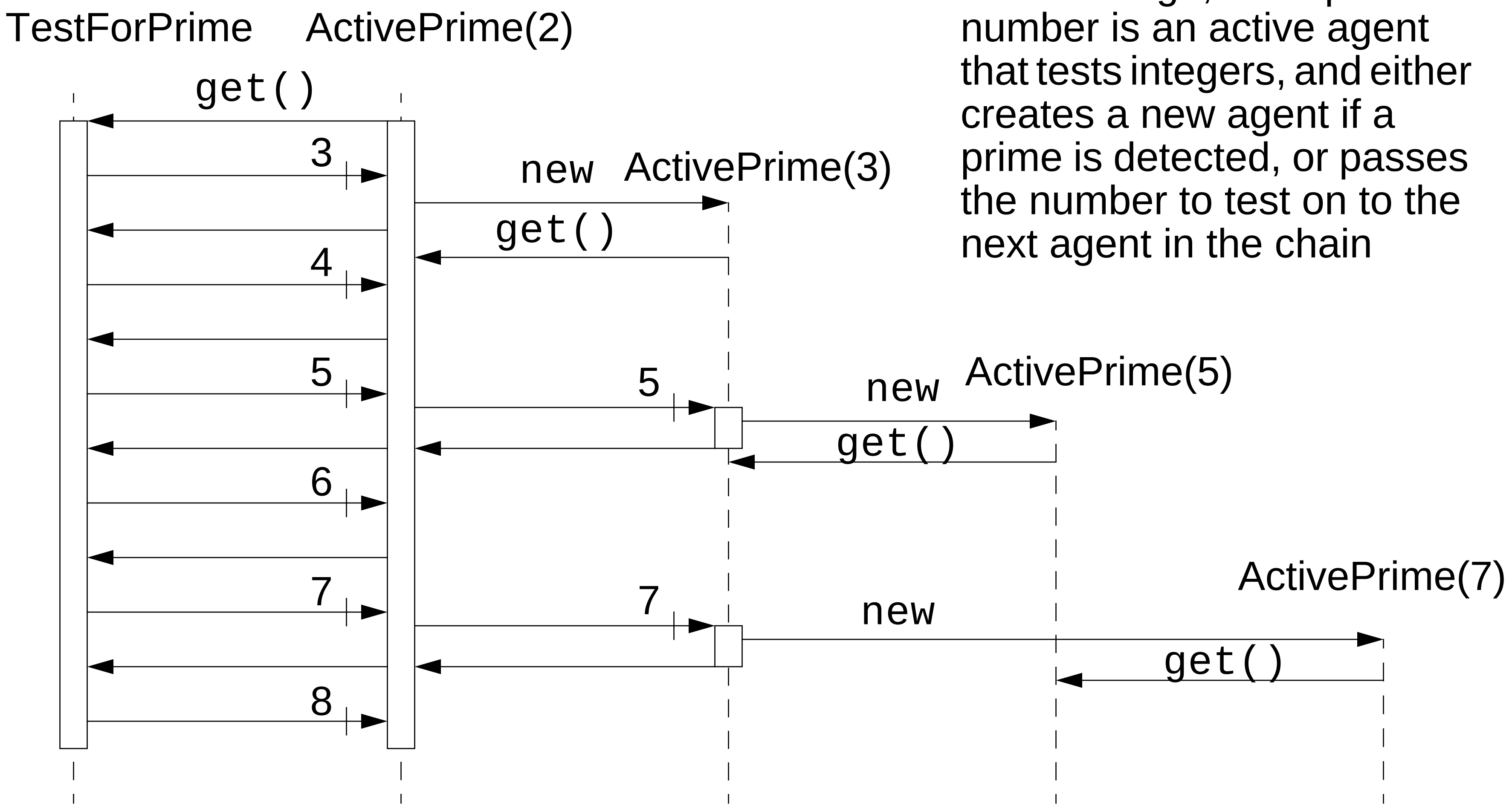
- ❑ Pull-based flow: Consumers take results from Producers
- ❑ Push-based flow: Producers put results to Consumers
- ❑ Buffers:
  - ☞ Put-only buffers (relays) connect push-based stages
  - ☞ Take-only buffers (pre-fetch buffers) connect pull-based stages
  - ☞ Put-Take buffers connect push-based stages to pull-based stages



Limiting Flow

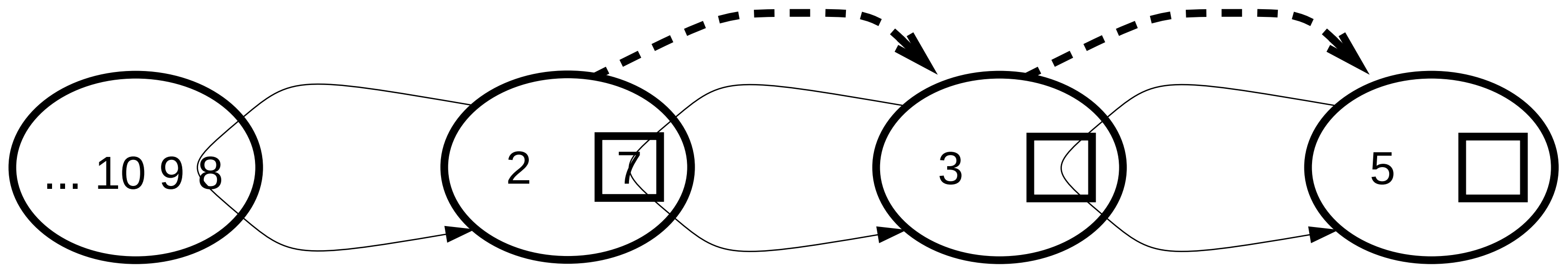
- ❑ Unbounded buffers:
  - ☞ If producers are faster than consumers, buffers may exhaust available memory
- ❑ Unbounded threads:
  - ☞ Having too many threads can exhaust system resources more quickly than unbounded buffers
- ❑ Bounded buffers:
  - ☞ Tend to be either always full or always empty, depending on relative speed of producers and consumers
- ❑ Bounded thread pools:
  - ☞ Harder to manage than bounded buffers

Example: a Pull-based Prime Sieve



In this design, each prime number is an active agent that tests integers, and either creates a new agent if a prime is detected, or passes the number to test on to the next agent in the chain

Using Put-Take Buffers



Each ActivePrime will use a one-slot buffer to feed values to the next ActivePrime.

```
public class PrimeSieve {
  public static void main(String args[]) { genPrimes(1000); }
  public static void genPrimes(int n) {
    try {
      ActivePrime firstPrime =
              new ActivePrime(2, new TestForPrime(n));
    } catch (Exception e) { }
  }
}
```

The first ActivePrime holds the seed value 2, gets values from a TestForPrime, and creates new ActivePrime instances whenever it detects a prime value.

Pull-based integer sources

Active primes get numbers to test from an `IntSource` interface:

```
interface IntSource {
   int getInt();
}
class TestForPrime implements IntSource {
   private int nextValue;
   private int maxValue;
   public TestForPrime(int max) {
      this.nextValue = 3;
      this.maxValue = max;
   }
   public int getInt() {        // No synchronization needed
      if (nextValue < maxValue) { return nextValue++; }
      else { return 0; }
   }
}
```

One-Slot Buffer

```
class Slot {                                        // a one-slot buffer
   private Object slotVal_;                         // initially null
   public synchronized void put(Object val) {       // put contents,
      while (slotVal_ != null) {                    // if there is room
         try { wait(); }                            // otherwise wait
         catch (InterruptedException e) { }
      }
      slotVal_ = val;
      notifyAll();                                  // wake up waiting consumer
      return;
   }
   public synchronized Object get() {               // return contents,
      Object rval;
      while (slotVal_ == null) {                    // if available
         try { wait(); }                            // otherwise wait
         catch (InterruptedException e) { }
      }
      rval = slotVal_;
      slotVal_ = null;
      notifyAll();                                  // wake up waiting producer
      return rval;
   }
}
```

The ActivePrime Class

```
class ActivePrime extends Thread implements IntSource {
  private static IntSource lastPrime;  // end of the chain
  private int value;                 // value of this prime
  private int square;                // square of this prime
  private IntSource intSrc;          // source of ints to test
  private Slot slot;                 // to pass values on
  public ActivePrime(int value, IntSource intSrc)
    throws ActivePrimeFailure
  {
    this.value = value;
    this.square = value*value;
    this.intSrc = intSrc;
    slot = new Slot();               // NB: private
    lastPrime = this;                // unsynchronized (!)
    System.out.print(value + " ");
    System.out.flush();
    this.start();                    // become active
  }
```

The ActivePrime Class ...

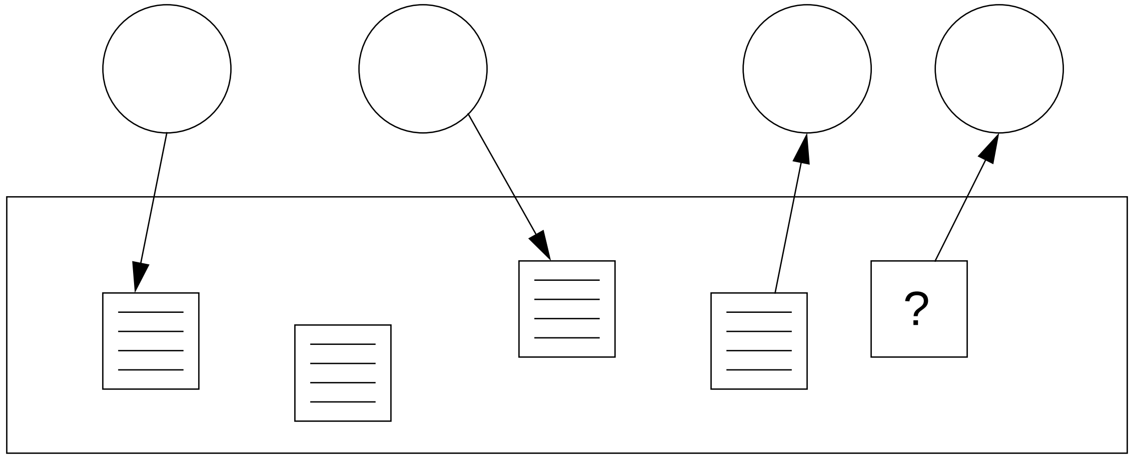
```
...
  public int value() {
    return this.value;
  }
  private void putInt(int val) {
    slot.put(new Integer(val));                      // may block
  }
  public int getInt() {
    return ((Integer) slot.get()).intValue(); // may block
  }
...
```

The ActivePrime Class ...

```
...
  public void run() {
    int testValue = intSrc.getInt();   // may block
    while (testValue != 0) {           // stop condition
      if (this.square > testValue) {   // must be prime
        try {
          new ActivePrime(testValue, lastPrime);
        } catch (Exception e) {
          break;                       // exit loop
        }
      } else if ((testValue % this.value) > 0) {
        this.putInt(testValue);        // may block
      }
      testValue = intSrc.getInt();     // may block
    }
    putInt(0);                         // stop next prime
  }
}
```

Blackboard Architectures

Blackboard architectures put all synchronization in a “coordination medium” where agents can exchange messages.

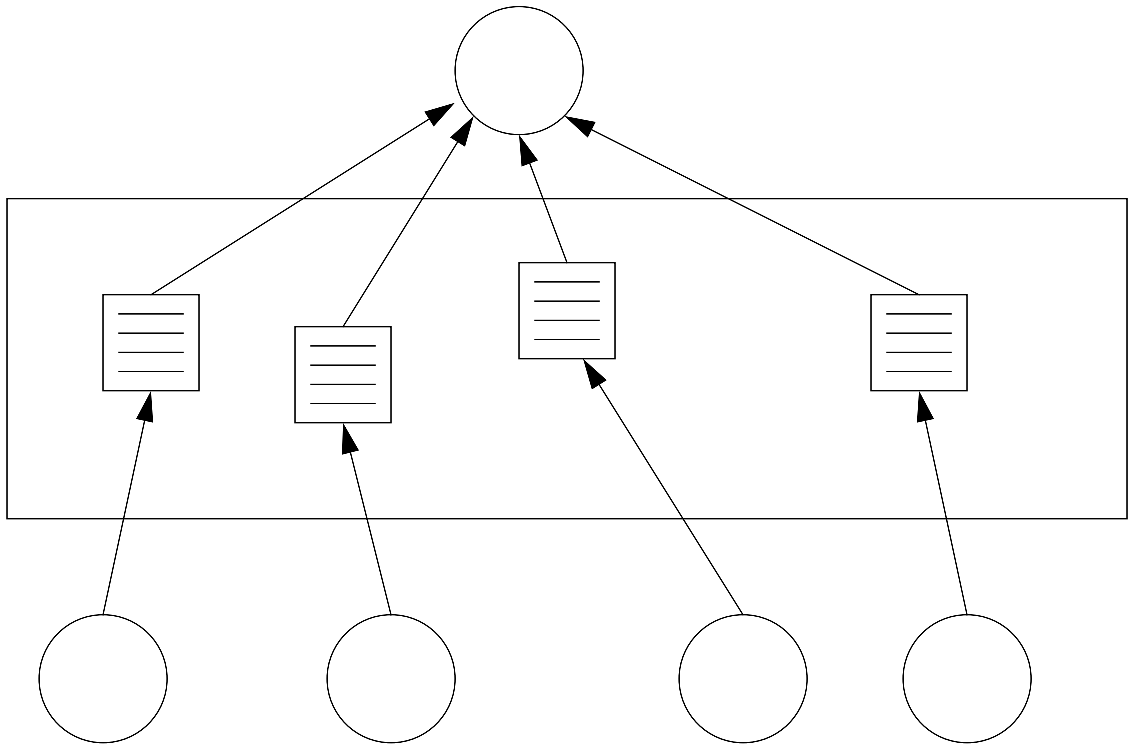


Agents do not exchange messages directly, but post messages to the blackboard, and retrieve messages either by reading from a specific location (i.e., a channel), or by posing a query (i.e., a pattern to match).

Linda is a “coordination language” that provides primitives for implementing blackboard architectures ...

Result Parallelism

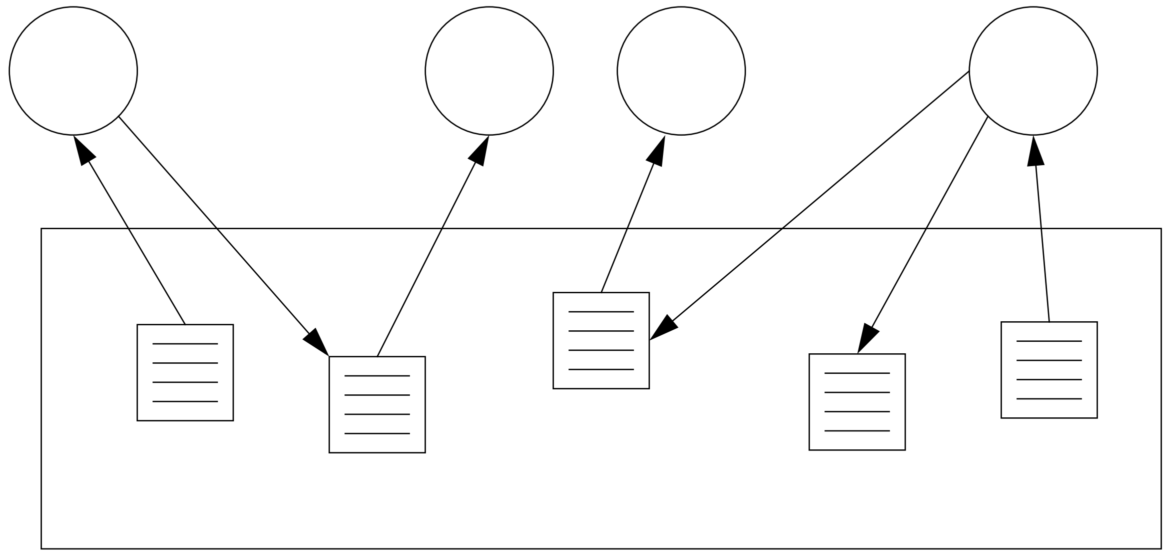
Result parallelism is a blackboard architectural style in which workers are spawned to produce each part of a more complex problem.



Workers may be arranged hierarchically ...

Agenda Parallelism

Agenda parallelism is a blackboard style in which workers retrieve tasks to perform from a blackboard, and may generate new tasks to perform.

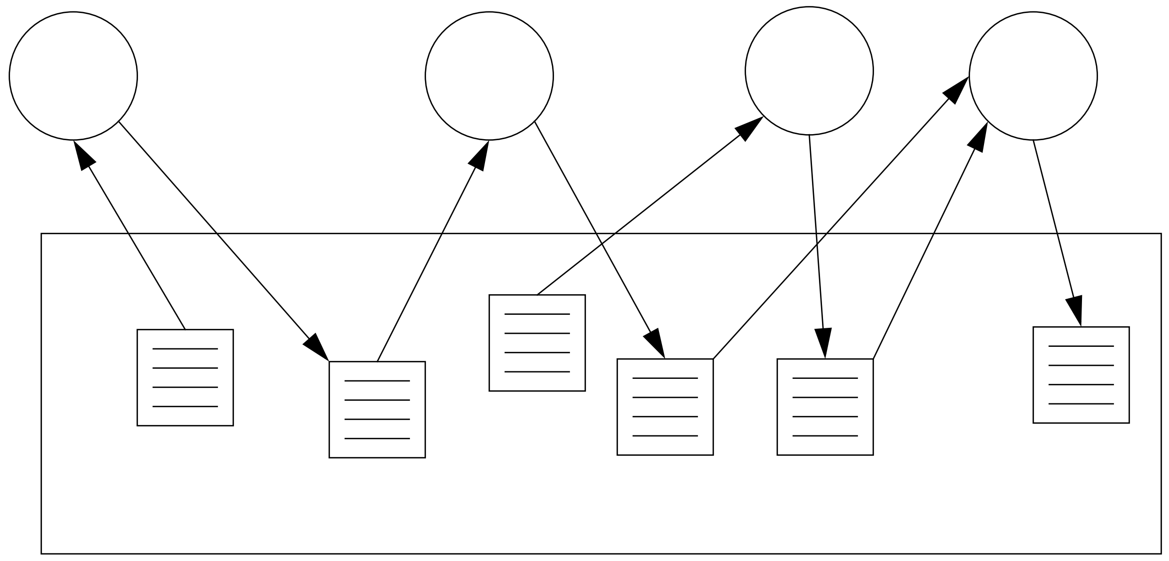


Workers repeatedly retrieve tasks until everything is done.

Workers are typically able to perform arbitrary tasks.

Specialist Parallelism

Specialist parallelism is a style in which each workers is specialized to perform a particular task.



Specialist designs are equivalent to message-passing, and are generally organized as flow architectures, with each specialist producing results for the next specialist to consume.

Summary

You Should Know The Answers To These Questions:

- ❑ What is a Software Architecture?
- ❑ What are advantages and disadvantages of Layered Architectures?
- ❑ What is a Flow Architecture? What are the options of tradeoffs?
- ❑ What are Blackboard Architectures? What are the options and tradeoffs?

Can You Answer The Following Questions?

- ✎ How would you model message-passing agents in Java?
- ✎ How would you classify Client/Server architectures?
  Are there other useful styles we haven’t yet discussed?
- ✎ How can we prove that the Active Prime Sieve is correct? Are you sure that new Active Primes will join the chain in the correct order?
- ✎ Which Blackboard styles are better when we have multiple processors?
  Which are better when we just have threads on a monoprocessor?

9. Concurrent Programming Approaches

Overview

- ❑ Process creation
  - ☞ Co-routines; Fork & Join; Cobegin blocks
- ❑ Communication and Synchronization
  - ☞ Synchronizing access to shared variables
  - ☞ Message Passing Approaches

Texts:

- ❑ G. R. Andrews and F. B. Schneider, “Concepts and Notations for Concurrent programming,’“ACM Computing Surveys, vol. 15, no. 1, Mar. 1983, pp. 3-43.
- ❑ M. Ben-Ari, Principles of Concurrent and Distributed Programming, Prentice Hall, 1990.
- ❑ L. Wilson & R. Clark, Comparative Programming Languages, Addison-Wesley, 1988.

Expressing Concurrency

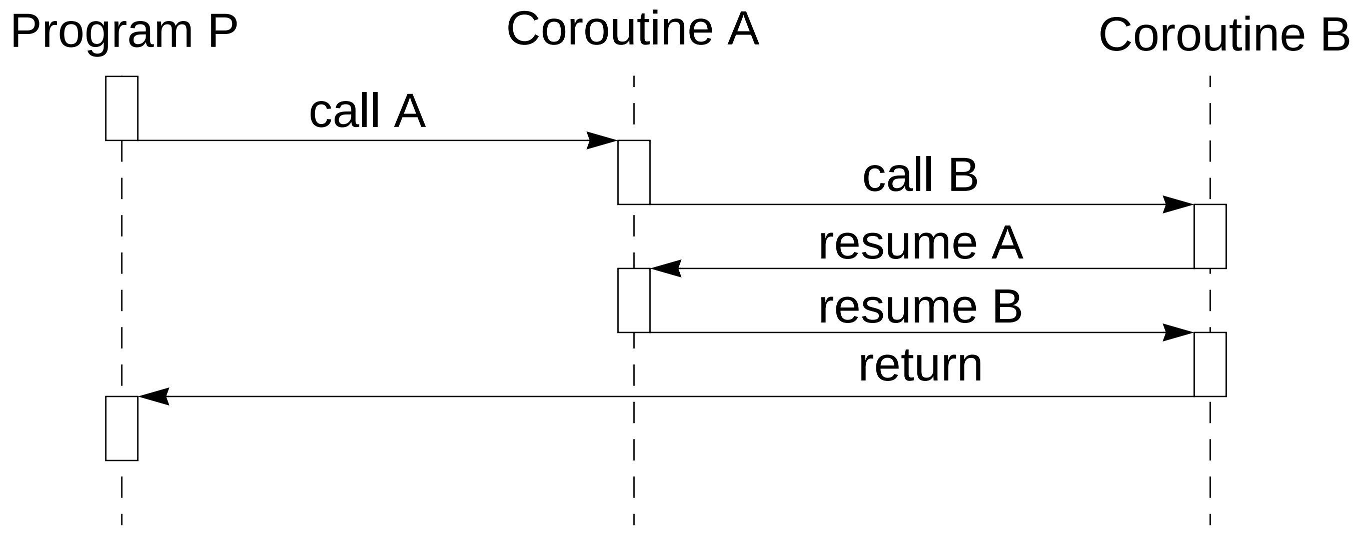
Recall:

Notations for expressing concurrent computation must address:

1. **Process Creation:** how is concurrent execution specified?
2. **Communication:** how do processes communicate?
3. **Synchronization:** how is consistency maintained?

Co-routines

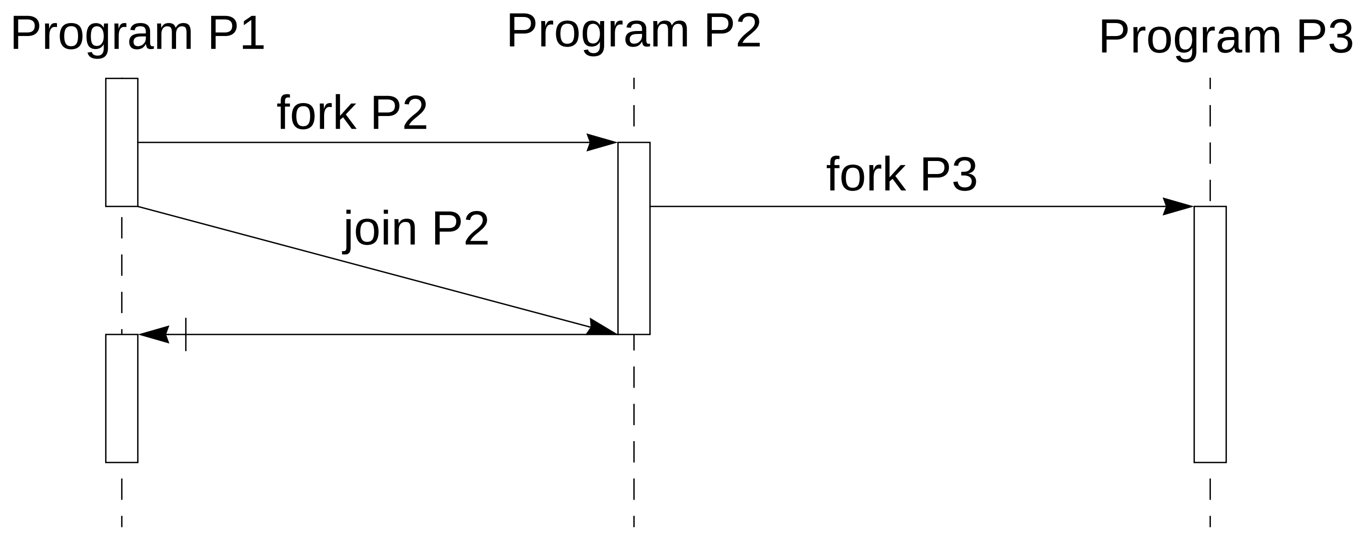
Co-routines are only pseudo-concurrent and require explicit transfers of control:



Co-routines can be used to implement most higher-level concurrent mechanisms.

Fork and Join

Fork can be used to create an arbitrary number of concurrent processes:



Join (“wait” in Unix) is used to wait for another process to terminate.

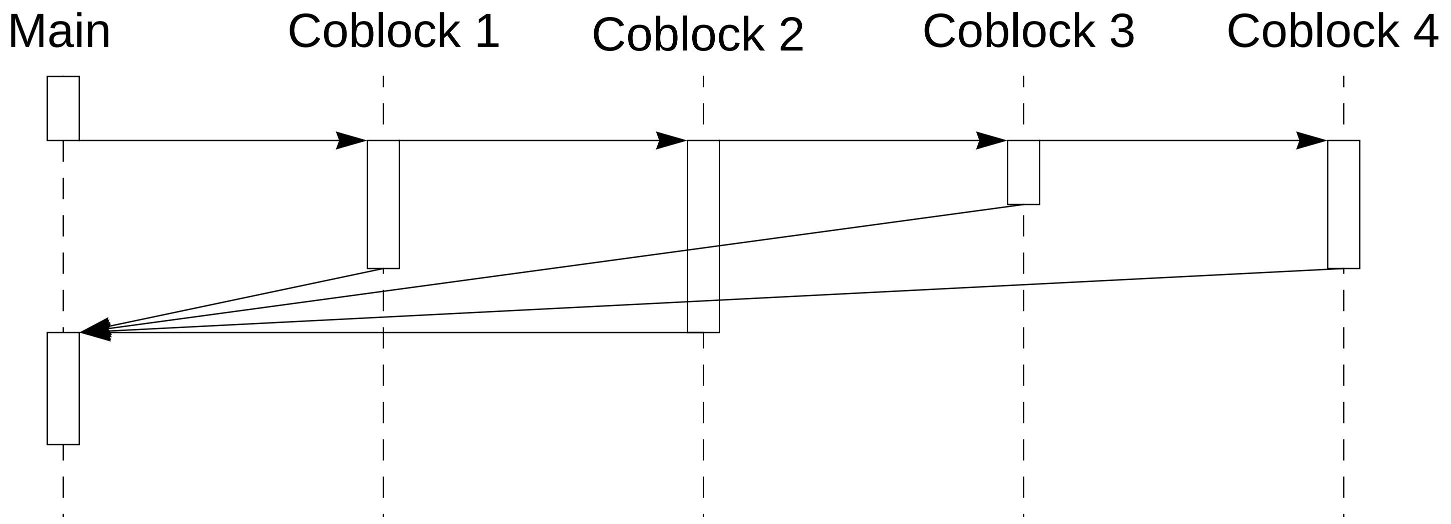
Since fork and join are unstructured, they must be used with care and discipline.

Cobegin/coend

Cobegin/coend blocks are better structured:

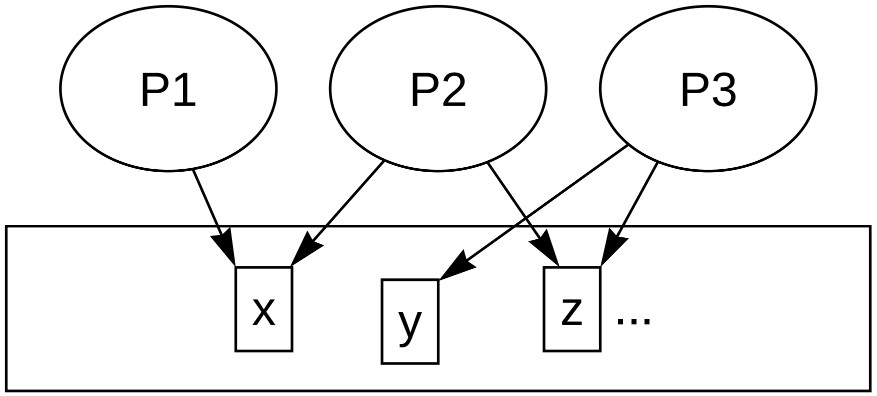
```
cobegin S1 || S2 || ... || Sn coend
```

but they can only be used to create a fixed number of processes.



The calling routine continues when all of the coblocks have terminated.

Communication and Synchronization

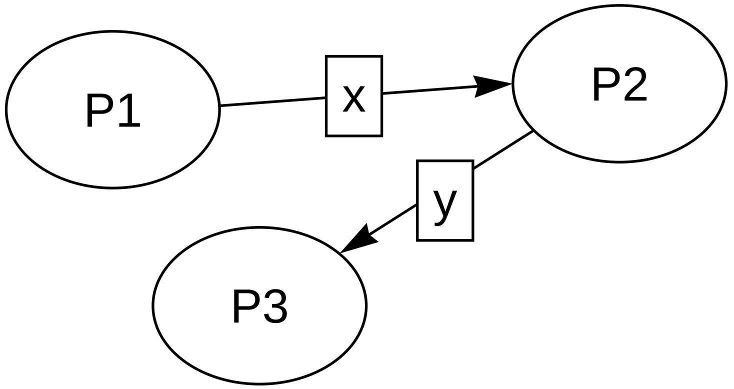


In approaches based on Shared Variables, processes communicate indirectly.

Explicit synchronization mechanisms are needed.

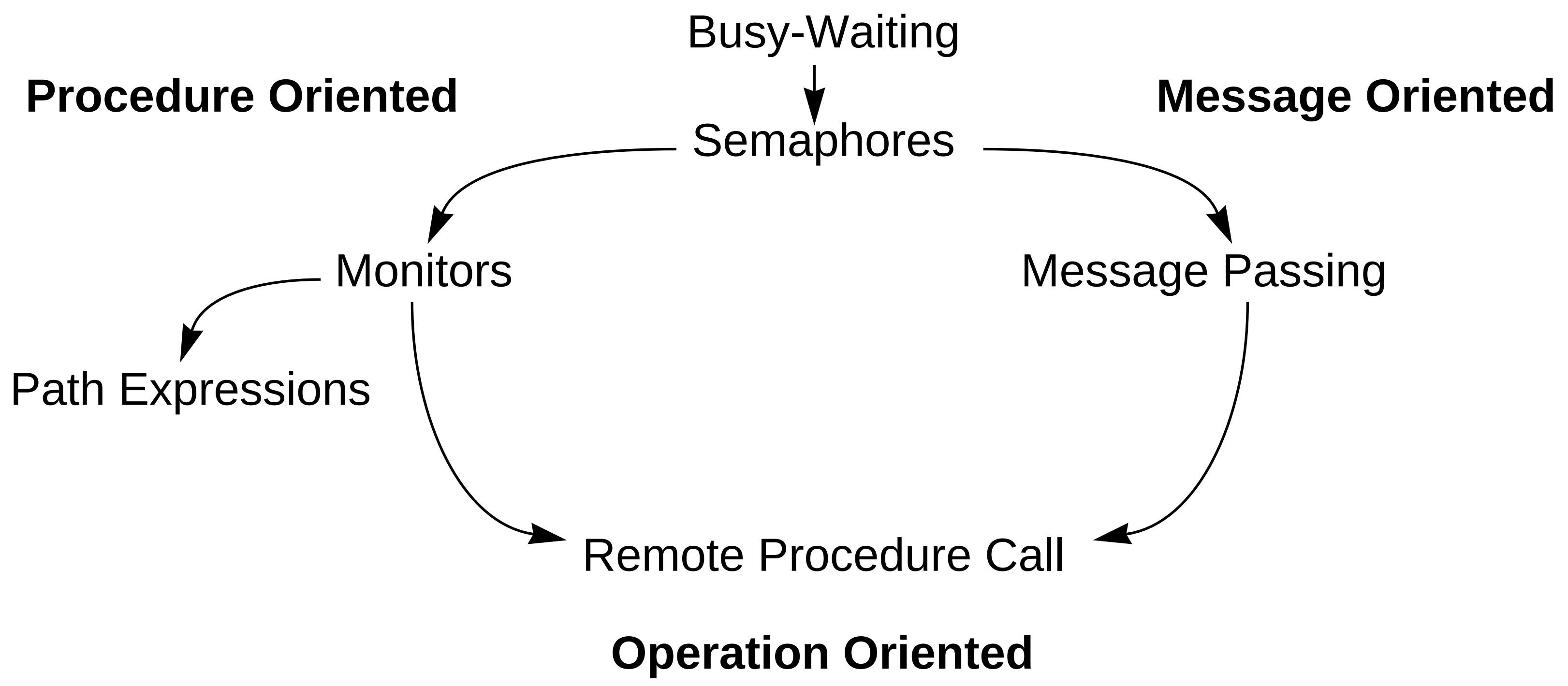
In Message Passing approaches, communication and synchronization are combined.

Communication may be either synchronous or asynchronous.



Synchronization Techniques

Different approaches are roughly equivalent in expressive power and can generally be implemented in terms of each other.



Each approach emphasizes a different style of programming.

Busy-Waiting

A simple approach to synchronization is for processes to atomically set and test shared variables.

Condition synchronization is easy to implement:

- ❑ to signal a condition, a process sets a shared variable
- ❑ to wait for a condition, a process repeatedly tests the variable

Mutual exclusion is more difficult to realize correctly and efficiently.

Busy-Wait Mutual Exclusion Protocol

P1 sets `enter1 := true` when it wants to enter its CS,
but sets `turn := “P2”` to yield priority to P2:

```
process P1
  loop
    enter1 := true
    turn := “P2”
    while enter2 and
          turn = “P2”
      do skip
    Critical Section
    enter1 := false
    Non-critical Section
  end
end
```

```
process P2
  loop
    enter2 := true
    turn := “P1”
    while enter1 and
          turn = “P1”
      do skip
    Critical Section
    enter2 := false
    Non-critical Section
  end
end
```

✎ Can you prove this protocol is correct? Is it fair? Deadlock-free?

Semaphores

Semaphores were introduced by Dijkstra (1968) as a higher-level primitive for process synchronization.

A semaphore is a non-negative integer-valued variable s with two operations:

- **P**(s): delays until s>0; when s>0, atomically executes s := s-1
- **V**(s): atomically executes s:= s+1

Many problems can be solved using binary semaphores, which take on values 0 or 1.

```
process P1                              process P2
   loop                                    loop
      P(mutex) { wants to enter }             P(mutex)
      Critical Section                        Critical Section
      V(mutex) { exits }                      V(mutex)
      Non-critical Section                    Non-critical Section
   end                                     end
end                                     end
```

Monitors

A monitor encapsulates resources and operations that manipulate them:

- ❑ operations are invoked with usual procedure call semantics
- ❑ procedure invocations are guaranteed to be mutually exclusive
- ❑ condition synchronization is realized using signal and wait primitives
  - ☞ there exist many variations of wait and signal ...

```
type buffer(T) = monitor
   var
   slots : array [0..N-1] of T;
   head, tail : 0..N-1;
   size : 0..N;
   notfull, notempty : condition;
procedure deposit(p : T);
   begin
      if size = N then notfull.wait
      slots[tail] := p;
      size := size + 1;
      tail := (tail+1) mod N;
      notempty.signal
   end
```

```
procedure fetch(var it : T);
   begin
      if size = 0 then notempty.wait
      it := slots[head];
      size := size - 1;
      head := (head+1) mod N;
      notfull.signal
   end
begin
   size := 0;
   head := 0;
   tail := 0;
end
```

Problems with Monitors

Although monitors provide a more structured approach to process synchronization than semaphores, they suffer from various shortcomings.

A signalling process is temporarily suspended to allow waiting processes to enter!

- ❑ Monitor state may change between signal and resumption of signaller
- ❑ Simultaneous signal and return is not supported
- ❑ Unlike with semaphores, multiple signals are not saved
- ❑ Boolean expressions are not explicitly associated to condition variables
- ❑ Nested monitor calls must be specially handled to prevent deadlock

Message Passing

Message Passing combines both communication and synchronization:

- ❑ A message is sent by specifying the message and a destination
  - ☞ The destination may be a process, a port, a set of processes, ...
- ❑ A message is received by specifying message variables and a source
  - ☞ The source may or may not be explicitly identified
  - ☞ Source and destination may be statically fixed or dynamically computed
- ❑ Message transfer may be synchronous or asynchronous
  - ☞ With asynchronous message passing, send operations never block
  - ☞ With buffered message passing, sent messages pass through a bounded buffer ; the sender may block if the buffer is full
  - ☞ With synchronous message passing, both the sender and receiver must be ready for a message to be exchanged

Unix Pipes

Unix pipes are bounded buffers that connect producer and consumer processes (sources, sinks and filters):

```
cat file                    # send file contents to output stream
| tr -c 'a-zA-Z' '\012'     # put each word on one line
| sort                      # sort the words
| uniq -c                   # count occurrences of each word
| sort -rn                  # sort in reverse numerical order
| more                      # and display the result
```

Processes should read from standard input and write to standard output streams.

☞ Misbehaving processes give rise to broken pipes!

Process creation and scheduling are handled by the O/S.
Synchronization is handled implicitly by the I/O system (through buffering).

Send and Receive

In CSP and Occam, source and destination are explicitly named:

```
PROC buffer(CHAN OF INT give, take, signal)
   VAL INT size IS 10:
   INT inindex, outindex, numitems:
   [size]INT thebuffer:
   SEQ
      numitems := 0
      inindex := 0
      outindex := 0
      WHILE TRUE
      ALT
         numitems ≤ size & give ? thebuffer[inindex]
            SEQ
               numitems := numitems + 1
               inindex := (inindex + 1) REM size
         numitems > 0 & signal ? any
            SEQ
               take ! thebuffer[outindex]
               numitems := numitems - 1
               outindex := (outindex + 1) REM size
```

Remote Procedure Calls and Rendezvous

In Ada, the caller identity need not be known in advance:

```
task body buffer is
   size : constant integer := 10;
   the_buffer : array (1 .. size) of item;
   no_of_items : integer range 0 .. size := 0;
   in_index, out_index : integer range 1 .. size := 1;
begin
   loop
      select
         when no_of_items < size =>
            accept give(x : in item) do
               the_buffer(in_index) := x;
            end give;
            no_of_items := no_of_items + 1;
            in_index := in_index mod size + 1;
      or when no_of_items > 0 =>
            accept take(x : out item) do
               x := the_buffer(out_index);
            end take;
            no_of_items := no_of_items - 1;
            out_index := out_index mod size + 1;
      end select;
   end loop;
end buffer;
```

Linda

Linda is a coordination medium, with associated primitives for coordinating concurrent processes, that can be added to an existing programming language.

The coordination medium is a tuple-space, which can contain:

- ❑ data tuples — tuples of primitives vales (numbers, strings ...)
- ❑ active tuples — expressions which are evaluated and turn into data tuples

The coordination primitives are:

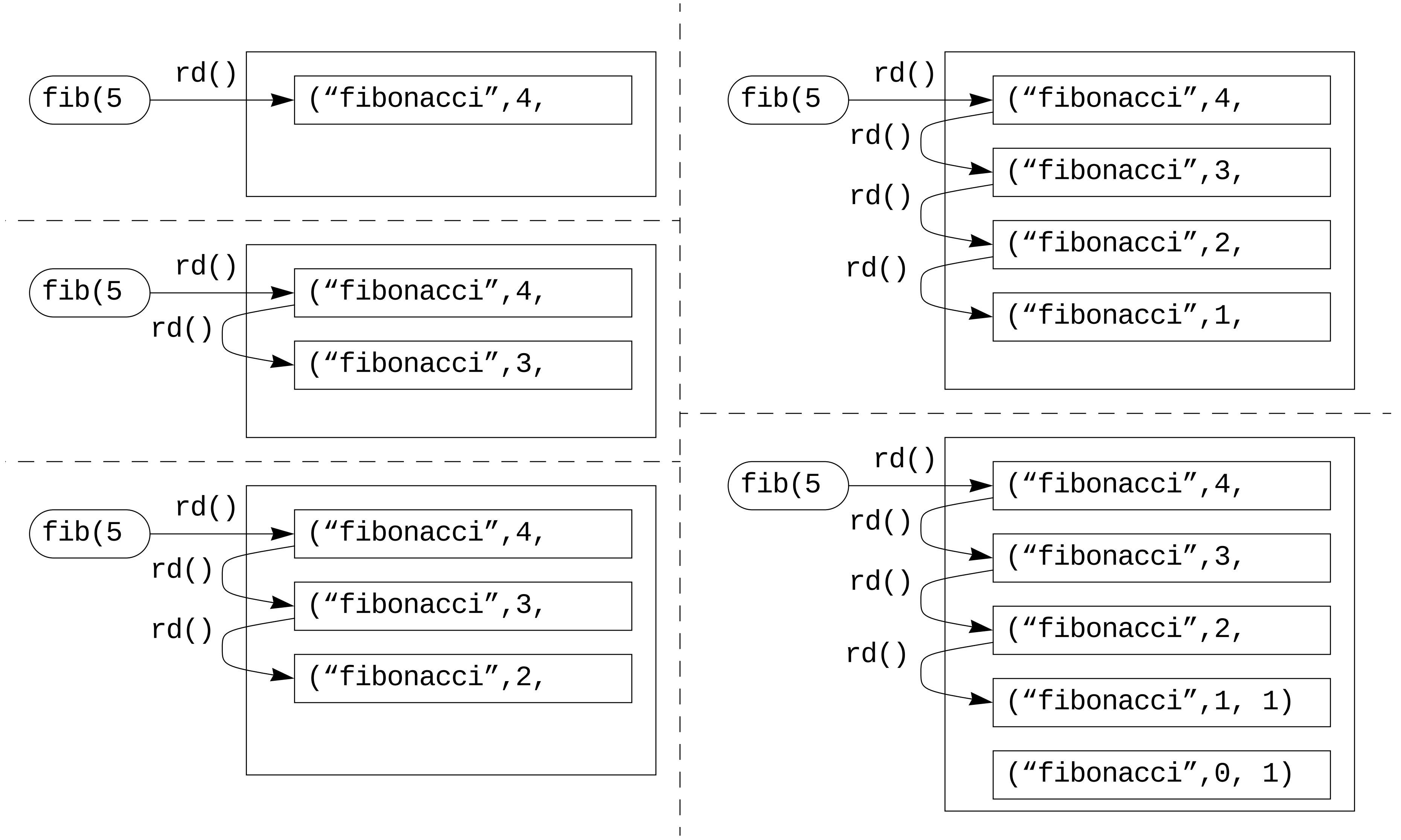
- ❑ **out(T)** — put a tuple T into the medium (non-blocking)
  - ☞ e.g., `out(“employee”, “pingu”, 35000)`
- ❑ **in(S)** — destructively input a tuple matching the pattern S (blocking)
  - ☞ e.g., `in(“employee”, “pingu”, ?salary)`
- ❑ **rd(S)** — non-destructively input a tuple (blocking)
- ❑ **inp(S), rdp(S)** — try to input a tuple; report success or failure (non-blocking)
- ❑ **eval(E)** — evaluate E in a new process; leave the result in the tuple space

Example: Fibonacci

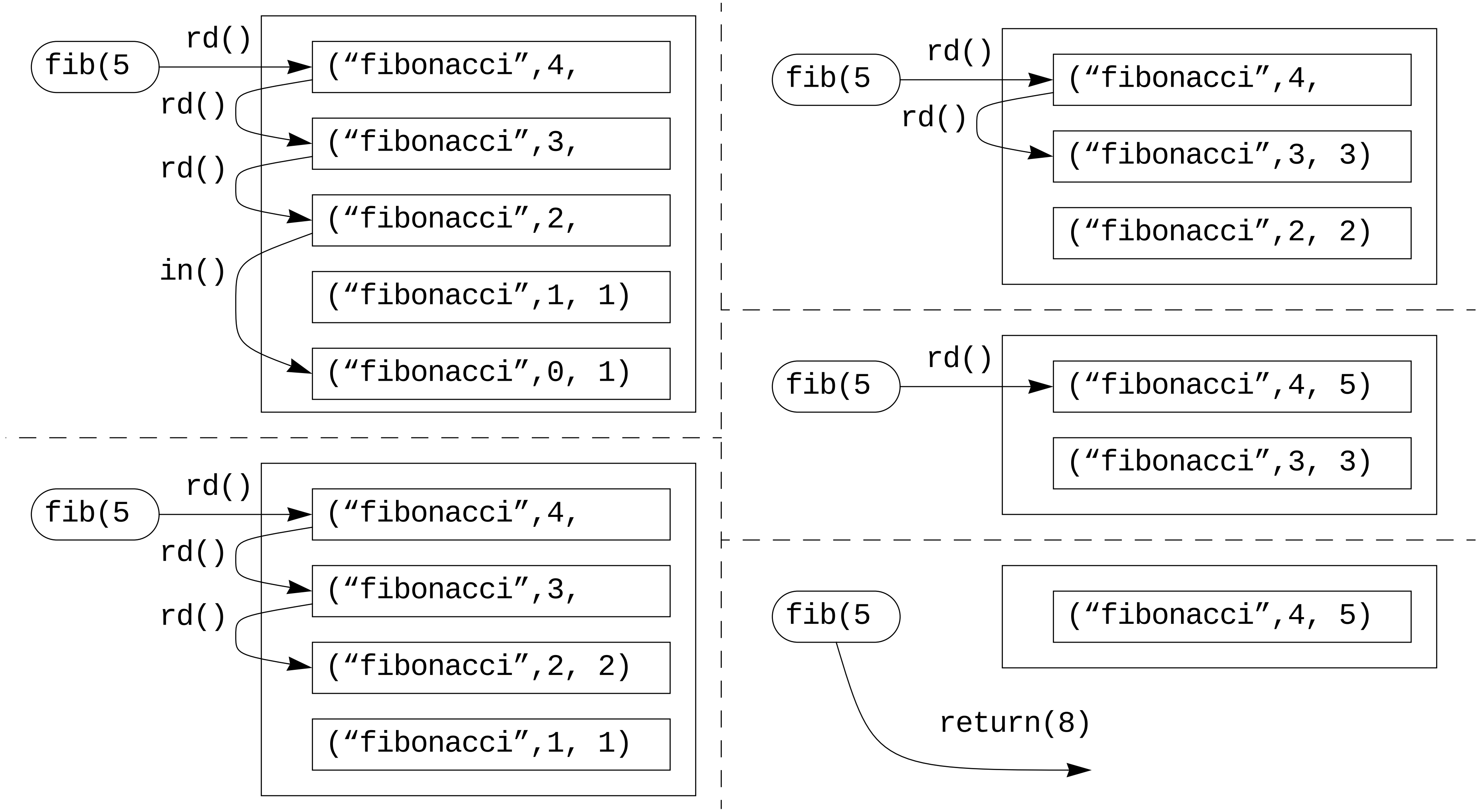
A (convoluted) way of computing fibonacci numbers with Linda:

```
int fibonacci(int n) {
  if (n<2) {
    out("fibonacci", n-1, 1);      // non-blocking
    return 1;
  }

  if (rdp("fibonacci",n-1,?fibn_1)==0) {   // non-blocking
    eval("fibonacci",n-1,fibonacci(n-1));  // async
  }

  rd("fibonacci",n-1,?fibn_1);   // blocking, non-destructive
  in("fibonacci",n-2,?fibn_2);   // blocking, destructive
  fn = fibn_2+fibn_1;
  return(fn);
} // Post-condition: rdp(fib(n-1)) == True
```

Evaluating Fibonacci



Evaluating Fibonacci ...



✎ What would happen if you ran fibonacci(5) twice?

Other Concurrency Issues

Atomic Transactions:

- ☞ RPC with possible failures
- ☞ failure atomicity
- ☞ synchronization atomicity

Real-Time Programming:

- ☞ embedded systems
- ☞ responding to interrupts within strict time limits

Summary

You Should Know The Answers To These Questions:

- ❑ How can you ensure mutual exclusion by busy-waiting?
- ❑ Are semaphores fair? In what way? Under what assumptions?
- ❑ How do monitors differ from semaphores?
- ❑ In what way are monitors equivalent to message-passing?
- ❑ What are “active tuples” in Linda?

Can You Answer The Following Questions?

- ✎ How could you implement a semaphore using monitors?
- ✎ How would you implement monitors using semaphores?
- ✎ Which concurrency mechanisms shown here are easily implemented in Java?
- ✎ Which of the known problems with monitors are also present in Java?
- ✎ How would you implement a message buffer in Linda?

10. Petri Nets

Overview

- ❑ Definition:
  - ☞ places, transitions, inputs, outputs
  - ☞ firing enabled transitions
- ❑ Modelling:
  - ☞ concurrency and synchronization
- ❑ Properties of nets:
  - ☞ liveness, boundedness
- ❑ Implementing Petri net models:
  - ☞ centralized and decentralized schemes

Sources

- ❑ J. L. Peterson, Petri Nets Theory and the Modelling of Systems, Prentice Hall, 1983.
- ❑ D. Lea, Concurrent Programming in Java — Design principles and Patterns, The Java Series, Addison-Wesley, 1996.

Petri nets: a definition

A Petri net C = ⟨P,T,I,O⟩ consists of:

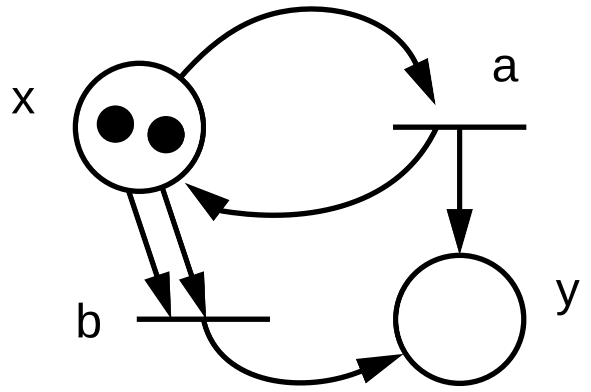
1. A finite set P of places
2. A finite set T of transitions
3. An input function I: T $\rightarrow \mathcal{N}^{P}$ (maps to bags of places)
4. An output function O: T $\rightarrow \mathcal{N}^{P}$

A marking of C is a mapping μ: P $\rightarrow \mathcal{N}$

Example:
P = { x, y }
T = { a, b }

I(a) = { x },	I(b) = { x, x }
O(a) = { x, y },	O(b) = { y }

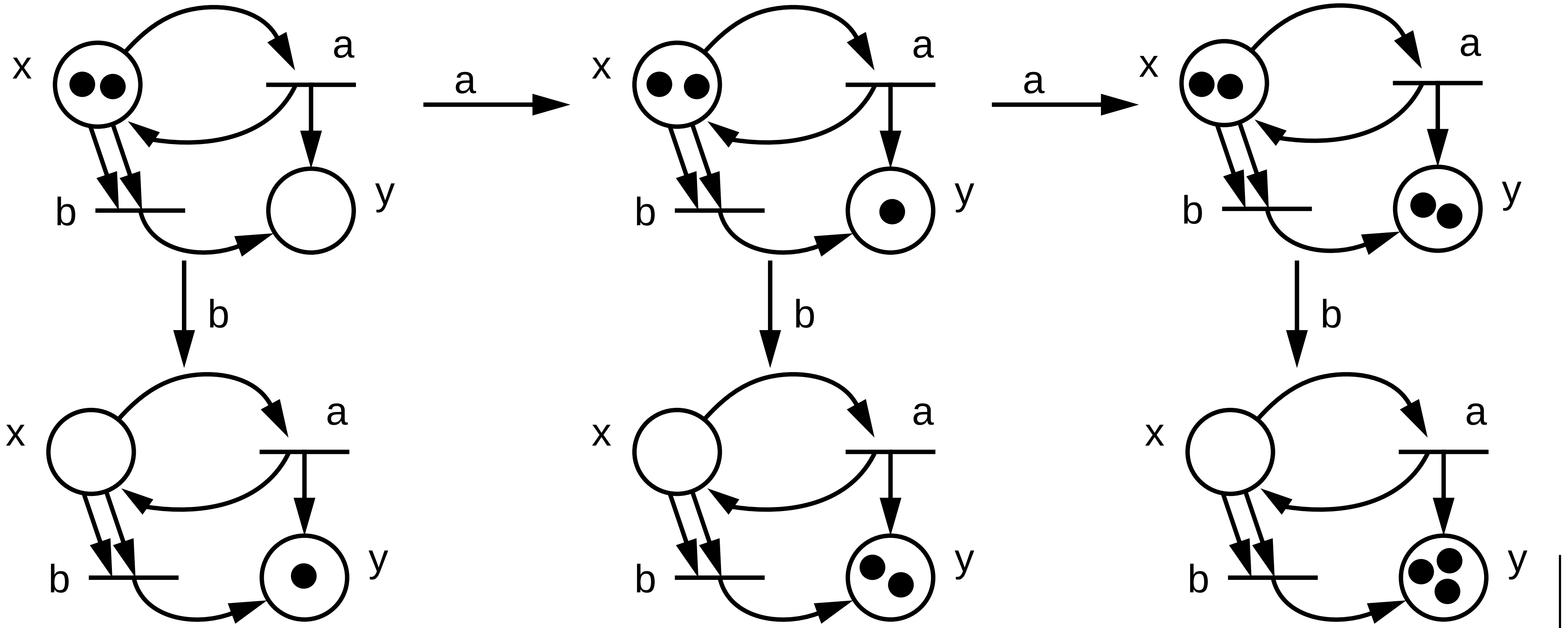
μ = { x, x }



Firing transitions

To fire a transition t:

1. There must be enough input tokens: $\mu \geq I(t)$
2. Consume inputs and generate output: $\mu' = \mu - I(t) + O(t)$



Modelling with Petri nets

Petri nets are good for modelling:

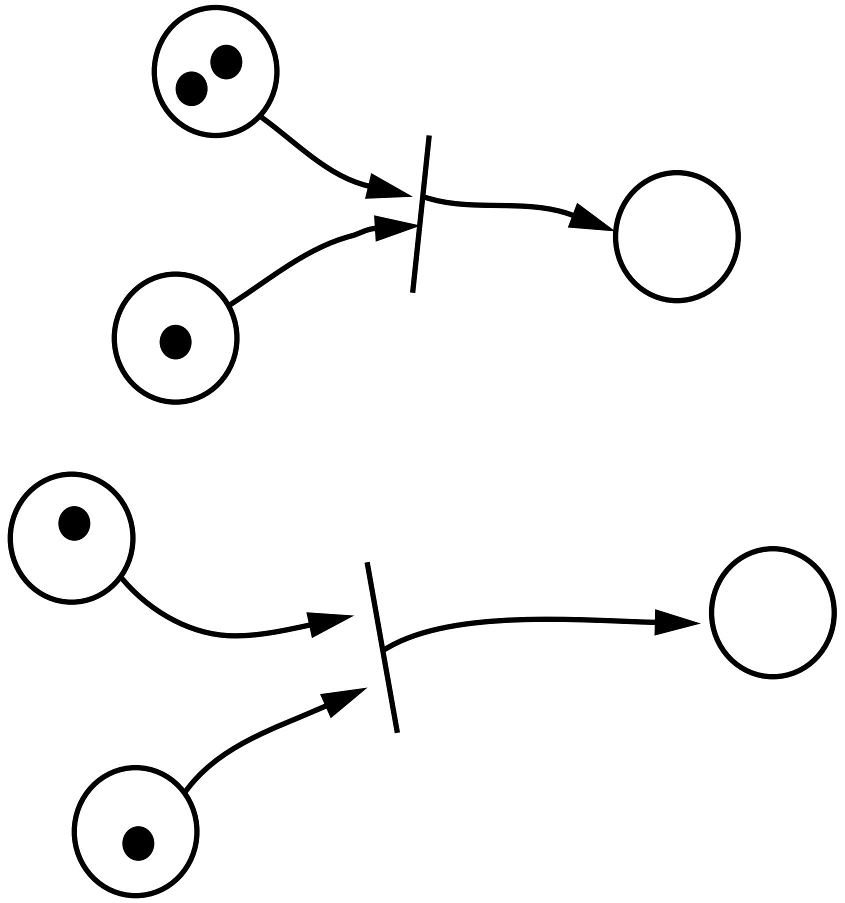
- concurrency
- synchronization

Tokens can represent:

- resource availability
- jobs to perform
- flow of control
- synchronization conditions ...

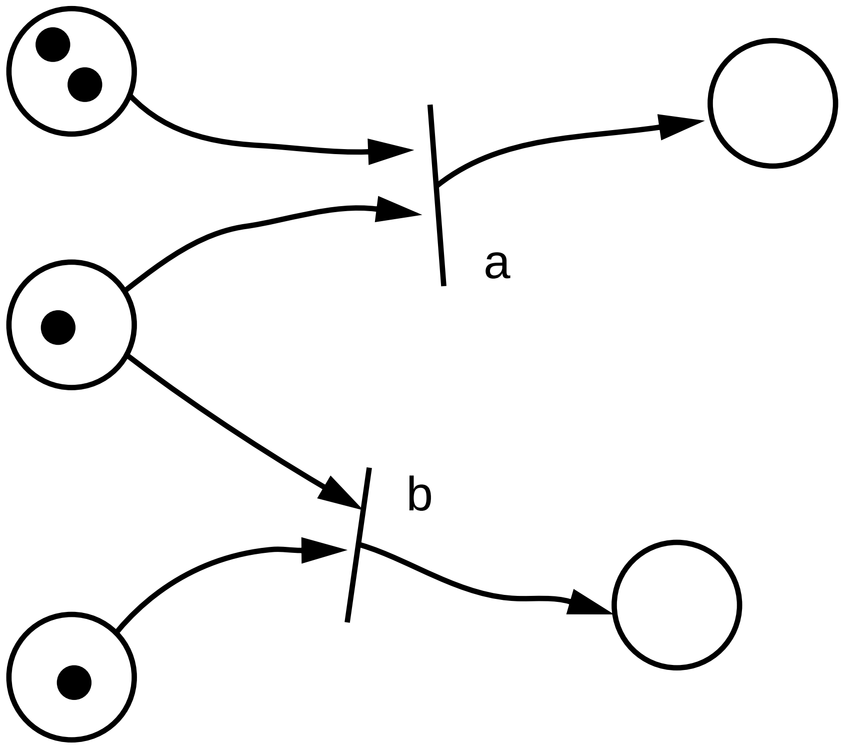
Concurrency

Independent inputs permit "concurrent" firing of transitions



Conflict

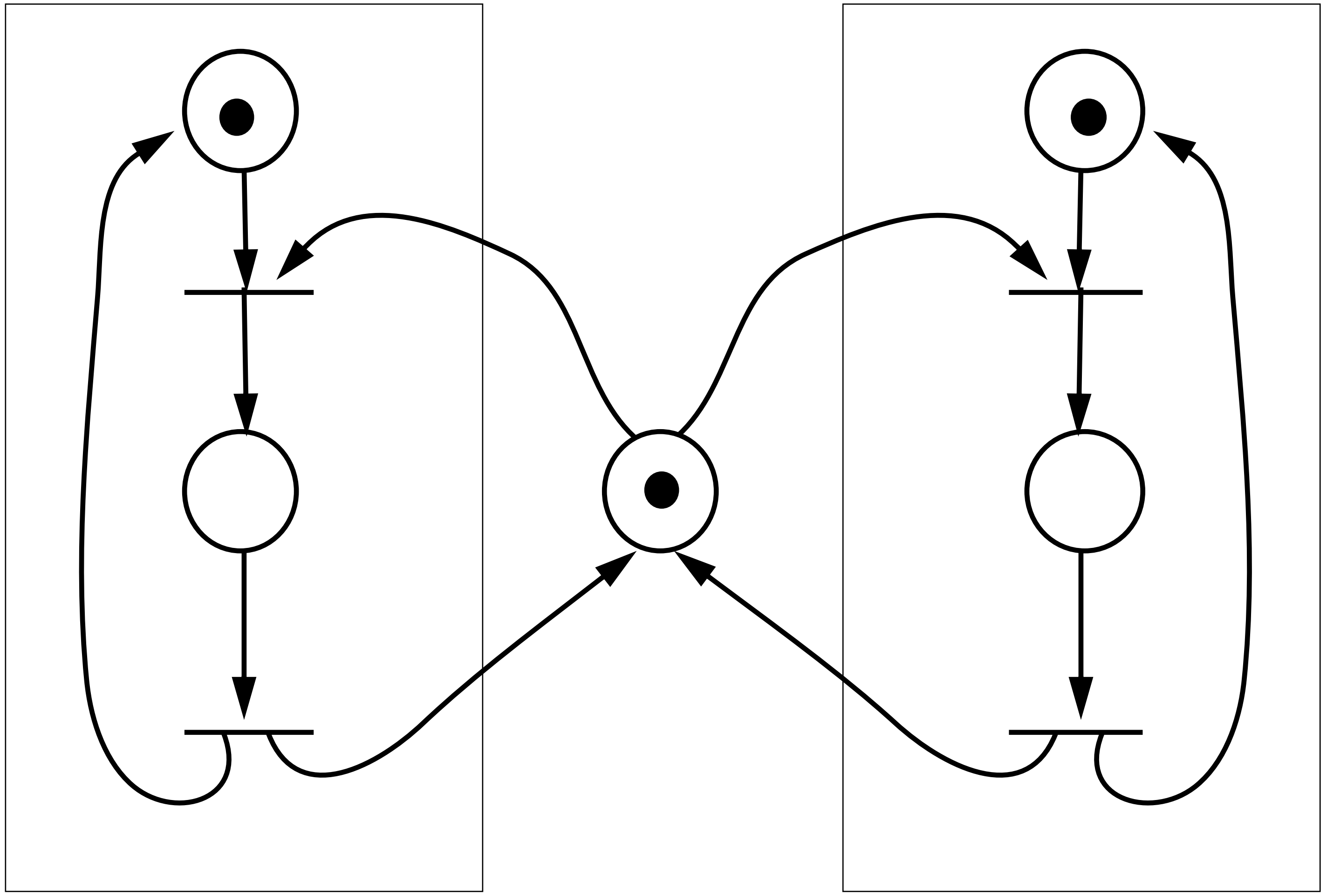
Overlapping inputs put transitions in conflict



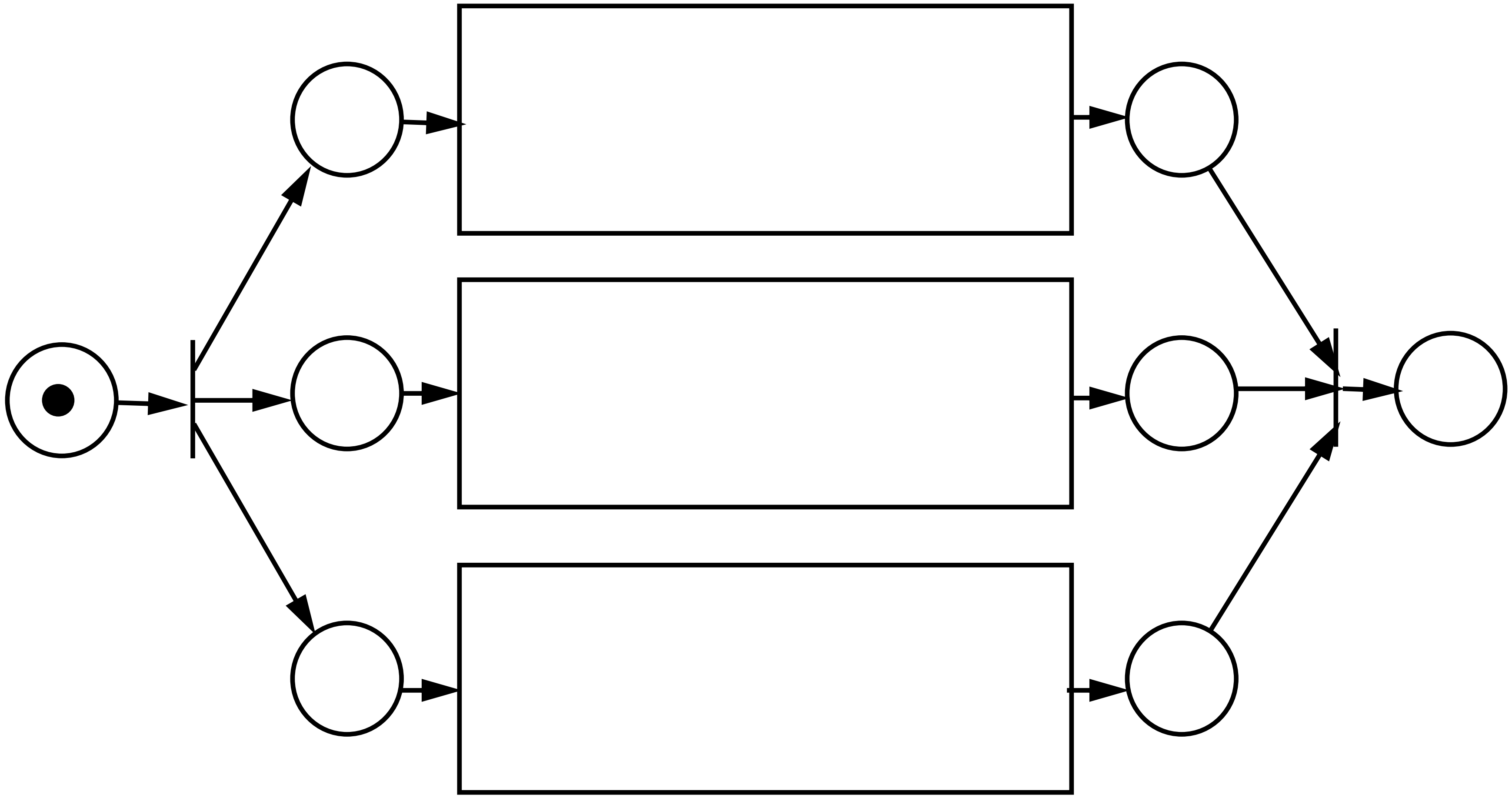
Only one of a or b may fire

Mutual Exclusion

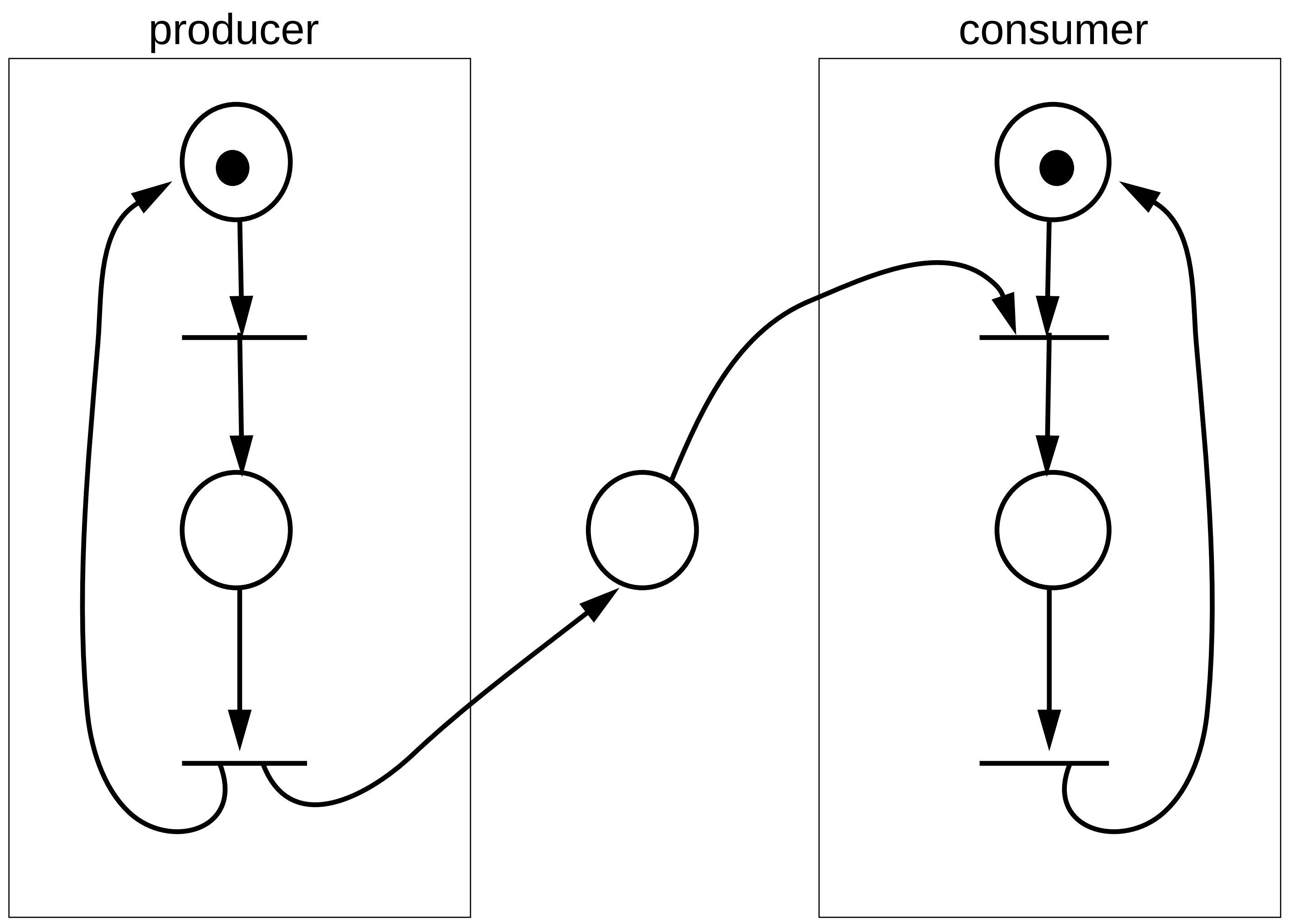
The two subnets are forced to synchronize



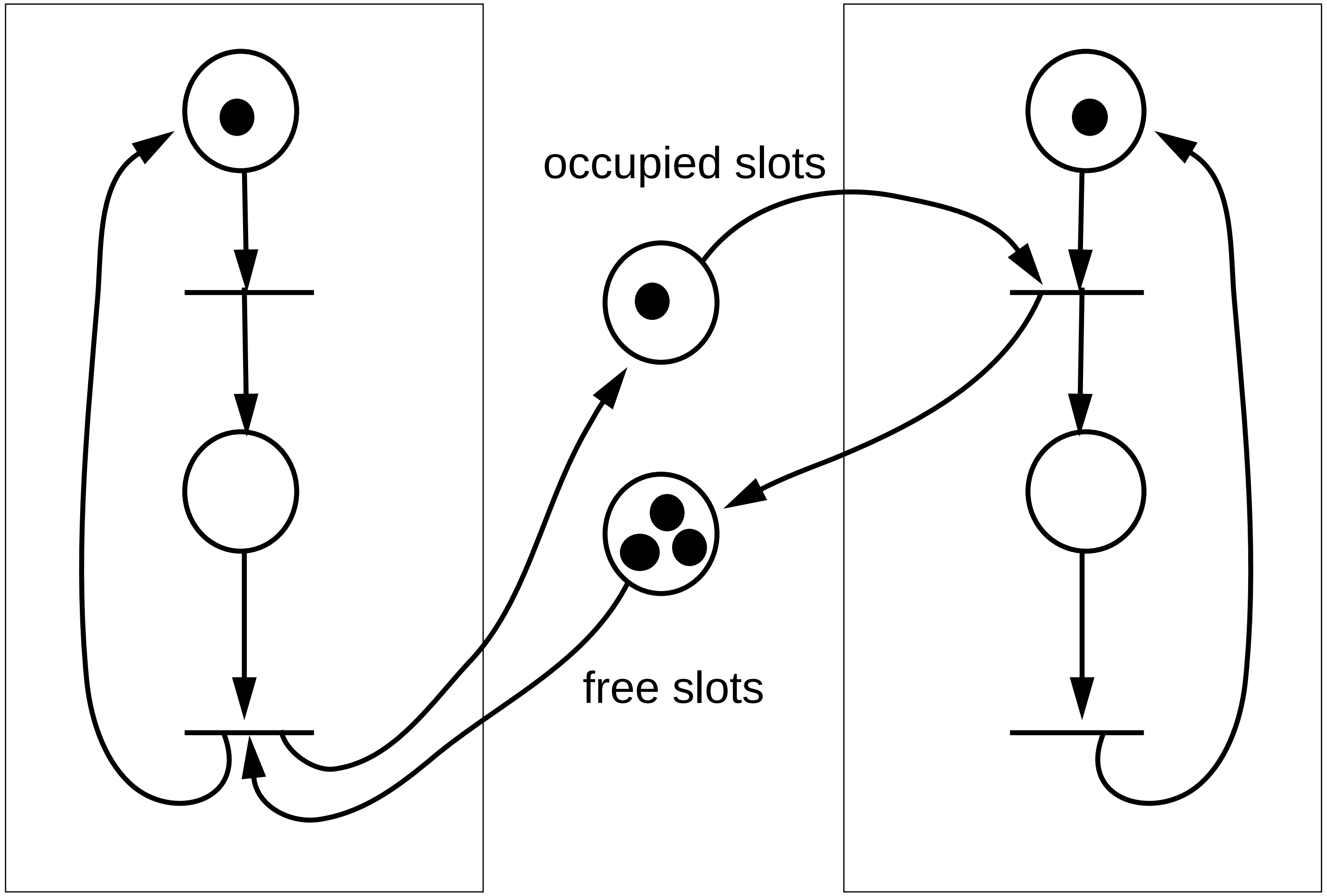
Fork and Join



Producers and Consumers



Bounded Buffers



Properties

Reachability:

- ❑ The reachability set R(C,μ) of a net C is the set of all markings μ′ reachable from initial marking μ.

Boundedness:

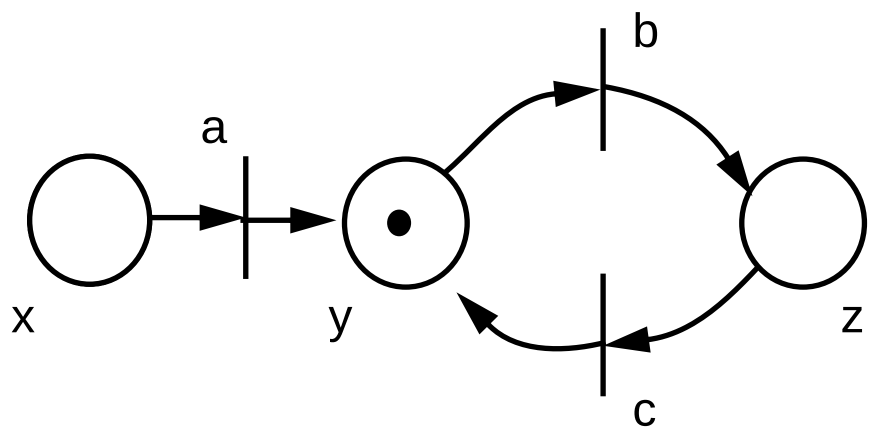
- ❑ A net C with initial marking μ is safe if places always hold at most 1 token.
- ❑ A marked net is (k-)bounded if places never hold more than k tokens.
- ❑ A marked net is conservative if the number of tokens is constant.

Liveness:

- ❑ A transition is deadlocked if it can never fire.
- ❑ A transition is live if it can never deadlock.

✎ Are the examples we have seen bounded? Are they live?

Liveness and Boundedness



This net is both safe and conservative.

Transition a is deadlocked.

Transitions b and c are both live.

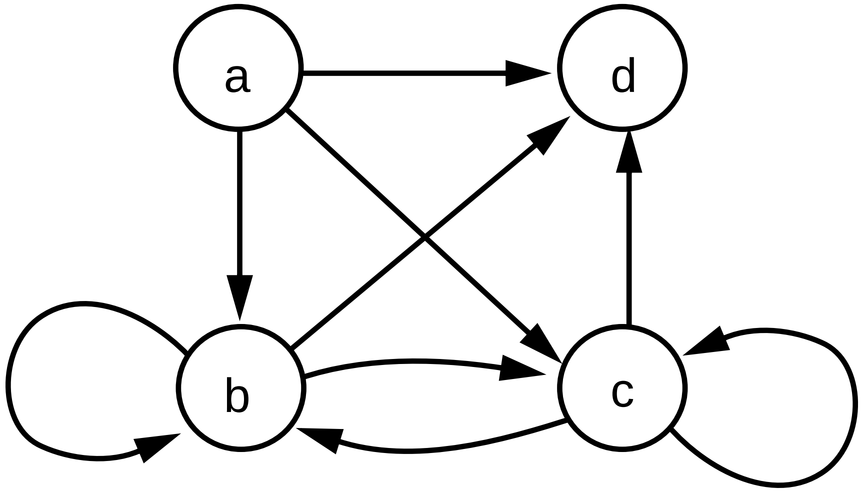
The reachability set is {{y}, {z}}.

Related Models

Finite State Automata

- ❑ Equivalent to regular expressions
- ❑ Can be modelled by one-token conservative nets
- ❑ Cannot model unbounded Petri nets

The FSA for: a(b|c)*d



Computational Power

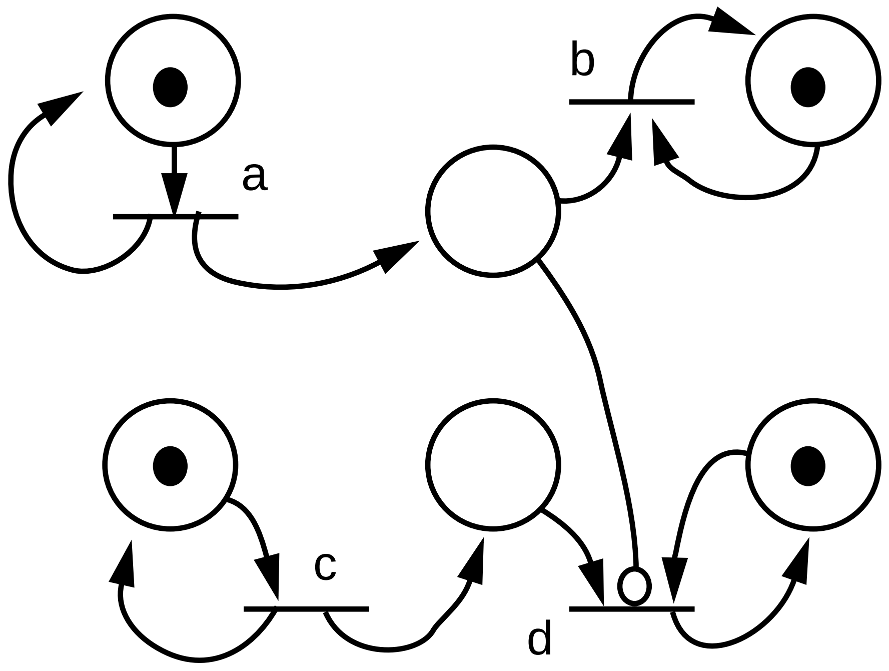
Petri nets are not computationally complete

- ❑ Cannot model “zero testing”
- ❑ Cannot model priorities

A zero-testing net:

An equal number of
a and b transitions may fire
as a sequence during any
sequence of matching
c and d transitions.

(#a ≥ #b, #c ≥ #d)



Applications of Petri nets

Modelling information systems:

- ❑ Workflow
- ❑ Hypertext (possible transitions)
- ❑ Dynamic aspects of OODB design

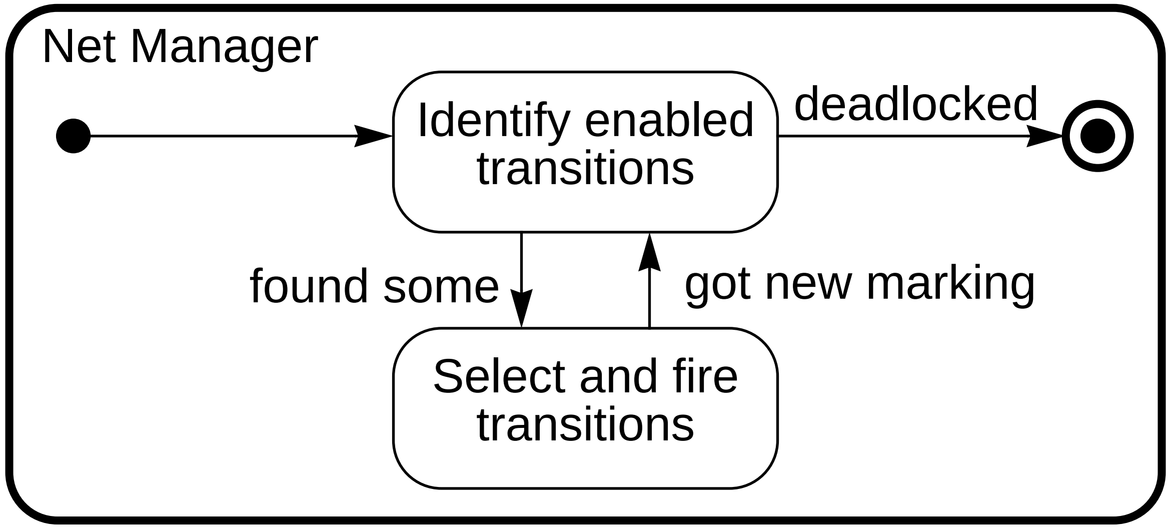
Implementing Petri nets

We can implement Petri net structures in either centralized or decentralized fashion:

- Centralized:
  - ☞ A single “net manager” monitors the current state of the net, and fires enabled transitions.
- Decentralized:
  - ☞ Transitions are processes, places are shared resources, and transitions compete to obtain tokens.

Centralized schemes

In one possible centralized scheme, the Manager selects and fires enabled transitions.

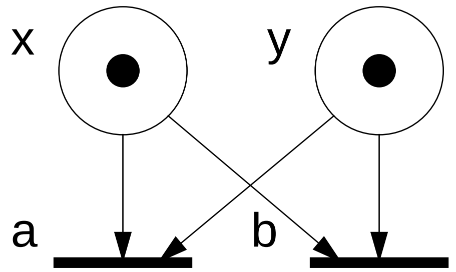


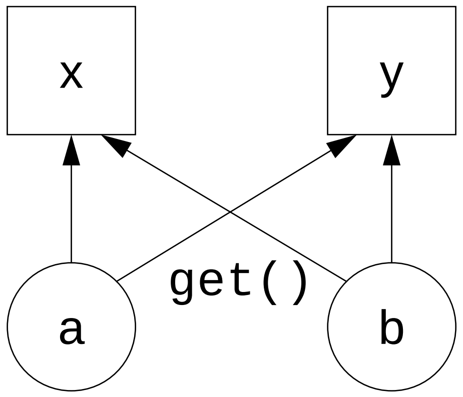
Concurrently enabled transitions can be fired in parallel.

✎ What liveness problems can this scheme lead to?

Decentralized schemes

In decentralized schemes transitions are processes and tokens are resources held by places:



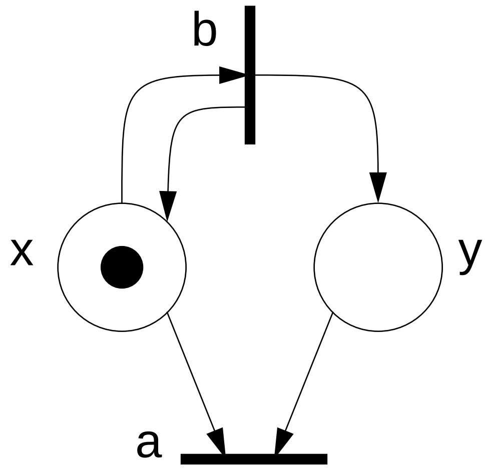


Transitions can be implemented as thread-per-message gateways so the same transition can be fired more than once if enough tokens are available.

Tokens must be grabbed in a consistent order, or the net can deadlock even though transitions are enabled!

Transactions

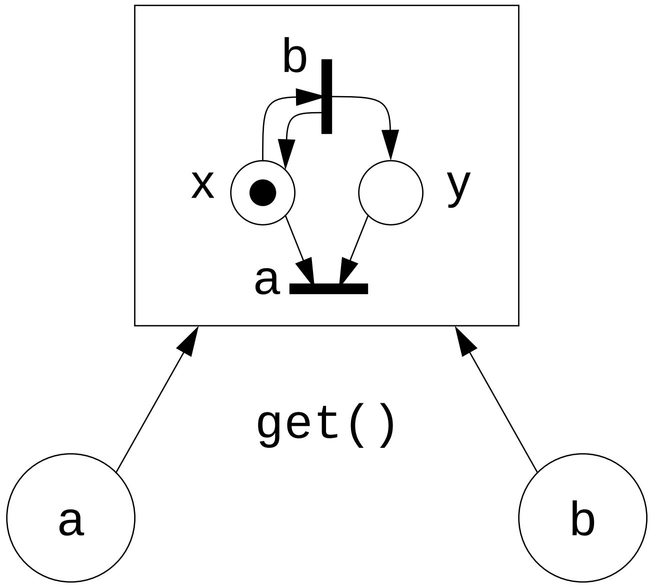
Transitions attempting to fire must grab their input tokens as an atomic transaction, or the net may deadlock even though there are enabled transitions:



If a and b are implemented by independent processes, and x and y by shared resources, this net can deadlock even though b is enabled if a (incorrectly) grabs x and waits for y.

Coordinated interaction

A simple solution is to treat the state of the entire net as a single, shared resource:



If a transition is not enabled, it waits and releases the net till it changes state again. When a transition fires and updates the net, it notifies all waiting transitions.

✎ How could you refine this scheme to work in a distributed setting?

Summary

You Should Know The Answers To These Questions:

- ❑ How are Petri nets formally specified?
- ❑ How can nets model concurrency and synchronization?
- ❑ What is the “reachability set” of a net? How can you compute this set?
- ❑ What kinds of Petri nets can be modelled by finite state automata?
- ❑ How can a (bad) implementation of a Petri net deadlock even though there are enabled transitions?
- ❑ If you implement a Petri net model, why is it a good idea to realize transitions as “gateways”?

Can You Answer The Following Questions?

- ✎ What are some simple conditions for guaranteeing that a net is bounded?
- ✎ How would you model the Dining Philosophers problem as a Petri net?
  Is such a net bounded? Is it conservative? Is it live?
- ✎ What could you add to Petri nets to make them Turing-complete?
- ✎ What constraints could you put on a Petri net to make it fair?

11. Scripting Agents

Overview

- ❑ What is Piccola
- ❑ Building and Using Coordination Abstractions
- ❑ Example I: Reader Writer
- ❑ Example II: Dining Philosophers
  - core Components
  - renaming Interfaces
  - wiring

An Overview of Piccola

Piccola is a small language for composition:

- ❑ only 4 keywords: **`def return root dynamic`**
- ❑ very few operators: **`( ) , = # \`**
- ❑ predefined services: `newChannel(), run(), ...`
- ❑ Access to Java objects

Concepts:

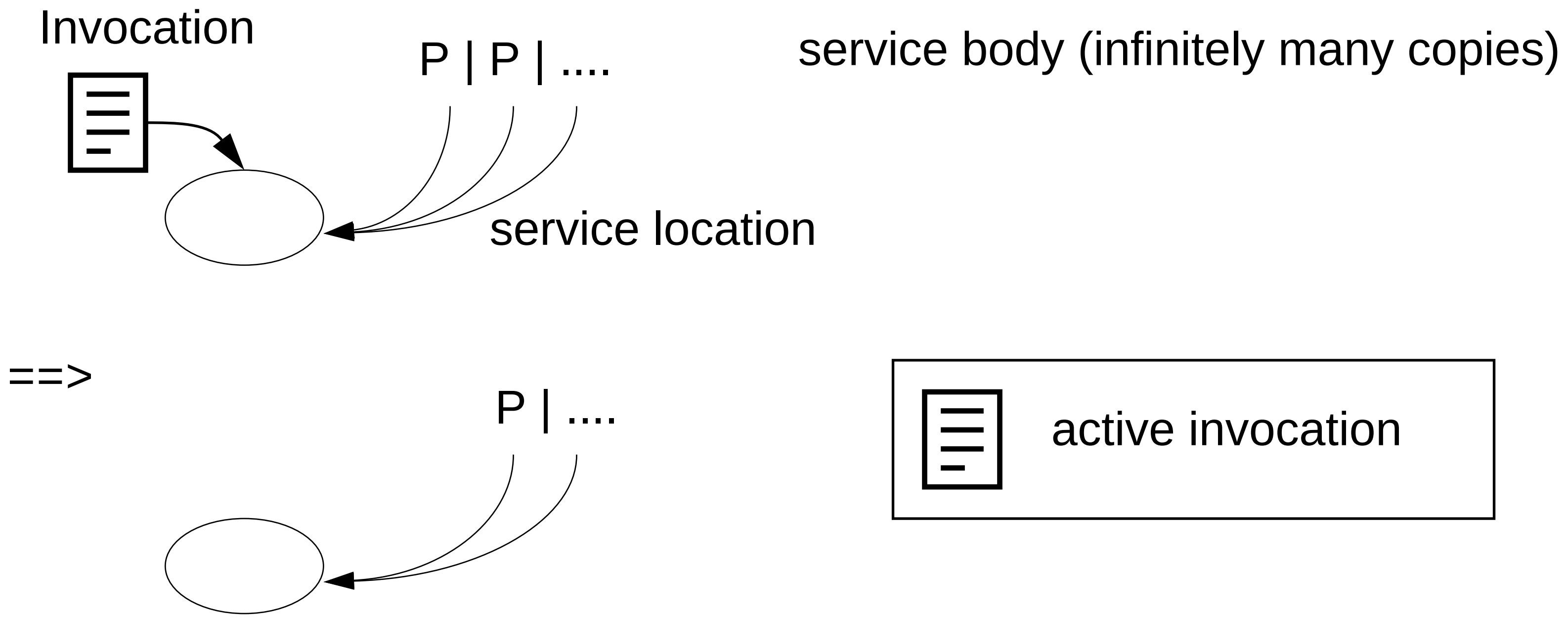
- ❑ Behaviour is represented by agents. An agent autonomously:
  - invokes services
  - composes forms
- ❑ State is represented by channels. Channels are the (only) location of communication for agents
- ❑ Structure is represented by forms. A form is a finite mapping of labels to values (other forms).

☞ services are represented by a channel and an associated agent.

Formal Concept

- the πL-calculus provides the semantics
- communication is the only notion for program progress

  `a ! F | a ? X > P ==> P[X/F]`
- a service:



The invocation form contains the context for the service: Arguments, result channel, exception channel etc.

Forms

Forms are finite mappings from label (=identifiers) to values.

- ❑ Projection: F.l
- ❑ Extension: (F, l = G)
- ❑ Polymorphic extension: (F, G)
- ❑ Restriction: (F \ l)

```
baseForm =
  Text = "foo"
  Name = Text
  Size = (x = 10, y = 20)     baseForm.Size.x ==> 10
coloredForm =
  baseForm
  Color = "green"             coloredForm.Name ==> “foo”
modForm =
  baseForm
  Size =
    baseForm.Size
    x = 15                    modForm.Size.y ==> 20
```

Forms (cont.)

Forms are ubiquitous in Piccola:

- ❑ Interfaces to Components
- ❑ Namespaces
- ❑ Keyword based arguments
- ❑ Modules and Packages
- ❑ Objects (immutable)
- ❑ Dictionaries (immutable)

as such, Piccola can manipulate forms as first class values

- ☞ explicit manipulation of interfaces
- ☞ explicit manipulation of argument lists
- ☞ explicit manipulation of environments

Intention

Scripting Components within one or more styles

- ❑ Components are external (foreign language, i.e. Java) or in Piccola.
- ❑ Scripting: high level, declarative operators to compose components
- ❑ Style:
  - – Defines kinds of valid compositions
  - – may ensure system properties

How are these requirements supported by Piccola?

- ❑ Uniform, general interface to components by Forms.
- ❑ User defined infix and prefix operators.
- ❑ Any (public) Java object can be scripted.

Communicating Agents

```
ch = newChannel()         # create a new Channel
run
  do:
    ch.send("Hello")      # sender agent
run
  do:
    v = ch.receive()      # receiver agent
    println(v)
```

- ☞ The whole script evaluates to a form with label `ch`
- ☞ `run` is a service that evaluates its `do:` block in a new agent and returns `()`
- ☞ `newChannel()` returns a form with services `send()` and a blocking `receive()`

Communicating Agents (cont.)

or we can use a style for Channels:

Components	C A	Channels Agents
Connectors	!, ?, ?*	output, input, replicated input
Rules	C ! Form ==> A C ? Abstraction ==> A C ?* Abstraction ==> A	send form along channel C receive form and run abstraction multiple receive

```
root = (root, load("pil"))
ch = newChannel()        # redefined in pil-style
ch ! "Hello"
ch ? \(v) = println(v)
```

✔ No run() invocations anymore

✔ Infix operators for sending and receiving

Coordination Abstractions

Channels are the only primitive means to coordinate agents.

For example: run two agents in parallel and return whatever result the first one delivers. Assuming the other gets blocked.

```
OrJoin(X):
   receptor = newBlackboard()
   run (do: receptor.write(X.left()))
   run (do: receptor.write(X.right()))
   return receptor.remove()

stop() =
   newChannel().receive()     # will never remove anything
```

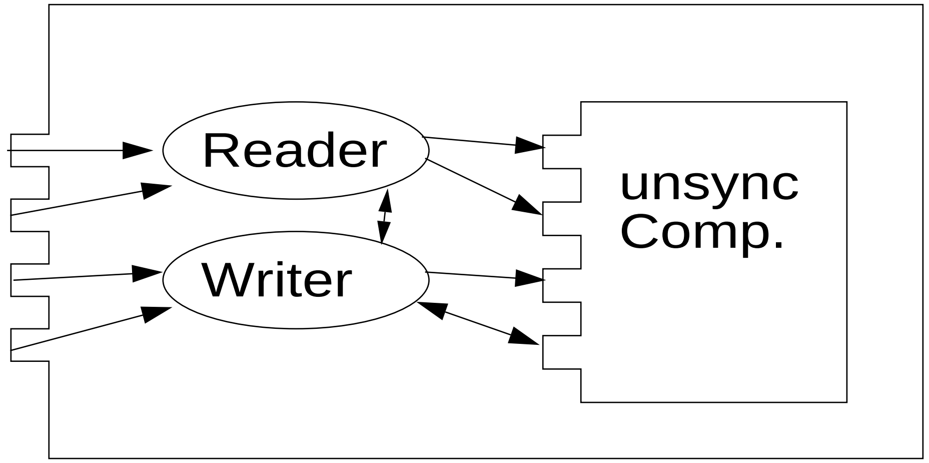
A semaphore is a channel:

```
newSemaphore():
   ch = newChannel()
   ch.send()                  # initially not locked
   return
      lock = ch.receive
      unlock = ch.send
```

Reader Writer

a Component with a set of intended Reader and Writer methods.

Need a generic wrapper:



```
Need core RW Policies: Safe, Fair, Writer Priority ... with
newRWPolicy(): ...
   return
      preReader: ...
      postReader: ...
      preWriter: ...
      postWriter: ...
```

Wiring

Generic Wrapper:

```
# X.reader = form of reader methods
# X.writer = form of writer methods
# X.policy = reader writer policy
# return wrap(X.reader), wrap(X.writer)
wireRWPolicy(X) =
   wrapAllLambda
      form = X.reader
      map(service)(Args) =
         X.policy.preReader()
         service(Args)
         X.policy.postReader()
   wrapAllLambda
      form = X.writer
      map(service)(Args) =
         X.policy.preWriter()
         service(Args)
         X.policy.postWriter()
```

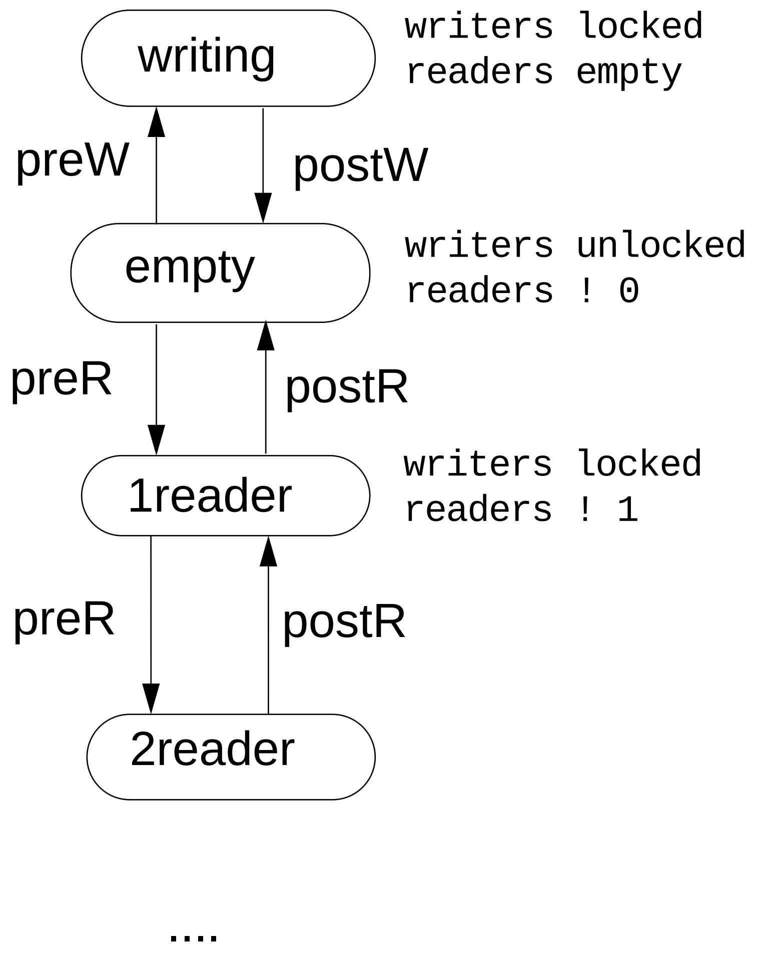
Usage: given a Form `f` with `r1, r2` Readers and `w1` writer method. Then:

```
wrappedF = wireRWPolicy
   policy = newRWPolicy()
   reader =
      r1 = f.r1
      r2 = f.r2
   writer = (w1 = f.w1)
```

Then: `wrappedF` has methods `r1, r2`, and `w1` and guarantees the RW Policy for `f`.

Safe RW Policy

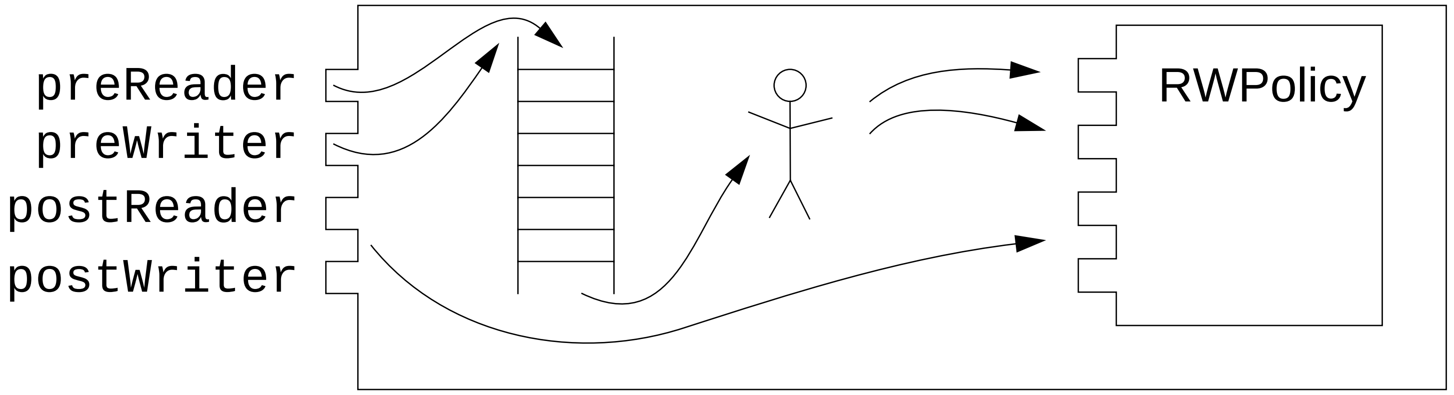
```
newRWPolicy():
   writers = newSemaphore()
   readers = newBlackboard()
   readers.write(0)

   return
      preReader:
         r = readers.remove()
         if (r == 0) (then: writers.lock())
         readers.write(r + 1)

      postReader:
         r = readers.remove()
         if (r == 1) (then: writers.unlock())
         readers.write(r - 1)

      preWriter = writers.lock
      postWriter = writers.unlock
```



Safe + Queue = Fair

The above version is safe, but writers may starve...

- ☞ serve `preWriter` and `preReader` using a FIFO policy
- ☞ New policy blocks `preReader`, when a Writer is waiting and vice versa
- ☞ Use a passive queue (with `put()`, and `get()`) and a single worker:



- ☞ Need a generic abstraction to queue services:

Queue

```
queuedService(Form) =
   q = newQueue()
   def worker() =
      q.get().do()                                  # execute this Job
      worker()                                      # wait for next Job in the queue
   run(do: worker())                                # start worker agent

   return wrapAllLambda
      form = Form
      map(service)(Args) =
         result = newBlackboard()
         q.put(do: result.write(service(Args)))     # send Job to worker
         return(result.remove())                    # return (blocking) result
```

Adapting the policy:

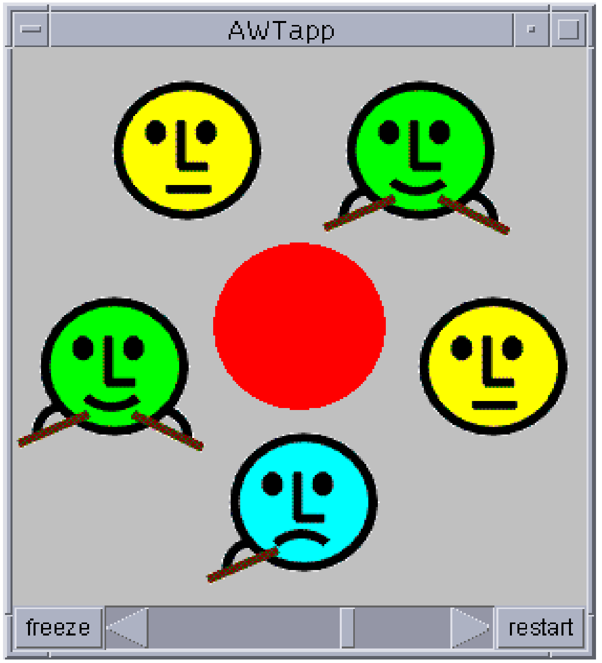
```
myPolicy = rwPolicy.newRWPolicy()

myFifoPolicy =
   myPolicy
   queuedService
      preWriter = myPolicy.preWriter
      preReader = myPolicy.preReader
```

Dining Philosophers

The well known example:

Components:

- 5 Forks as Semaphores
- Group of forks to freeze/initialize
- Policy
- 5 Philosophers
- Group of philosophers to initialize/start
- View
- main script to initialize and connect these components.

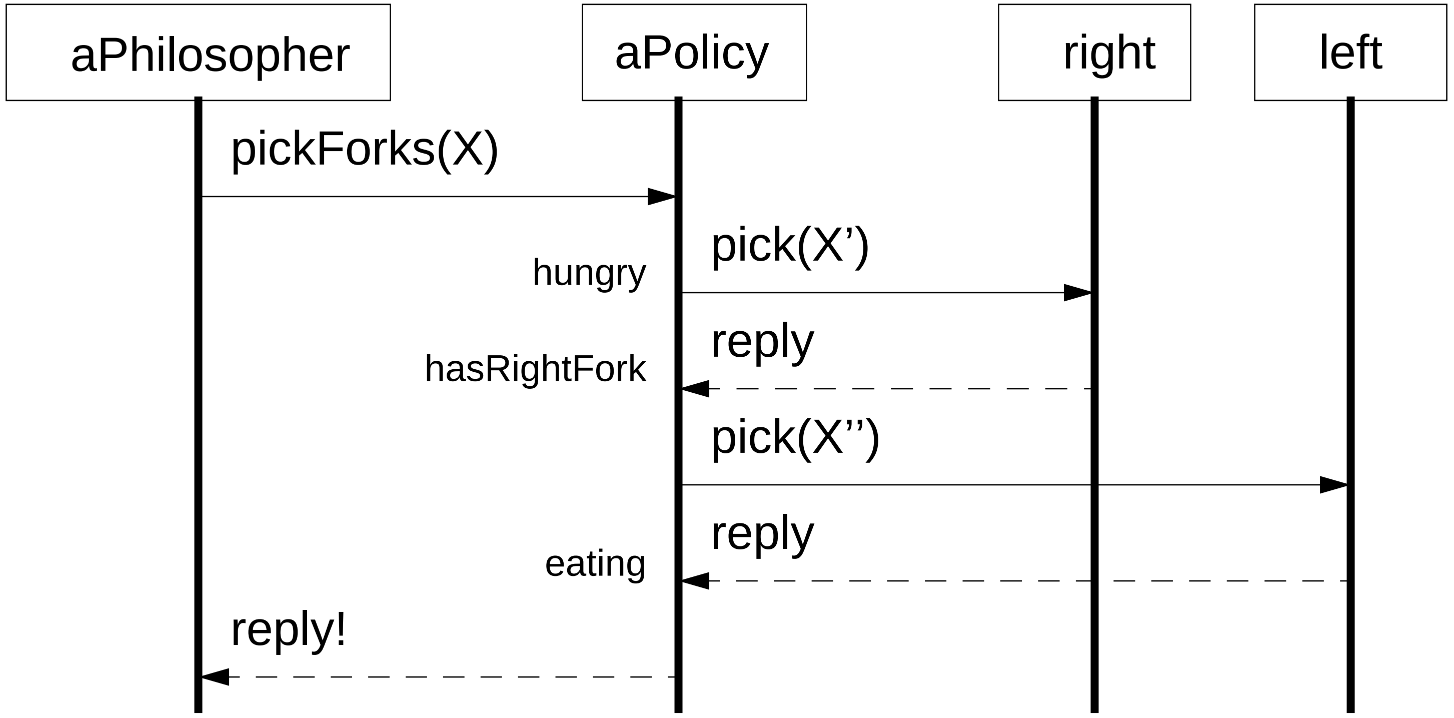


A Philosopher

```
newPhilosopher(X) =
   running = ...    # Flag is true while philo. active
   def agent():
      running.raiseWhenFalse()
      sleep(X.thinkTime())
      X.policy.pickForks(X)
      sleep(X.eatTime())
      X.policy.dropForks(X)
      agent()

   start() =
      running.setTrue()
      run(do:
         try
            do: agent()
            catch(E): println("Phil got: " + E))

   stop() = running.setFalse()
```

- this is a minimal Philosopher: no Identifier ...
- Picking Forks is delegated to policy
- Active philosopher is represented by agent that runs an endless loop

The Philosopher’s lifetime



- ❑ use pre- and post-hook of the Forks to notify display
- ❑ use Argument Forms to pass information: `PhilId, ForkId, has, fork`

Group of Philosophers

```
philosGroup = newAssembling                  # multiplex start, stop.
   start: ()
   stop: ()

def init(N) =
   if (N < 5)
      then:
         philos = newPhilosopher
            thinkTime: ...
            eatTime: ...
            PhilId = N
            left = ...         the left Fork
            right = ....       the right Fork
            policy = ...       the policy

         philosGroup.extend(philos)
            # add this philosopher to group
         init(N + 1)

init(0)

philosGroup.group().start()
```

Wrap forks such that `pick()` and `drop()` include default context information (`PhilId, ForkId`). E.g.:

```
f = table.getFork(leftForkId)
left =
   pick(X): f.pick
      PhilId = PhilId
      ForkId = leftForkId
      X
   drop(X): f.drop
      PhilId = PhilId
      ForkId = leftForkId
      X
```

Policies

all Right handed:

```
policy =
   pickForks(X) =
      X.right.pick(has = 0, fork = "RIGHT")
      X.left.pick(has = 1, fork = "LEFT")
   dropForks(X) =
      X.left.drop(has = 2, fork = "LEFT")
      X.right.drop(has = 1, fork = "RIGHT")
```

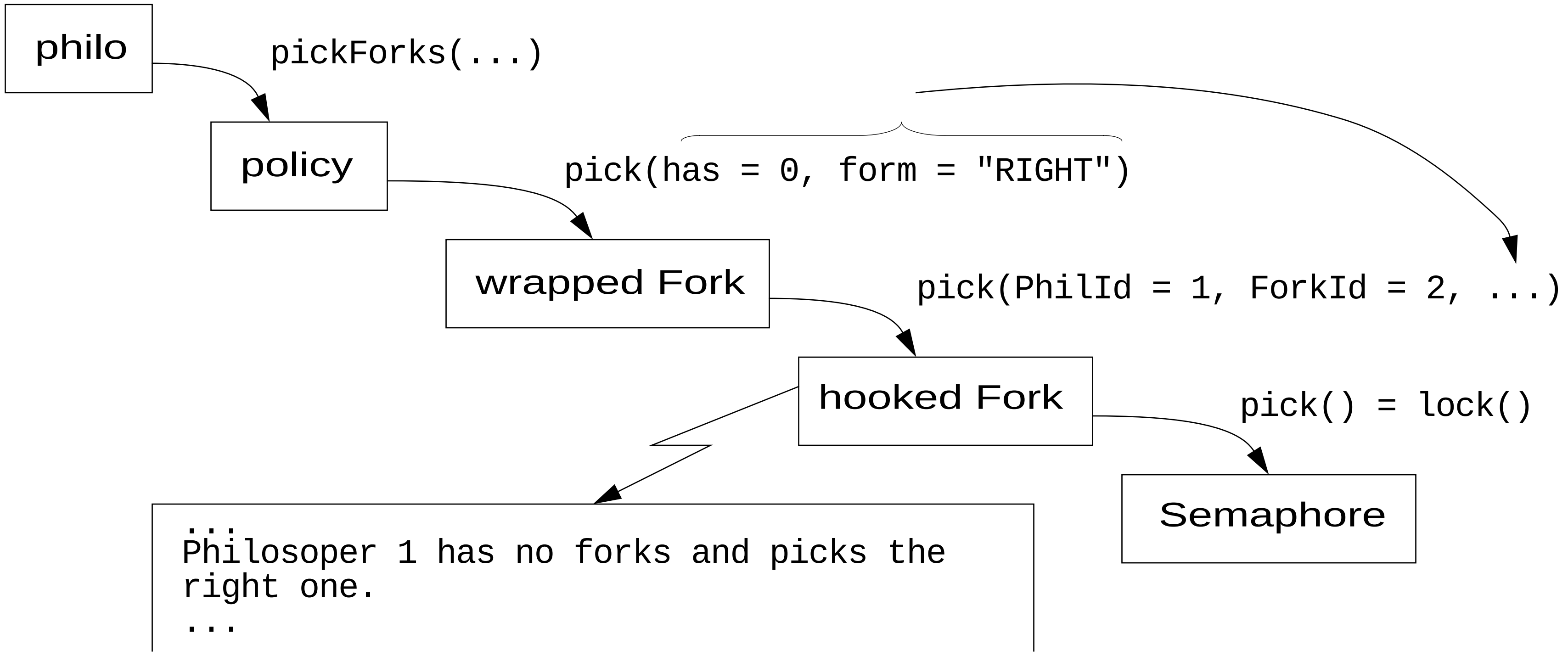
Deadlock free:

```
policy =
   pickForks(X) =
      if (X.PhilId == 0)               # avoid cycles
         then:
            X.right.pick(has = 0, fork = "RIGHT")
            X.left.pick(has = 1, fork = "LEFT")
         else:
            X.left.pick(has = 0, fork = "LEFT")
            X.right.pick(has = 1, fork = "RIGHT")

   dropForks(X) = ...
```

First Class Arguments

Observe how contextual information is available a fork gets picked and dropped:

- This information is needed neither by the forks nor by the philosophers
- It is needed to hook in notifications for the GUI, log File etc.



```
...
Philosoper 1 has no forks and picks the
right one.
...
```

Wiring Philosophers, Forks, and the display

Provide a factory service to create Forks and add hooks to pre- and post methods:

```
newFork() =
   s = newSemaphore()
   return wrapServices
      form = (pick = s.lock, drop = s.unlock)
      wrap =
         pick =
            pre(X):
               CoutView.prePickFork(X)          # to log console
               view.view.prePickFork(X)         # notify GUI
            post(X):
               CoutView.postPickFork(X)
               view.view.postPickFork(X)
               if (X.has == 0)
                  then: sleep(2500)
         drop =
            pre(X):
               CoutView.preDropFork(X)
               view.view.preDropFork(X)
            post(X):
               CoutView.postDropFork(X)
               view.view.postDropFork(X)
```

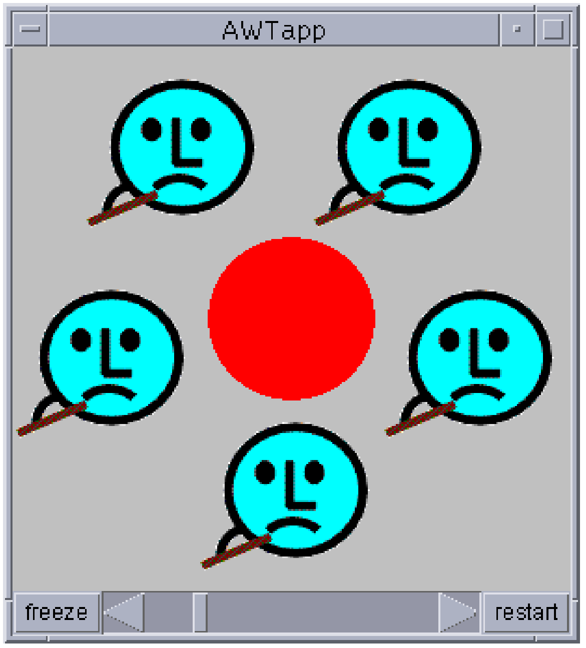
A Style for GUI Events

Components	C E R L	GUI-Component Event type Response Listener
Connectors	( ), ?	
Rules	E(R) ==> L C ? L ==> ()	compose an event type with a response connect a component to a listener

```
...
restart() =
   freeze()
   view.reset()
   philosGroup.group().start()

restartButton = awtComponent("java.awt.Button").set(Label = "restart")

restartButton ? Action(do: restart())
```

Graphical Layout

```
f = newFrame
   newBorderPanel
      center = view.img
      south = newBorderPanel
         center = slider
         west = freezeButton
         east = restartButton

view.draw()

f.show()
```



hungry !!!

Piccola Projects

- Visualization (Debugging)
- Reasoning
- Distribution (using Corba, RMI, DCOM...)
- Composition Workbench (including Repositories)
- Implementation and optimization of forms.

Communicating and Mobile Systems - the π-calculus

Markus Lumpe

Institute of Computer Science and Applied Mathematics (IAM)

University of Berne

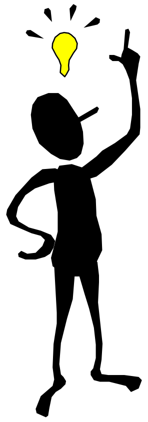
Neubrückstrasse 10, CH-3012 Bern

E-mail: *lumpe@iam.unibe.ch*

WWW: *http://www.iam.unibe.ch/~lumpe*

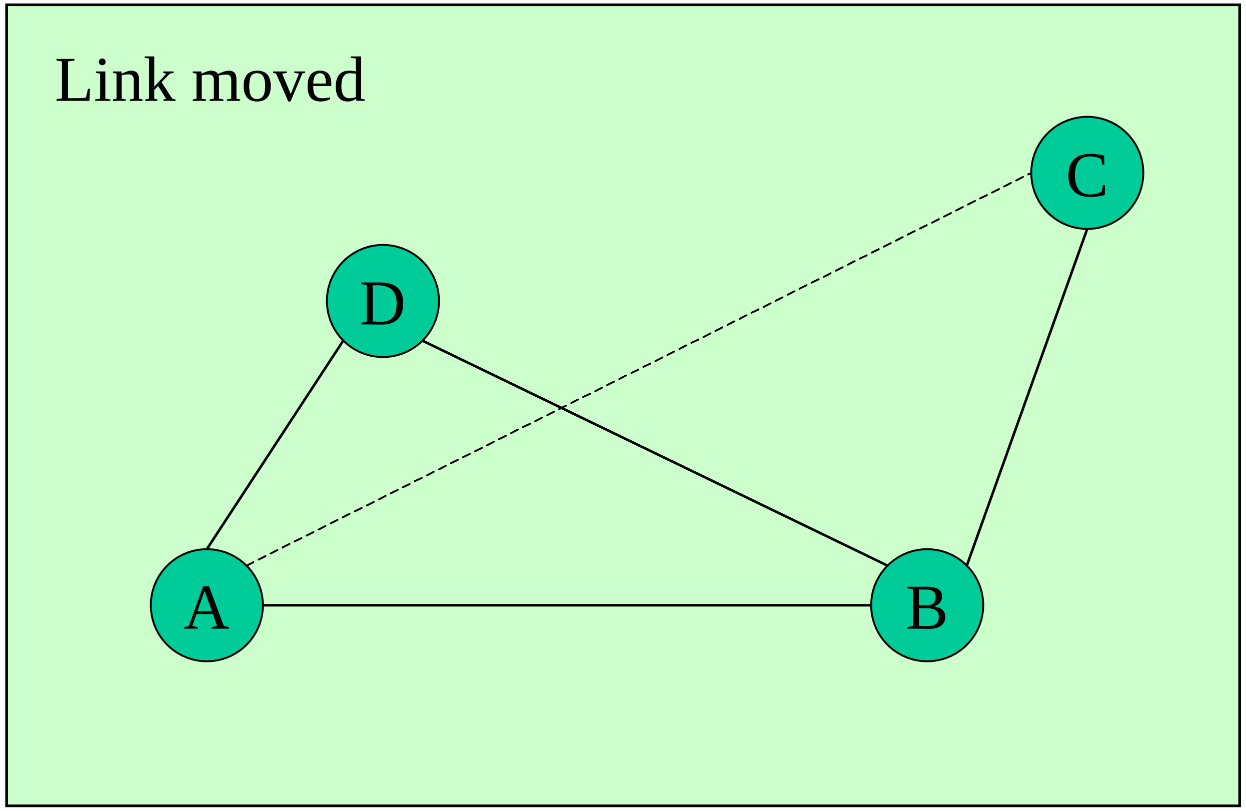
The programming model

- Communication is a fundamental and integral part of computing, whether between different computers on a network, or between components within a single computer.
- Robin Milner’s view: Programs are built from communicating parts, rather than adding communication as an extra level of activity.



Programs proceed by means of communication.

Evolving Automata



Automata

Starting point: The components of a system are interacting automata.

An automaton is a quintuple $A = (\Sigma, Q, q_0, \sigma, F)$ with:

- a set Σ of *actions* (sometimes called an alphabet),
- a set $Q = \{ q_0, q_1, \ldots \}$ of *states*,
- a subset F of Q called the *accepting states*,
- a subset σ of $Q \times A \times Q$ called the *transitions*,
- a designated start state q_0.

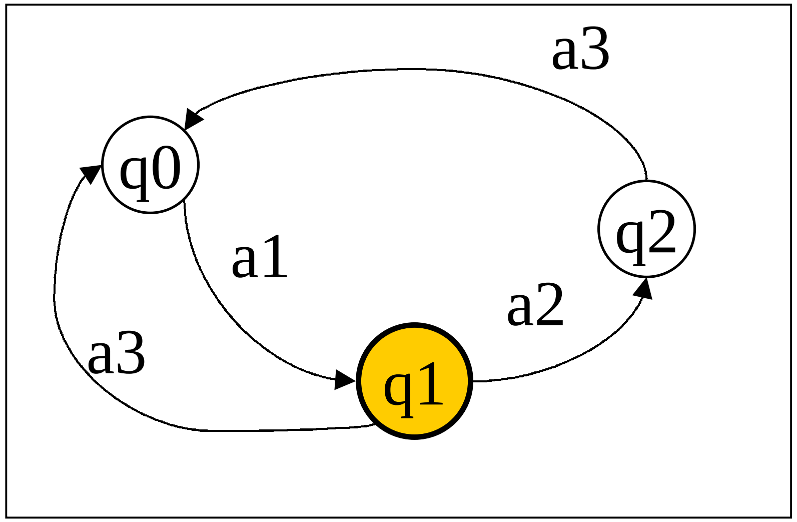
A transition $(q, a, q') \in \sigma$ is usually written $q \xrightarrow{a} q'$.

The automaton A is said to be finite if Q is finite.

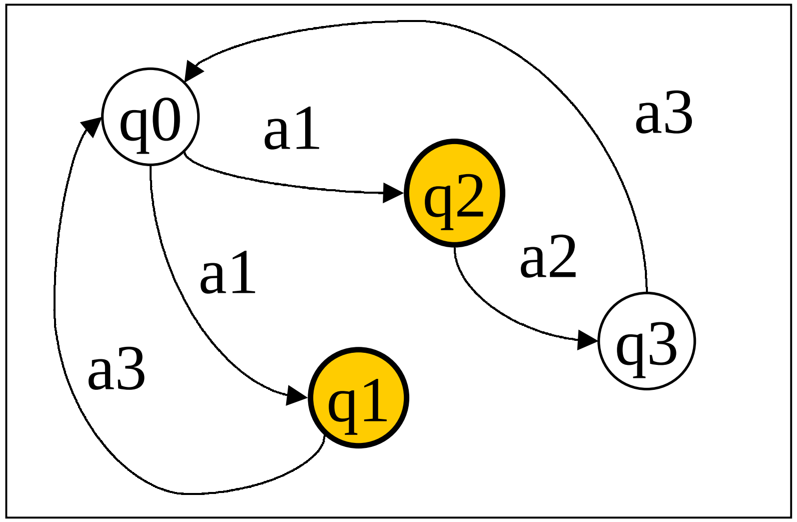
Behaviour of Automata

An automaton is deterministic if for each pair $(q, a) \in Q \times \Sigma$ there is exactly one transition $q \xrightarrow{a} q'$.

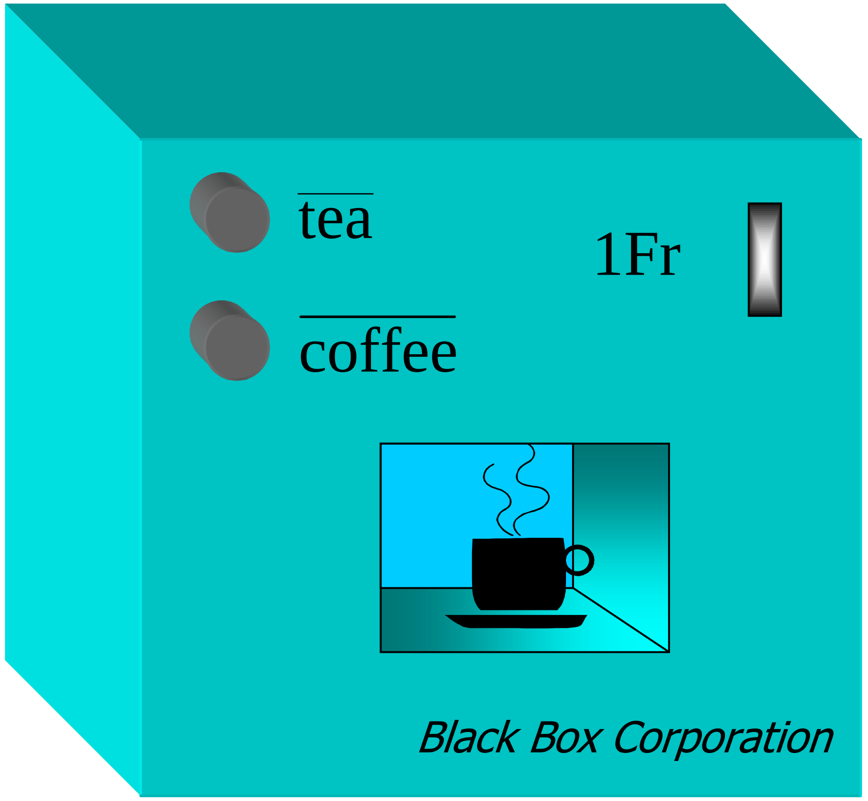
deterministic automata:



non-deterministic automata:



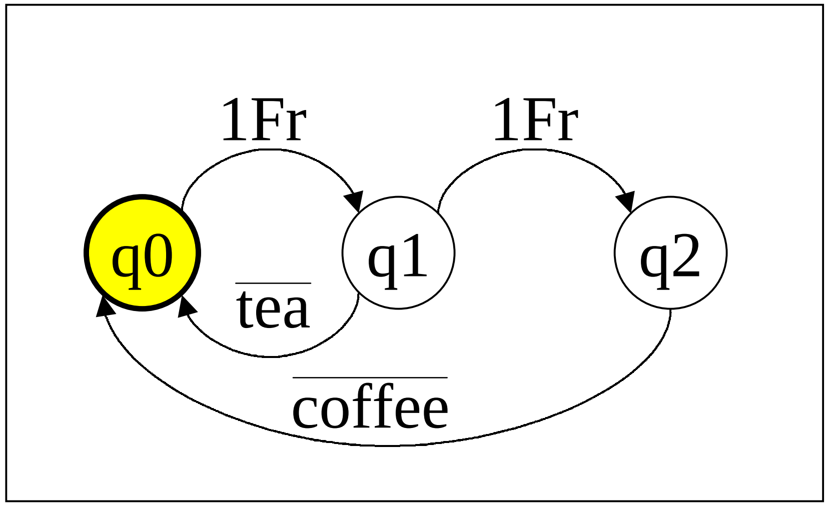
Vending machine



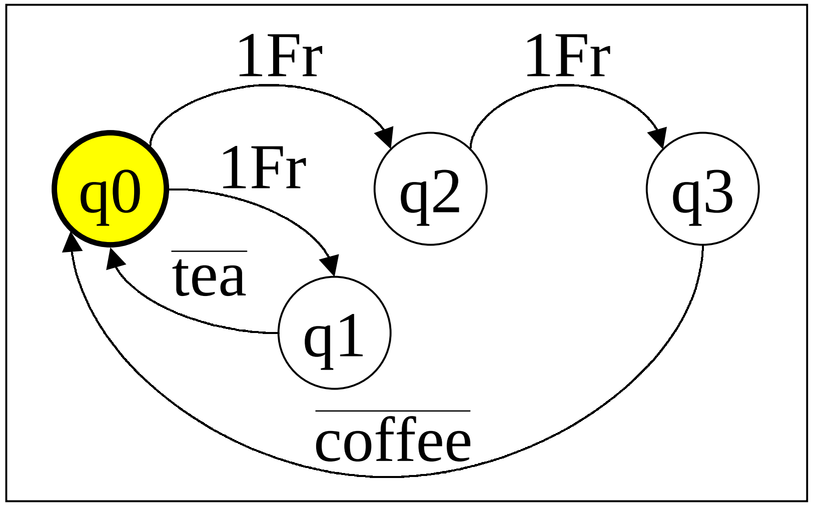
A tea/coffee vending machine is implemented as black box with a three-symbol alphabet $\{1\text{Fr}, \overline{\text{tea}}, \overline{\text{coffee}}\}$.

Internal transition diagrams

Deterministic system S1:



Non-deterministic system S2:



Are both systems equivalent?

S1 = S2?

S1:

$$q0 = 1Fr \cdot q1 + \varepsilon$$
$$q1 = \overline{tea} \cdot q0 + 1Fr \cdot q2$$
$$q2 = \overline{coffee} \cdot q0$$

$$q1 = \overline{tea} \cdot q0 + 1Fr \cdot \overline{coffee} \cdot q0$$
$$q0 = 1Fr \cdot (\overline{tea} \cdot q0 + 1Fr \cdot \overline{coffee} \cdot q0) + \varepsilon$$
$$q0 = 1Fr \cdot (\overline{tea} + 1Fr \cdot \overline{coffee}) \cdot q0 + \varepsilon$$
$$q0 = (1Fr \cdot (\overline{tea} + 1Fr \cdot \overline{coffee}))^*$$

S2:

$$q0 = 1Fr \cdot q1 + 1Fr \cdot q2 + \varepsilon$$
$$q1 = \overline{tea} \cdot q0$$
$$q2 = 1Fr \cdot q3$$
$$q3 = \overline{coffee} \cdot q0$$

$$q2 = 1Fr \cdot \overline{coffee} \cdot q0$$
$$q0 = 1Fr \cdot \overline{tea} \cdot q0 + 1Fr \cdot 1Fr \cdot \overline{coffee} \cdot q0 + \varepsilon$$
$$q0 = 1Fr \cdot (\overline{tea} \cdot q0 + 1Fr \cdot \overline{coffee} \cdot q0) + \varepsilon$$
$$q0 = 1Fr \cdot (\overline{tea} + 1Fr \cdot \overline{coffee}) \cdot q0 + \varepsilon$$
$$q0 = (1Fr \cdot (\overline{tea} + 1Fr \cdot \overline{coffee}))^*$$

The systems S1 and S2 are language-equivalent, but the *observable behaviour* is not the same.

Automata - Summary

Language-equivalence is not suitable for all purposes. If we are interested in interactive behaviour, then a non-deterministic automaton cannot correctly be equated behaviourally with a deterministic one.

A different theory is required!

Labelled transition systems

A labelled transition system over actions *Act* is a pair (*Q*, *T*) consisting of:

• a set $Q = \{ q_0, q_1, \ldots \}$ of *states*,

• a ternary relation $T \subseteq (Q \times Act \times Q)$, known as a *transition relation*.

If $(q, \alpha, q') \in T$ we write $q \xrightarrow{\alpha} q'$, and we call *q* the source and *q*' the target of the transition.

States and Actions

Important conceptual changes:

- What matters about a string *s* - a sequence of actions - is not whether it drives the automaton into an accepting state (since we cannot detect this by interaction) but whether the automaton is able to perform the sequence of *s* interactively.
- A labelled transition system can be thought of as an automaton without a start or accepting states.
- *Any* state can be considered as the start.

Actions consist of a set L of *labels* and a set $\overline{L}$ of *co-labels with* $\overline{L} = \{\overline{a} | a \in L\}$. We use α, β, … to range over actions *Act*.

Strong Simulation - Idea

- In 1981 D. Park proposed a new approach to define the equivalence of automatons - bisimulation.
- Given a labelled transition system there exists a standard definition of bisimulation equivalence that can be applied to this labelled transition system.
- The definition of bisimulation is given in a *coinductive* style that is, two systems are bisimular if we cannot show that they are not.
- Informally, to say a ‘system S1 simulates system S2’ means that S1’s observable behaviour is at least as rich as that of S2.

Strong Simulation - Definition

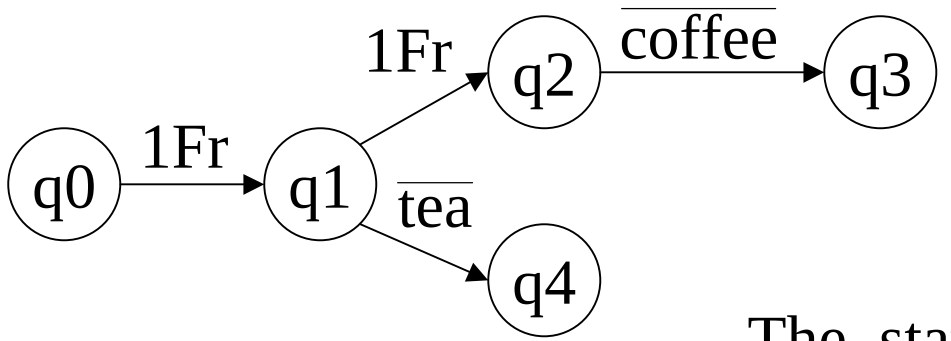
Let (*Q*, *T*) be an labelled transition system, and let ***S*** be a binary relation over *Q*. Then S is called a strong simulation over (*Q*, *T*) if, whenever p***S***q,

if $p \xrightarrow{\alpha} p'$ then there exists *q*' $\in$ *Q* such that $q \xrightarrow{\alpha} q'$ and *p*'***S****q*'.

We say that *q* strongly simulates *p* if there exists a strong simulation ***S*** such that *p****S****q*.

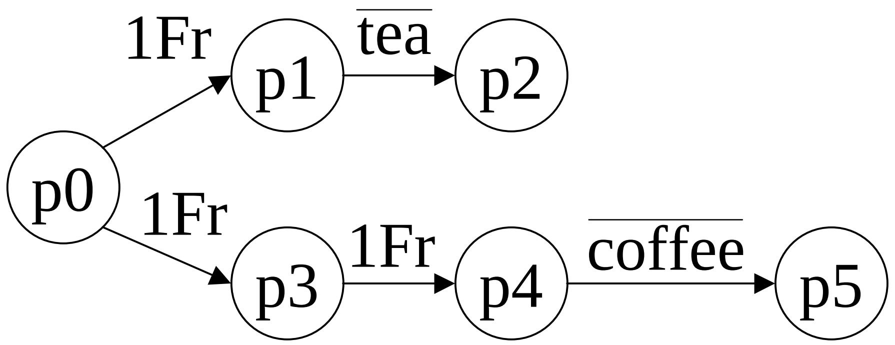
Strong Simulation - Example

S1:



The states q0 and p0 are different. Therefore, the systems S1 and S2 are not considered to be equivalent.

S2:



Strong Simulation - Example II

Define ***S*** by

S = {(p0, q0), (p1, q1), (p3, q1), (p2, q4), (p4, q2), (p5, q3)}

then ***S*** is a strong simulation; hence q0 strongly simulates p0. To verify this, for every pair (*p*, *q*) ∈ ***S*** we have to consider each transition of p, and show that it is properly matched by some transition of q.

However, there exists no strong simulation ***R*** that contains the pair (q1, p1), because one of q1's transition could never be matched by p1. Therefore, the states q0 and p0 are different, and the systems S1 and S2 are not considered to be equivalent.

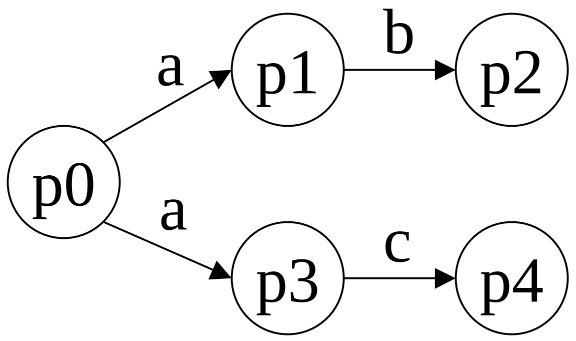
Strong Bisimulation

The converse $\boldsymbol{R}^{-1}$ of any binary relation $\boldsymbol{R}$ is the set of pairs (y, x) such that (x, y) $\in$ $\boldsymbol{R}$.

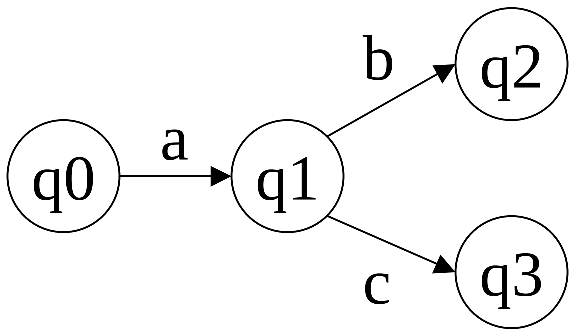
Let (Q, T) be an labelled transition system, and let $\boldsymbol{S}$ be a binary relation over Q. Then $\boldsymbol{S}$ is called a strong bisimulation over (Q, T) if both $\boldsymbol{S}$ and its converse $\boldsymbol{S}^{-1}$ are strong simulations. We say that p and q are strongly bisimular or strongly equivalent, written $p \sim q$, if there exists a strong bisimulation S such that $p\boldsymbol{S}q$.

Checking Bisimulation

S1:



S2:



S1 ~ S2?

To construct ***S*** start with (p0, q0) and check whether S2 can match all transitions of S1:

S = { (p0, q0), (p1, q1), (p3, q1), (p2, q2), (p4, q3) }

System S2 can simulate system S1. Now check, whether ***S***$^{-1}$ is a simulation or not:

S $^{-1}$ = { (q0, p0), (q1, p1), (q1, p3), (q2, p2), (q3, p4) }

Start with (q0, p0) $\in$ ***S*** $^{-1}$.

1: q0 has one transition ‘a’ that can be matched by two transitions of S1 (target p1 and p3, respectively) and we have (q1, p1) $\in$ ***S*** $^{-1}$ and (q1, p3) $\in$ ***S*** $^{-1}$.

2: q1 has two transitions ‘b’ and ‘c’, which, however, cannot appropriately be matched by the related states p1 and p3 of system S1 (p1 has only a ‘b’ transition whilst p3 has only a ‘c’ transition).
We have, therefore, S1 $\not\sim$ S2.

Some Facts on Bisimulations

~ is an equivalence relation.

If $\boldsymbol{S}_i$, i = 1, 2,... is a family of strong bisimulations, then the following relations are also strong bisimulations:

- $Id_{\boldsymbol{P}}$
- $\boldsymbol{S}_1 \circ \boldsymbol{S}_2 = \{(P, Q) \in P \times P$ if R exists with $(P, R) \in \boldsymbol{S}_1, (R, Q) \in \boldsymbol{S}_2 \}$
- $\boldsymbol{S}_i^{-1}$
- $\bigcup_{i \in I} \boldsymbol{S}_i$

Some Facts on Bisimulations II

$\boldsymbol{S}_1 \circ \boldsymbol{S}_2 = \{(P, Q) \in P \times P$ if R exists with $(P, R) \in \boldsymbol{S}_1$, $(R, Q) \in \boldsymbol{S}_2 \}$

Proof:

Let $(P, Q) \in \boldsymbol{S}_1 \circ \boldsymbol{S}_2$. Then there exists a R with $(P, R) \in \boldsymbol{S}_1$ *and* $(R, Q) \in \boldsymbol{S}_2$.

($\rightarrow$) If $P \xrightarrow{\alpha} P'$, then since $(P, R) \in \boldsymbol{S}_1$ there exists R' and $R \xrightarrow{\alpha} R'$ and $(P', R') \in \boldsymbol{S}_1$. Furthermore, since $(R, Q) \in \boldsymbol{S}_2$ there exists a Q' with $Q \xrightarrow{\alpha} Q'$ and $(R', Q') \in \boldsymbol{S}_2$. Due to the definition of $\boldsymbol{S}_1 \circ \boldsymbol{S}_2$ it holds that $(P', Q') \in \boldsymbol{S}_1 \circ \boldsymbol{S}_2$ as required.

($\leftarrow$) similar to ($\rightarrow$).

Bisimulation - Summary

Bisimulation is an equivalence relation defined over a labelled transition system which respects non-determinism. The bisimulation technique can therefore be used to compare the observable behaviour of interacting systems.

Note: Strong bisimulation does not cover unobservable behaviour which is present in systems that have operators to define reaction (i.e., internal actions).

The π-Calculus

- The π-calculus is a model of concurrent computation based upon the notion of *naming*.
- The π-calculus is a calculus in which the topology of communication can evolve dynamically during evaluation.
- In the π-calculus communication links are identified by *names*, and computation is represented purely as the communication of names across links.
- The π-calculus is an extension of the process algebra CCS, following the work by Engberg and Nielsen who added mobility to CCS while preserving its algebraic properties.
- The most popular versions of the π-calculus are the monadic π-calculus, the polyadic π-calculus, and the simplified polyadic π-calculus.

The π-Calculus - Basic Ideas

• The most primitive in the π-calculus is a *name*, Names, infinitely many, are x, y,… $\in$ ***N***; they have no structure.
• In the π-calculus we only have one other kind of entity: a *process*. We use P, Q, … to range over processes.

Polyadic prefixes:

• input prefix: $x(\tilde{y})$

“input some names $y_1,\ldots,y_n$ along the link named x”

• output prefix: $\overline{x}\langle\tilde{y}\rangle$

“output the names $y_1,\ldots,y_n$ along the link named x”

The π-Calculus - Syntax

Note: We only consider the simplified polyadic version.

P,Q	$::=$	$P \mid P$	Parallel composition
		$(\upsilon\, x)\, P$	Restriction
		$x(y_1,...,y_n).P$	Input
		$\overline{x}\langle y_1,...,y_n\rangle$	Output
		$!P$	Replication (input-only)
		$\mathbf{0}$	Null

Reduction Semantics

Milner proposed first a reduction semantics technique. Using the reduction semantics technique allows us to separate the laws which govern the neighbourhood relation among processes from the rules that specify their interaction.

$$
\begin{aligned}
&!P \equiv P \mid !P \\
&P \equiv P \mid \mathbf{0} \\
&P \mid Q \equiv Q \mid P \\
&(P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
&(\upsilon\, x)P \mid Q \equiv (\upsilon\, x)(P \mid Q),\ x \notin \mathrm{fn}(Q)
\end{aligned}
$$

$$
\frac{Q \longrightarrow R}{Q \mid P \longrightarrow R \mid P} \qquad \frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q \equiv Q'}{P \longrightarrow Q} \qquad \frac{P \longrightarrow Q}{(\upsilon\, x)P \longrightarrow (\upsilon\, x)Q}
$$

$$
x(y_1,\ldots,y_n).P \mid \overline{x}\langle z_1,\ldots,z_n\rangle \longrightarrow P\{y_1,\ldots,y_n \backslash z_1,\ldots,z_n\}
$$

Evolution

$\overline{x}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$ *can evolve to* $\overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle$ *or* $\overline{x}\langle y\rangle \mid \overline{z}\langle v\rangle$

$(\upsilon\, x)(\overline{x}\langle y\rangle \mid x(u).\overline{u}\langle v\rangle) \mid \overline{x}\langle z\rangle$ *can evolve to* $\overline{y}\langle v\rangle \mid \overline{x}\langle z\rangle$

$\overline{x}\langle y\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$ *can evolve to*

$\overline{y}\langle v\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{x}\langle z\rangle$ *or* $\overline{x}\langle y\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{z}\langle v\rangle$

and

$\overline{y}\langle v\rangle \mid !x(u).\overline{u}\langle v\rangle \mid \overline{z}\langle v\rangle$

Church’s Encoding of Booleans

$$
\begin{array}{lcl}
\boldsymbol{True}(b) & \equiv & b(t,f).\bar{t} \\
\boldsymbol{False}(b) & \equiv & b(t,f).\overline{f} \\
\boldsymbol{Not}(b,c) & \equiv & b(t,f).\bar{c}(f,t)
\end{array}
$$

$$
\begin{array}{l}
(\upsilon\, c)(\boldsymbol{Not}(b,c) \mid \boldsymbol{True}(c)) = \boldsymbol{False}(b) \\
(\upsilon\, c)(b(t,f).\bar{c}(f,t) \mid c(t,f).\bar{t}) = b(t,f).\overline{f}
\end{array}
\qquad ?
$$

Actions:

$a(\tilde{b})$	Input action; *x* is the name at which it occurs, $\tilde{b}$ is the tuple of names which are received
$\overline{a}\langle\tilde{b}\rangle$	Output action; *x* is the name at which it occurs, $\tilde{b}$ is the tuple of names which are emitted
$(\upsilon\,\tilde{x})a\langle\tilde{b}\rangle$	Output action; *x* is the name at which it occurs, $\tilde{b}$ is the tuple of names which are emitted; $(\upsilon\,\tilde{x})$ denotes private names which are carried out from their current scope (*scope extrusion*)
τ	Silent action; this action denotes unobservable internal communication.

Labelled Transition Semantics

$$\mathrm{IN}: a(\tilde{x}).P \xrightarrow{a(\tilde{b})} P\{\tilde{x} \backslash \tilde{b}\} \qquad \mathrm{OUT}: \bar{a}\langle\tilde{b}\rangle \xrightarrow{\bar{a}\langle\tilde{b}\rangle} \mathbf{0}$$

$$\mathrm{OPEN}: \frac{P \xrightarrow{(\upsilon\,\tilde{x})\bar{a}\langle\tilde{b}\rangle} P' \quad y \neq a \quad y \in \tilde{b} - \tilde{x}}{(\upsilon\, y)P \xrightarrow{(\upsilon\, y,\tilde{x})\bar{a}\langle\tilde{b}\rangle} P'}$$

$$\mathrm{COM}: \frac{P \xrightarrow{\bar{a}\langle\tilde{b}\rangle} P' \qquad Q \xrightarrow{a(\tilde{b})} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\boxed{\begin{array}{l} \mathrm{fn}(a(\tilde{b})) = \{a\} \\ \mathrm{fn}(a\langle\tilde{b}\rangle) = \{a, \tilde{b}\} \\ \mathrm{fn}((\upsilon\,\tilde{x})a\langle\tilde{b}\rangle) = \{a, \tilde{b}\} - \{\tilde{x}\} \\ \mathrm{fn}(\tau) = \varnothing \end{array}}$$

$$\boxed{\begin{array}{l} \mathrm{bn}(a(\tilde{b})) = \{\tilde{b}\} \\ \mathrm{bn}(a\langle\tilde{b}\rangle) = \varnothing \\ \mathrm{bn}((\upsilon\,\tilde{x})a\langle\tilde{b}\rangle) = \{\tilde{x}\} \\ \mathrm{bn}(\tau) = \varnothing \end{array}}$$

$$\mathrm{CLOSE}: \frac{P \xrightarrow{(\upsilon\,\tilde{x})\bar{a}\langle\tilde{b}\rangle} P' \quad Q \xrightarrow{a(\tilde{b})} Q' \quad \tilde{x} \notin \mathrm{fn}(Q)}{P \mid Q \xrightarrow{\tau} (\upsilon\,\tilde{x})(P' \mid Q')}$$

$$\mathrm{RES}: \frac{P \xrightarrow{\alpha} P' \qquad x \notin \mathrm{n}(\alpha)}{(\upsilon\, x)P \xrightarrow{\alpha} (\upsilon\, x)P'}$$

$$\mathrm{PAR}: \frac{P \xrightarrow{\alpha} P' \qquad \mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \varnothing}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\mathrm{REPL}: \frac{a(x).P \xrightarrow{a(\tilde{b})} P\{\tilde{x} \backslash \tilde{b}\}}{!a(x).P \xrightarrow{a(\tilde{b})} P\{\tilde{x} \backslash \tilde{b}\} \mid\ !a(x).P}$$

Some Facts

• The side conditions in the transition rules ensure that names do not become accidentally be bound or captured.
In the rule RES the side condition prevents transitions like

$$(\upsilon\, x)a(b).P \xrightarrow{a(x)} (\upsilon\, x)P\{b \backslash x\}$$

which would violate the static binding assumed for restriction.

• In the given system bound names of an input are instantiated as soon as possible, namely in the rule for input - it is therefore an *early* transition system. Late instantiation is done in the rule for communication.
• The given system implements an asynchronous variant of the π-calculus. Therefore, output action are not directly observable.
• There is no rule for α-conversion. It is assumed that α-conversion is always possible.

Experiments

$$(\upsilon\, c)(\boldsymbol{Not}(b,c) \mid \boldsymbol{True}(c)) = \boldsymbol{False}(b)$$
$$(\upsilon\, c)(b(t,f).\overline{c}(f,t) \mid c(t,f).\overline{t}) = b(t,f).\overline{f}$$

?

Experiment 1:

$$\begin{array}{cc} & (\upsilon\, c)(b(t,f).\overline{c}(f,t) \mid c(t,f).\overline{t}) \\ \xrightarrow{\overline{b}\langle x,y\rangle} & (\upsilon\, c)(\overline{c}(y,x) \mid c(t',f').\overline{t'}) \\ \xrightarrow{\tau} & (\upsilon\, c)(\overline{y}) \\ \xrightarrow{y()} & \mathbf{0} \end{array}$$

Experiment 2:

$$\begin{array}{cc} & b(t,f).\overline{f} \\ \xrightarrow{\overline{b}\langle x,y\rangle} & \overline{y} \\ \xrightarrow{y()} & \mathbf{0} \end{array}$$

Using strong bisimulation, the systems are not equivalent. Furthermore, an asynchronous observer can only indirectly see that an output message has been consumed.

Bisimulation - A Board Game

The central idea of bisimulation is that an external observer performs experiments with both processes *P* and *Q* observing the results in turn in order to match each others process behaviour step-by-step.

Checking the equivalence of processes this way one can think of this as a game played between two persons, the ***unbeliever***, who thinks that *P* and *Q* are not equivalent, and the ***believer***, who thinks that *P* and *Q* are equivalent. The underlying strategy of this game is that the unbeliever is trying to perform a process transition which cannot be matched by the believer.

Synchronous Interactions

There exists two kinds of experiments to check process equivalence: *input-experiments* and *output-experiments*. Both experiments are triggered by their corresponding opposite action.

In the synchronous case, input actions for a process *P* are only generated if there exists a matching receiver that is enabled within *P*. The existence of an input transition such that *P* evolves to *P'* reflects precisely the fact that a message offered by the observer has actually been consumed.

Asynchronous Interactions

In an synchronous system the sender of an output message does not know when the message is actually consumed. In other words, at the time of consumption of the message, its sender is not participating in the event anymore. Therefore, an asynchronous observer, in contrast to a synchronous one, cannot directly detect the input actions of the observed process. We need therefore a different notion of input-experiment.

Solution: Asynchronous input-experiments are incorporated into the definition of bisimulation such that inputs of processes have to be simulated only indirectly by observing the output behaviour of the process in context of arbitrary messages (e.g., $P \mid \overline{a}\langle \widetilde{b} \rangle$).

The Silent Action

Strong bisimulation does not respect silent actions (τ-transitions).

Silent transitions denote unobservable internal communication. From the observer's point of view we can only notice that the system takes more time to respond.

Silent actions do not denote any interacting behaviour. Therefore, we may consider two systems P and Q to be equivalent if they only differ in the number of internal communications.

We write $P \overset{\alpha}{\Rightarrow} P'$ if $P\,(\xrightarrow{\tau})^{*}\xrightarrow{\alpha}(\xrightarrow{\tau})^{*}\,P'$. In other words, a given observable action can have an arbitrary number of preceding or following internal communications.

Asynchronous Bisimulation

A binary relation ***S*** over processes *P* and *Q* is a weak (observable) bisimulation if it is symmetric and *P* ***S*** *Q* implies

- whenever $P \xrightarrow{\alpha} P'$, where α is either τ or output with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(P|Q) = \emptyset$, then *Q'* exists such that $Q \overset{\alpha}{\Rightarrow} Q'$ and *P'* ***S*** *Q'*.

- $(P \mid \overline{a}\langle \widetilde{b} \rangle)$ ***S*** $(Q \mid \overline{a}\langle \widetilde{b} \rangle)$ for all messages $\overline{a}\langle \widetilde{b} \rangle$.

Two processes *P* and *Q* are weakly bisimular, written $P \approx Q$, if there is a weak bisimulation ***S*** with *P* ***S*** *Q*.

Some Facts

• $\approx$ is an equivalence relation.

• $\approx$ is a congruence relation.

• Leading τ-transitions are significant, i.e., they cannot be omitted.

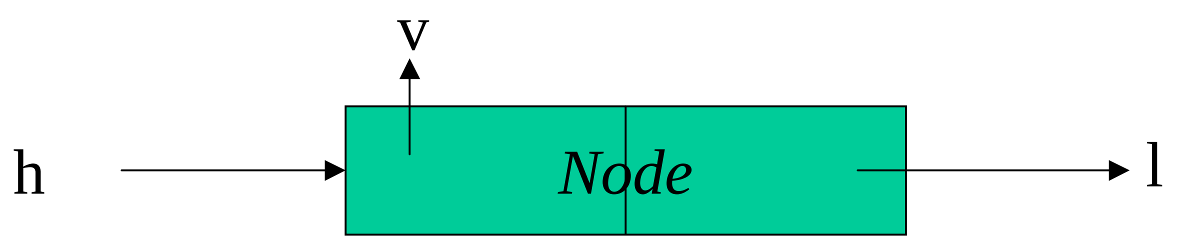
• Asynchronous bisimulation is the framework that enables us to state $P = Q$ iff $P \approx Q$ and vice versa.

An Simple Object Model

$$
\begin{aligned}
ReferenceCell \equiv (\upsilon\, v, s, g) \\
&(\ \overline{v}\langle 0 \rangle \\
&\ \mid\, !s(n, r).v(\_).(\overline{v}\langle n \rangle \mid \overline{r}\langle\rangle) \\
&\ \mid\, !g(r).v(i).(\overline{v}\langle i \rangle \mid \overline{r}\langle i \rangle)\ )
\end{aligned}
$$

A List

A list is either *Nil* or *Cons* of value and a list.



The constant *Nil*, the construction *Cons*(*V*, *L*), and a list of *n* values are defined as follows:

$$\begin{array}{lcl} Nil & = & h(n,c).\overline{n} \\ Cons(V,L) & = & (\upsilon\, v,l)(h(n,c).\overline{c}(v,l) \,|\, V\langle v\rangle \,|\, L\langle l\rangle) \\ [V_1,\ldots V_n] & = & Cons(V_1, Cons(\ldots, Cons(V_n, Nil)\ldots)) \end{array}$$

A Concurrent Language

V ::= X | Y | …	Variable
F ::= + | - | … | 0 | 1 | …	Function symbols
C ::= $V = E$	Assignment
C ; C	Sequential Composition
if E then C else C	Conditional Statement
while E do C	While Statement
let D in C end	Declaration
C par C	Parallel Composition
input V	Input
output E	Output
skip	
D ::= var V	Variable Declaration
E ::= V	Variable Expression
$F(E_1,\ldots, E_n)$	Function Call

Ambiguous Meaning

```
X = 0;
X = X + 1 par X = X +2
```

What is the value of X at the end of the second statement?

Basic Elements

• We assume that each element of the source language is assigned a process expression.

Variables: $$\begin{aligned} X(init) \equiv (\upsilon\, v, setX, getX)\, & \\ (\, \overline{v}\langle init\rangle & \\ \mid !setX(n,r).v(\_).(\overline{v}\langle n\rangle \mid \overline{r}\langle\rangle) & \\ \mid !getX(r).v(i).(\overline{v}\langle i\rangle \mid \overline{r}\langle i\rangle)\,) & \end{aligned}$$

Skip: $\overline{done}\langle\rangle$

C$_1$; **C$_2$** = $(\upsilon\, c)(C_1\{done \backslash c\} \mid c().C_2)$

C$_1$ par **C$_2$** = $$\begin{aligned} & (\upsilon\, l,r,t)(\overline{t}\langle \text{true}\rangle \mid C_1\{done\backslash l\} \mid C_2\{done\backslash r\} \mid \\ & (l().t(b).(\, if\ b\ then\ r().Skip\ else\ \overline{l}\langle\rangle \mid \overline{t}\langle\, false\rangle) \mid \\ & (r().t(b).(\, if\ b\ then\ l().Skip\ else\ \overline{r}\langle\rangle \mid \overline{t}\langle\, false\rangle)) \end{aligned}$$

Expressions

X = $(\upsilon\, ack)(\overline{getX}(ack) \mid ack(v).\overline{res}(v))$

F(E$_1$,…,E$_n$) = $arg_1(x_1).....arg_n(x_n).\overline{F}(x_1,...,x_n,res)$

M[F(E$_1$,…,E$_n$)] =

$(\upsilon\, arg_1,…,arg_n)(M[E_1]\{res \backslash arg_1\} \mid …$

$M[E_n]\{res \backslash arg_n\} \mid M[F])$

Operation Sequence

```
X = 0;
X = X + 1 par X = X +2
```

What is the value of X at the end of the second statement?

According to the former definitions the value of X is either 1, 2, or 3. The three values are possible since every atomic action can occur in an arbitrary and meshed order.

To guarantee a specific result (e.g., 1 or 2), we need to employ semaphors.

What have we learned?

• Classical automata theory does not cope correctly with interacting behaviour
• Bisimulation is an equivalence relation defined over a labelled transition system which respects non-determinism andcan therefore be used to compare the observable behaviour of interacting systems.
• The π-calculus is a name-passing system in which program progress is expressed by communication.
• Which the π-calculus we can model higher-level programming abstractions like objects and lists.
• A concurrent programming language can be assigned a semantics based on the π-calculus.

Software Composition Group

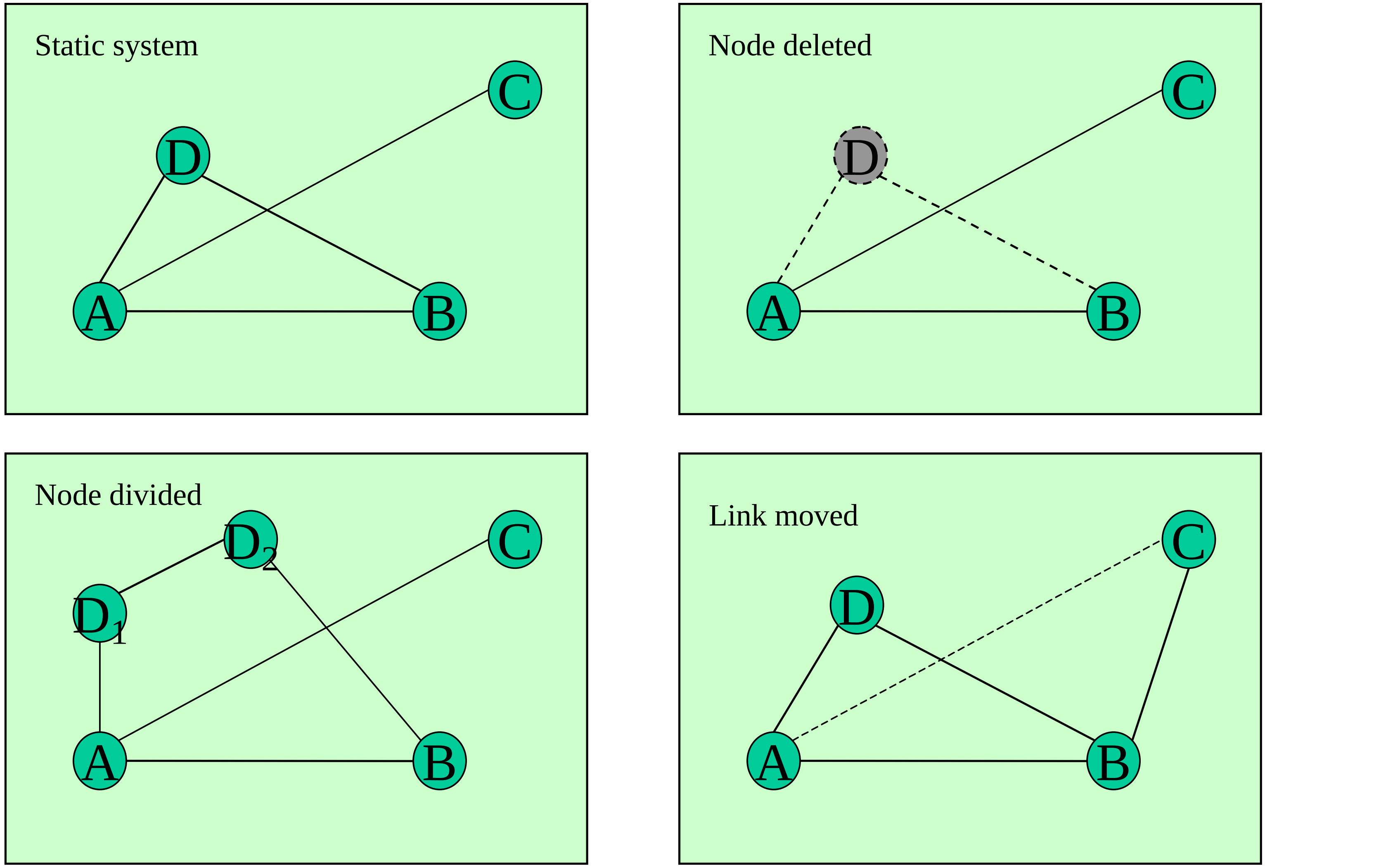
University of Berne

Research:

- Piccola - a small composition language
- The πL-calculus - a formal foundation for software composition.
- COORDINA - coordination models and languages

Resources: **http://www.iam.unibe.ch/~scg**

Evolving Automata


Static system
A
B
C
D
Node deleted
A
B
C
D
Node divided
A
B
C
D_1
D_2
Link moved
A
B
C
D