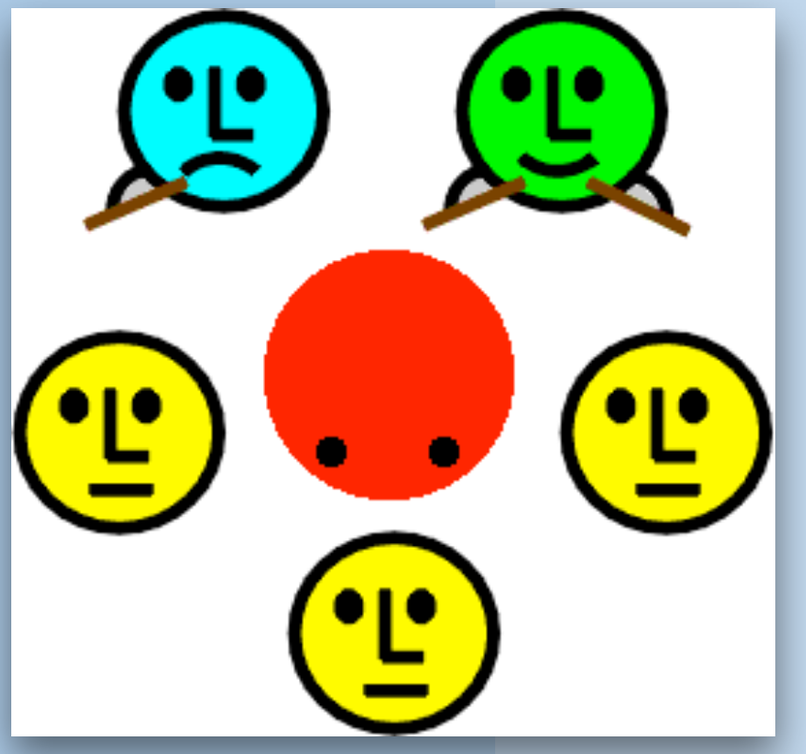
b
UNIVERSITÄT
BERN

5. Liveness and Guarded Methods

Oscar Nierstrasz



Roadmap

- Liveness
  - —Progress Properties
- Deadlock
  - —The Dining Philosophers problem
  - —Detecting and avoiding deadlock
- Guarded Methods
  - —Checking guard conditions
  - —Handling interrupts
  - —Structuring notification

Roadmap

- **Liveness**
  - —Progress Properties
- Deadlock
  - —The Dining Philosophers problem
  - —Detecting and avoiding deadlock
- Guarded Methods
  - —Checking guard conditions
  - —Handling interrupts
  - —Structuring notification

Liveness

- A liveness property asserts that *something good eventually happens*

- A progress property asserts that it is *always the case that an action is eventually executed*

- Progress is the opposite of starvation, the name given to a concurrent programming situation in which *an action is never executed*

Liveness Problems

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

Starvation: (AKA "indefinite postponement")

> The system as a whole makes progress, but some individual processes don't

Dormancy:

> A waiting process fails to be woken up

Premature termination:

> A process is killed before it should be

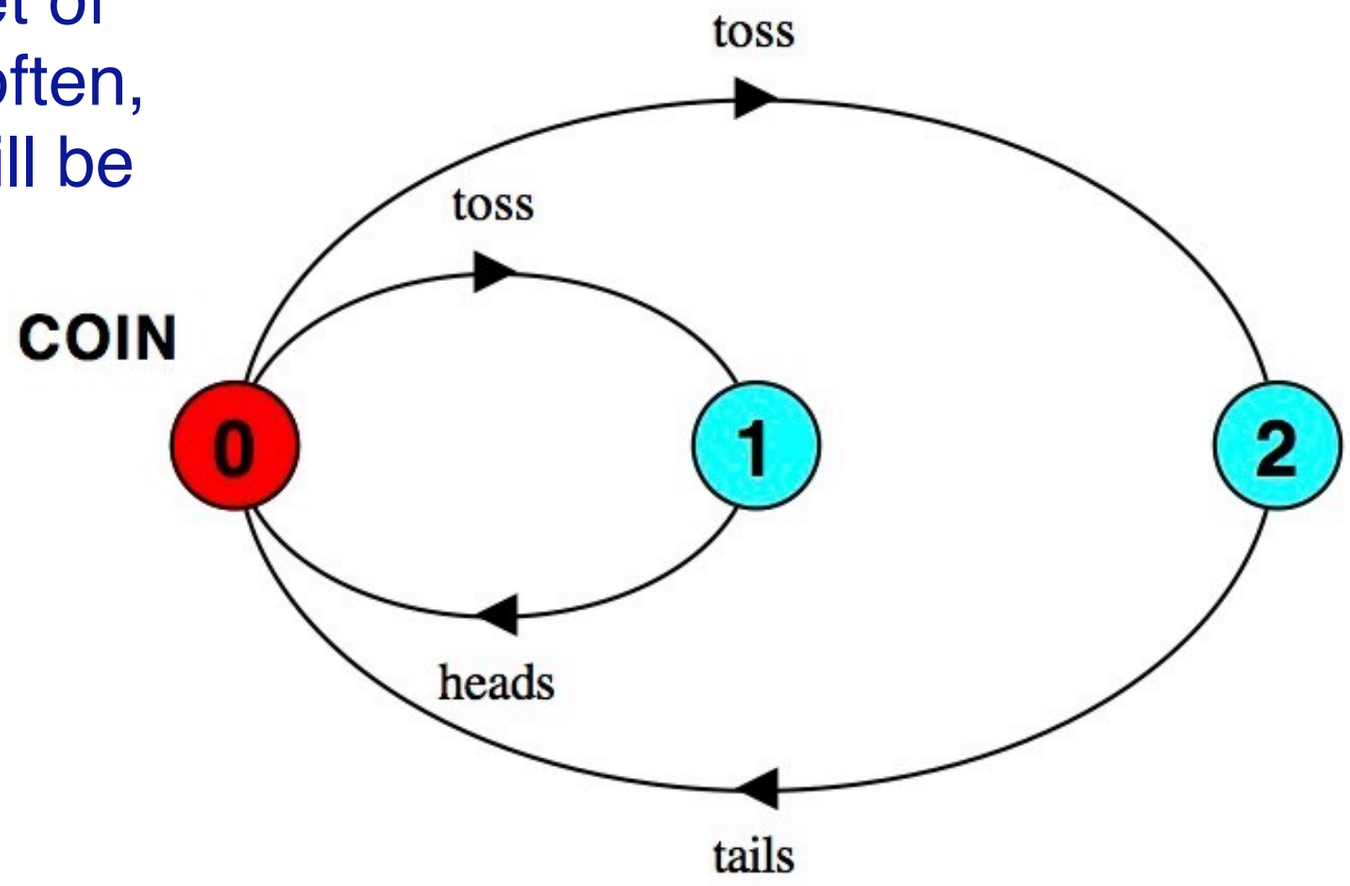
Deadlock:

> Two or more processes are blocked, each waiting for resources held by another

There are various kind of liveness problems. Deadlock is perhaps the most notorious, and we will look at it in some detail, but there are many other kinds. Starvation and dormancy are particularly evil.

Progress properties — fair choice

Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.



If a coin were tossed an infinite number of times, we would expect that both heads and tails would each be chosen infinitely often.
This assumes fair choice !

```
COIN = ( toss -> heads -> COIN
       | toss -> tails -> COIN ).
```

The notion of fair choice doesn’t say anything about the relative frequency or the probably of any given choice, just that all choices will be made infinitely often. In this case both heads and tails will come up infinitely often (even if we get 100 x as many heads as tails!).

Safety vs. Liveness

Consider:

```
property SAFE =    ( heads -> SAFE
                   | tails -> SAFE ).
```

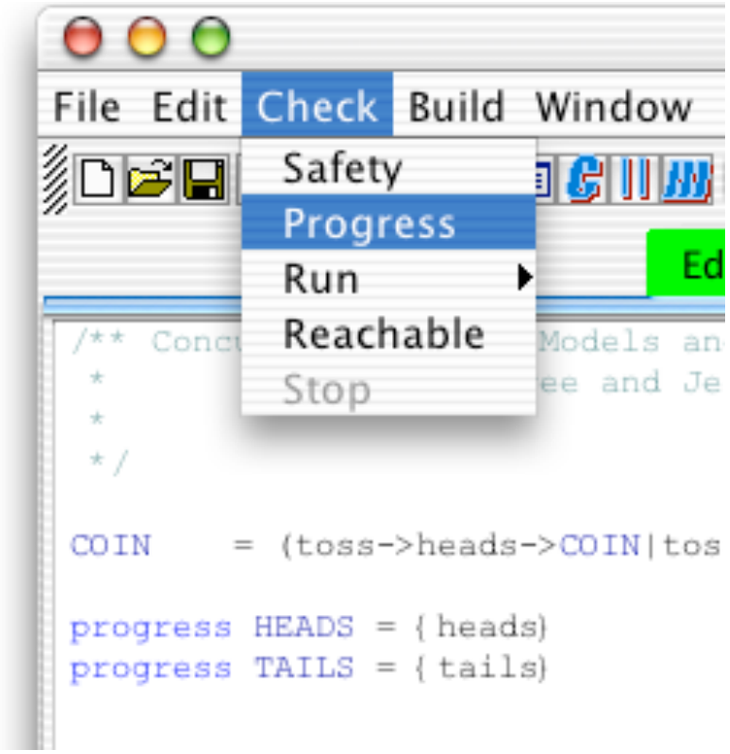
The safety properties of COIN are not very interesting.
How do we express what must happen?

Safety just tells us what may happen, not what must happen. We need a difference way of stating liveness properties.

Progress properties

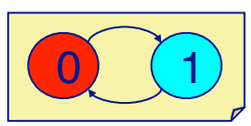
```
progress P = {a1,a2..an}
```

asserts that in an *infinite execution* of a target system, *at least one* of the actions a1,a2...an will be executed *infinitely often*.




```
progress HEADS = {heads}
progress TAILS = {tails}
```

```
No progress violations detected.
```



1-coin.lts

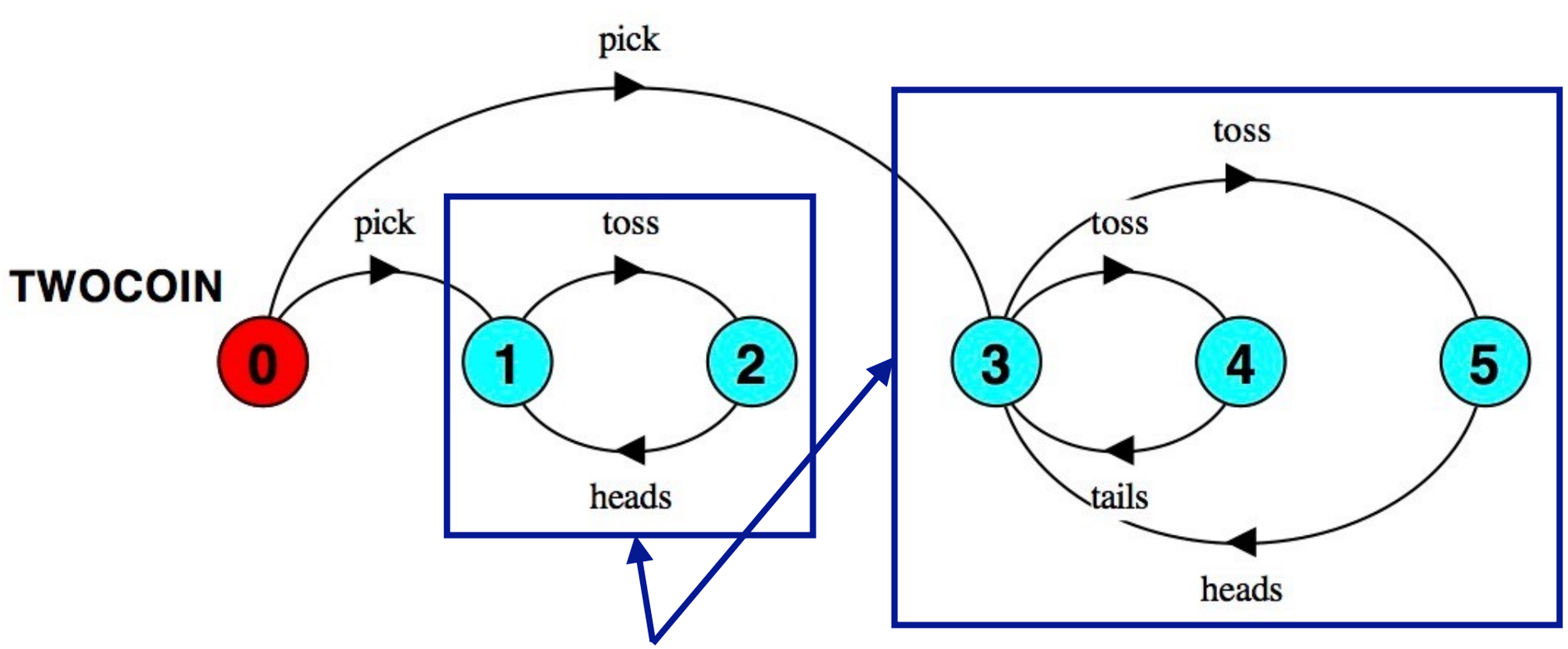
A progress property consists of a set of transitions, at least one of which will be selected infinitely often in an infinite sequence of choices.

Note that if we want both HEADS and TAILS to be fairly chosen, we need two progress properties, one for each of them.

Progress properties

Suppose we have both a normal coin and a trick coin

```
TWOCOIN = ( pick->COIN | pick->TRICK ),
TRICK   = ( toss->heads->TRICK ),
COIN    = ( toss->heads->COIN | toss->tails->COIN ).
```



Terminal sets

A terminal set is a set of mutually reachable states in the labeled transition system from which one cannot escape. In this system, after picking one of the two coins, one stays within one of the subgraphs `COIN` or `TRICK`.

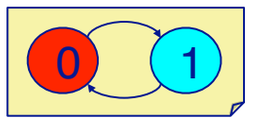
Progress analysis

A terminal set of states is one in which *every state is mutually reachable* but *no transitions lead out of the set*.

The terminal set {1, 2} violates progress property TAILS

```
progress HEADS = {heads}
progress TAILS = {tails}
progress HEADSorTAILS = {heads,tails}
```

```
Progress violation: TAILS
Trace to terminal set of states: pick
Actions in terminal set: {toss, heads}
```



LTSA will perform model checking to verify the progress properties. It will either tell us that all specified properties hold, or it will provide a counter-example, i.e., a trace to a state that violates the given property.

In this case it discovers that after a `pick` transition we may reach a terminal set including states (1) and (2) that violates the property `TAILS` (i.e., no tail transitions are possible any more).

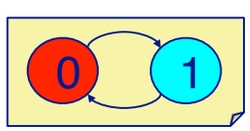
Safety vs. Liveness

Consider:

```
property REPICK = ( pick -> toss -> REPICK ).
```

```
Trace to property violation in REPICK:
        pick
        toss
        heads
        toss
```

How does this safety property expose the flaw in the system?
How would you fix the TWOCOIN to have this property pass?



2-twoCoin.lts

If you are allowed to pick the coin after each toss, the starvation problem goes away (though there is still a fairness issue).

How can we fix the `TWOCOIN` *system so this safety property passes?*

Roadmap

- Liveness
  - —Progress Properties
- **Deadlock**
  - —The Dining Philosophers problem
  - —Detecting and avoiding deadlock
- Guarded Methods
  - —Checking guard conditions
  - —Handling interrupts
  - —Structuring notification

Deadlock

Four necessary and sufficient conditions for deadlock:

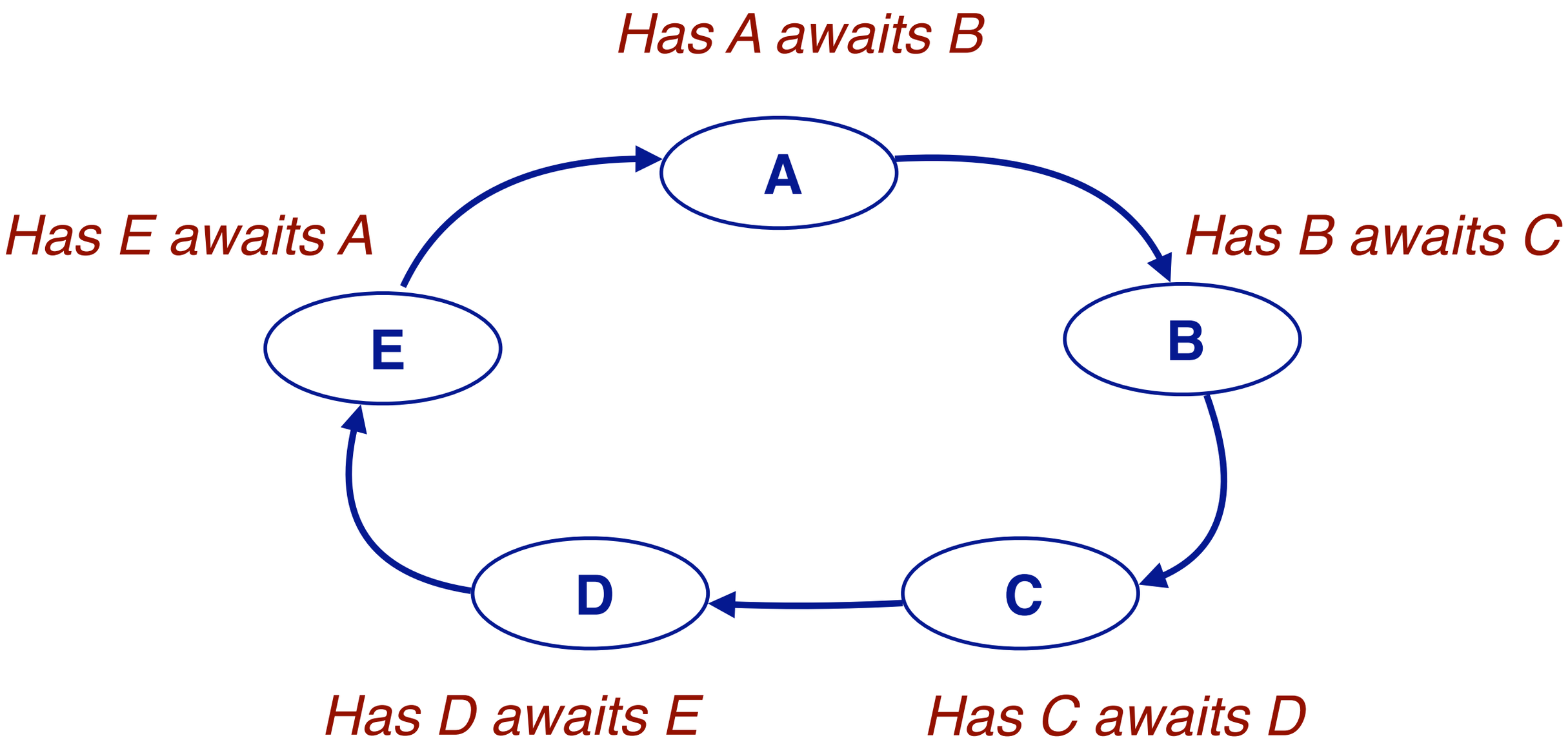
1. ***Serially reusable resources:***
   — the deadlocked processes *share resources under mutual exclusion.*
2. ***Incremental acquisition:***
   — processes *hold on to acquired resources while waiting* to obtain additional ones.
3. ***No pre-emption:***
   — once acquired by a process, *resources cannot be pre-empted* but only released voluntarily.
4. ***Wait-for cycle:***
   — a *cycle of processes* exists in which each process holds a resource which its successor in the cycle is waiting to acquire.

If any of these conditions is removed, deadlock cannot take place.

Without mutual exclusion, resources are not locked, so there can be no deadlock. If resources are not obtained incrementally, then no process can hold onto resources while waiting for others. If a process can be pre-empted, deadlock can be broken. Finally, without a waits-for cycle, there is no deadlock in the first place.

Approaches to resolving deadlock either try to detect and break deadlock by lifting one of these conditions, or they try to avoid deadlock by ensuring that certain conditions cannot arise.

Waits-for cycle



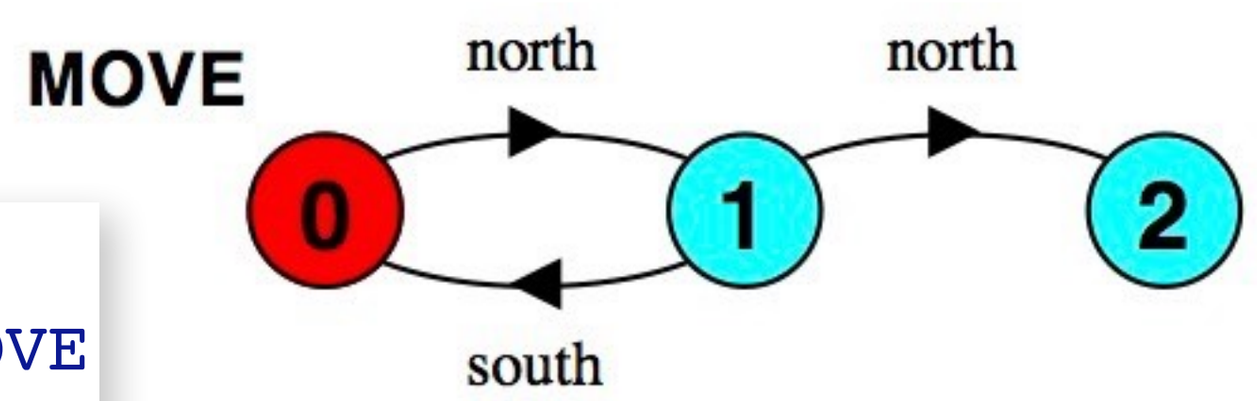
A classical deadlock always has a “waits-for” cycle with at least two participants. Each one holds some resources waited for by another participant. No one will release the resources they already have, so they are all “deadlocked”.

Deadlock analysis - primitive processes

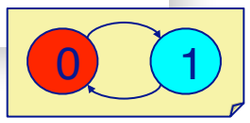
A deadlocked state is one with no outgoing transitions

In FSP: `STOP` process

```
MOVE = ( north ->
         ( south -> MOVE
         | north -> STOP
         )
       ).
```




```
Progress violation for actions: {north, south}
Trace to terminal set of states: north north
Actions in terminal set: {}
```

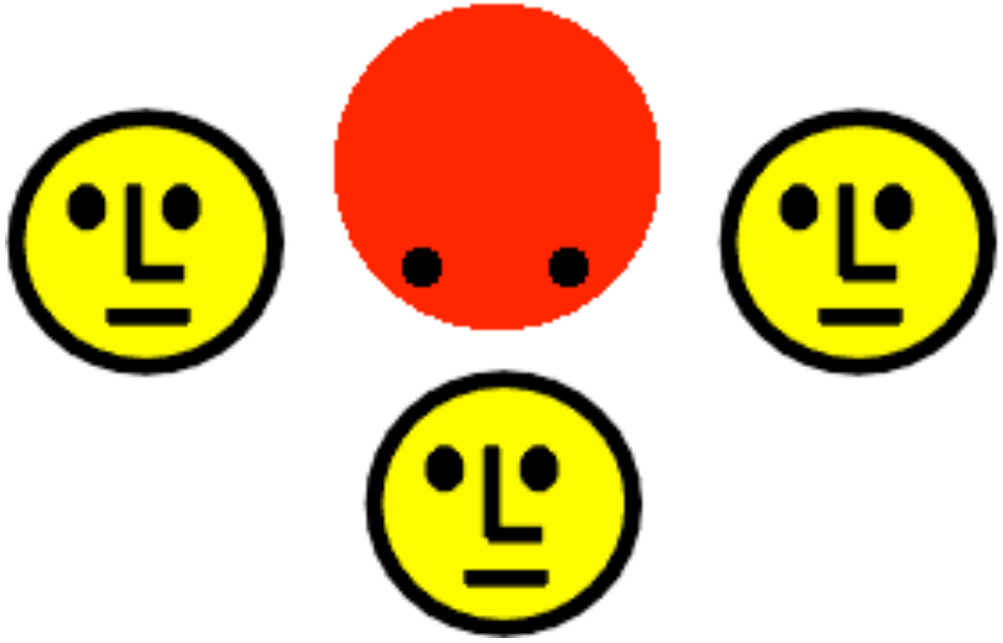


3-move.lts

A deadlocked state in LTSA is a particular kind of terminal set with no outgoing transitions. Such a process (state) is also known as `STOP`.

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 must be *no deadlock and no starvation.*
- Want efficient behaviour under absence of contention.



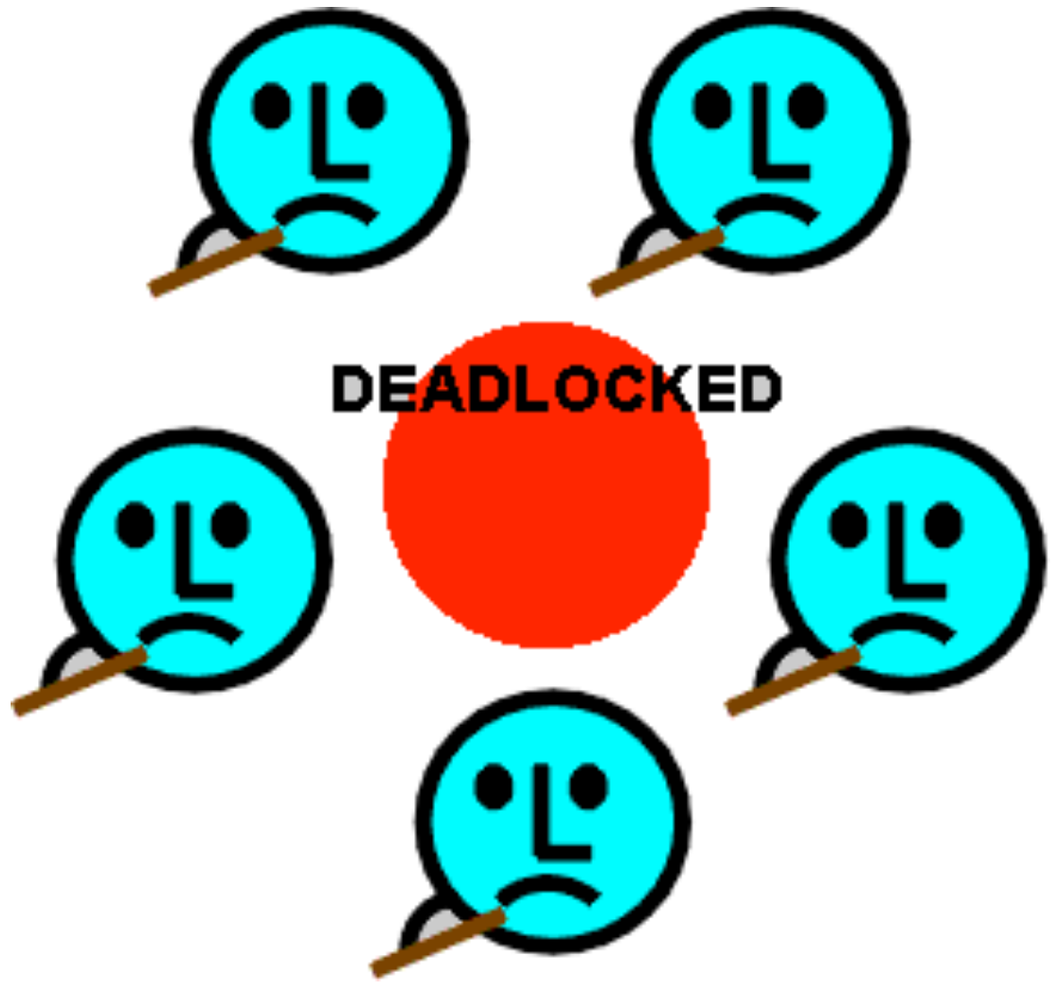
The “dining philosophers” problem is a classical problem meant to illustrate many issues in concurrent programming. There are five philosophers sitting around a table, each of whom takes turns thinking and eating. On the table is a large bowl of noodles. Between each philosopher is a fork (or a chopstick). Each philosopher needs two forks (or chopsticks) to eat.

A deadlock may arise if each philosopher succeeds in grabbing one fork, and then waits for his neighbour to release the other fork.

There are numerous variants of the problem, for example in which the philosophers may get up and sit down, in order to eat or to think, leading to different solutions.

Deadlocked diners

- A deadlock occurs if a *waits-for cycle* arises in which each philosopher grabs one fork and waits for the other.



Dining Philosophers, Safety and Liveness

Dining Philosophers illustrate 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
Race conditions	Anomalous behaviour depends on timing

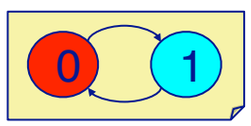
Modeling Dining Philosophers

```
PHIL  =  (  sitdown
         -> right.get -> left.get
         -> eat -> left.put -> right.put
         -> arise -> PHIL ).

FORK  =  (  get -> put -> FORK ).

||DINERS(N=5)=  forall [i:0..N-1]
   ( phil[i]:PHIL || {phil[i].left,phil[((i-1)+N)%N].right}::FORK ).
```

Is this system safe? Is it live?



4-DiningPhilosophers.lts

In this variant, philosophers get up from the table to think and sit down to eat. Note that each philosopher prepares to eat by first grabbing the right fork and then the left fork.

What do safety and liveness mean for this system?

Caveat: LTSA considers deadlock to be *both* an issue of liveness and safety. (It considers deadlock to be a kind of error state.)

Dining Philosophers Analysis

```
Trace to terminal set of states:
        phil.0.sitdown
        phil.0.right.get
        phil.1.sitdown
        phil.1.right.get
        phil.2.sitdown
        phil.2.right.get
        phil.3.sitdown
        phil.3.right.get
        phil.4.sitdown
        phil.4.right.get
Actions in terminal set: {}
```

No further progress is possible due to the waits-for cycle

As always, LTSA reports a progress violation by presenting a trace that leads to the erroneous state. Here we clearly see that each philosopher has grabbed his right fork and no further progress is possible.

How many different traces exist leading to a deadlock?

Eliminating Deadlock

There are two fundamentally different approaches to eliminating deadlock.

Deadlock detection:

> Repeatedly check for waits-for cycles. When detected, choose a victim and force it to release its resources.
  —Common in transactional systems; the victim should “roll-back” and try again

Deadlock avoidance:

> Design the system so that a waits-for cycle cannot possibly arise.

In both cases we try to break one of the four necessary conditions for deadlock. In the first case we break condition 3 by detecting deadlock and then breaking it. In the second case we avoid deadlock by ensuring that condition 4 cannot be reached?

How could we solve the problem by breaking conditions 1 or 2?

Dining Philosopher Solutions

There are many solutions offering varying degrees of liveness guarantees:

Break the cycle

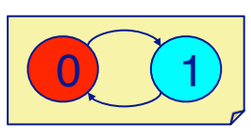
> Number the forks. Philosophers grab the *lowest numbered fork* first.

> One philosopher grabs forks in the reverse order.

Philosophers queue to sit down

> allow *no more than four* at a time to sit down

Do these solutions avoid deadlock? What about starvation? Are they "fair"?



5-DiningPhilosophersSol.lts

Roadmap

- Liveness
  - —Progress Properties
- Deadlock
  - —The Dining Philosophers problem
  - —Detecting and avoiding deadlock
- **Guarded Methods**
  - —Checking guard conditions
  - —Handling interrupts
  - —Structuring notification

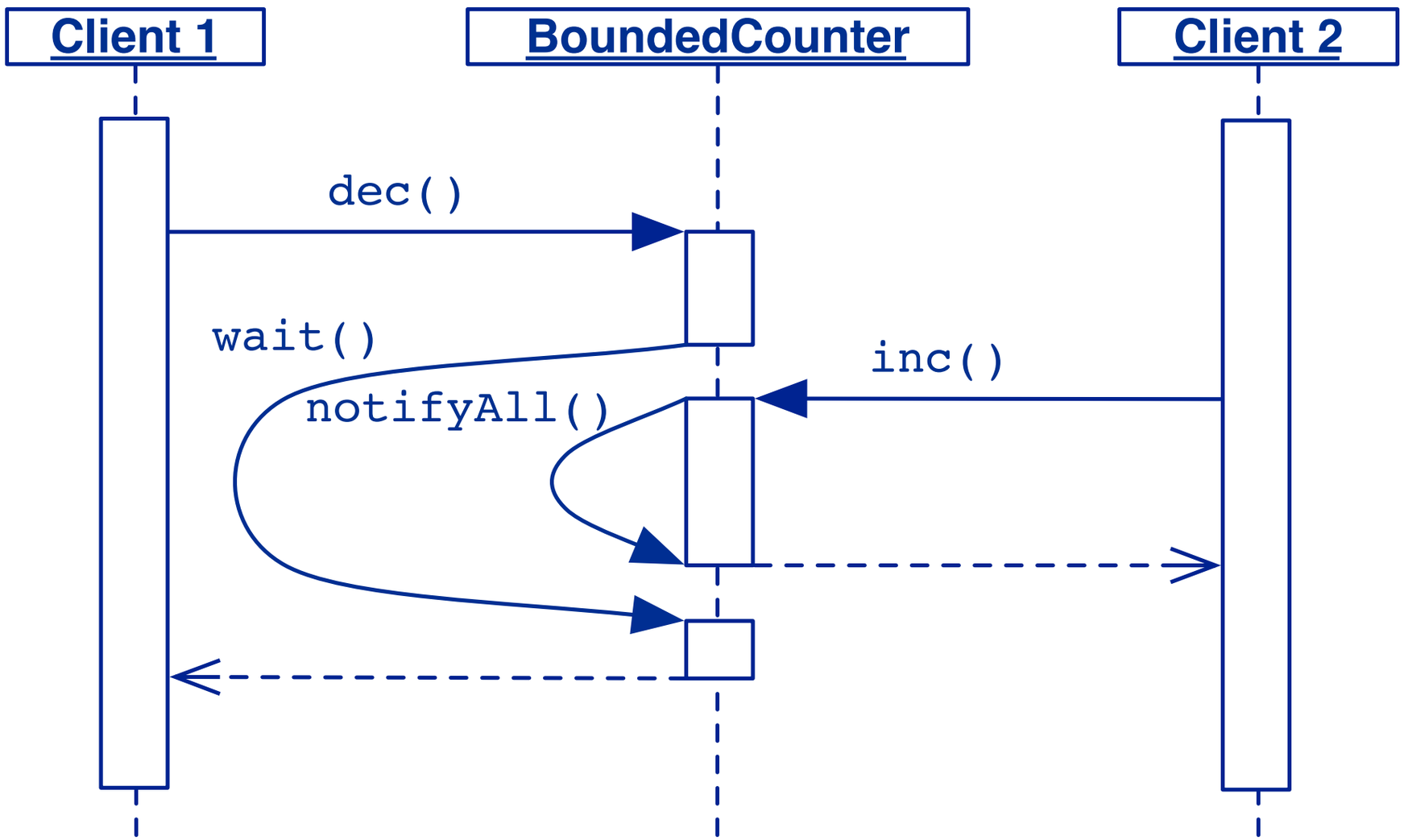
Achieving Liveness

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

Pattern: 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).



This is perhaps the most basic pattern that we will see. In this scenario a client invokes dec to decrement the bounded counter while it already is at its minimum value. Instead of balking, it checks the guard condition, detects that the counter is zero, and waits for the condition to change, thus releasing the mutual exclusion lock. Another thread invokes inc, incrementing the counter and thus changing the condition. It invokes `notifyAll`, waking all waiting threads, which may then proceed as soon as inc releases its lock.

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.)

- …

Guarded Methods — applicability …

…

- You can *avoid or cope with liveness problems* due to waiting threads retaining all synchronization locks.

- 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.)

Predicates may not be computable, for example, if they depend on external events.

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();
      return result;
   }
```

Step: 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;   // min value
  public static final long MAX = 10;  // max value
  public long value();    // inv't: MIN <= value() <= MAX
                          // init: value() == MIN
  public void inc();      // pre: value() < MAX
  public void dec();      // pre: value() > MIN
}
```

Counter

We will actually see many different versions of the BoundedCounter interface, implementing different synchronization policies.

Step: 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.

Step: Check guard conditions ...

- 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.

It is extremely rare in practice to see examples in which notification guarantees that the guard condition is true *and will remain true*. Optimizing the wait loop to a simple if test will not buy you much, and could lead to obscure bugs if the nature of the guard condition changes

Step: Handle interrupts

- Establish a *policy* to deal with InterruptedExceptions. Possibilities include::
  - —**Ignore interrupts** (i.e., an empty catch clause), which preserves safety at the possible expense of liveness. (Not recommended!)
  - —**Terminate** the current thread (stop). This 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. (Easiest thing to do.)
  - —**Cleanup** and restart.
  - —**Ask** for user intervention before proceeding.

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

Step: 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:
  - —use `notifyAll` to *wake up all threads* that are blocked in waits for the host object.
  - —use `notify` to wake 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*.)

Testing for safety violations

```
public abstract class BoundedCounterAbstract
    implements BoundedCounter {
  protected long count = MIN;
  private int errors = 0;

  protected void checkInvariant() {
     if (! (count >= BoundedCounter.MIN
          && count <= BoundedCounter.MAX) ) {
        errors++;
     }
  }
  public int errors() {
     return errors;
  }
}
```

Common behaviour to help us test for safety violations

Counter

In all of our examples of implementations of the `BoundedCounter` interface, we run tests that exercise the interface and catch any violations of the invariant.

Basic synchronization

```
public class BoundedCounterBasic
  extends BoundedCounterAbstract { ...
 public synchronized void inc() {
  while (count >= MAX) {
   try { wait(); }
    catch(InterruptedException ex) { };
  }
  count ++;
  notifyAll();
  checkInvariant(); // record safety violations
 }
 ...
}
```

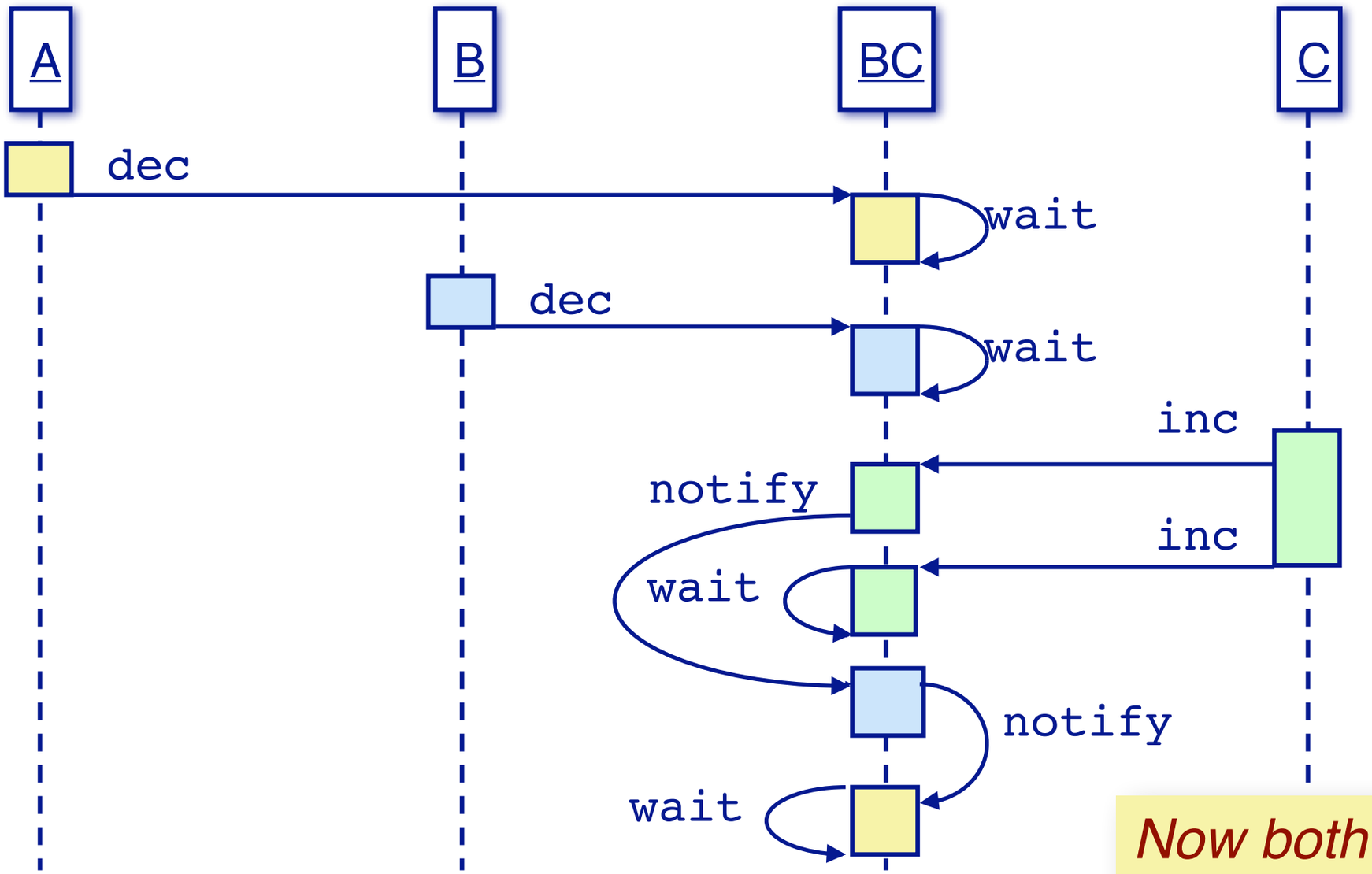
Race conditions

```
public class BoundedCounterNoSyncBAD
  extends BoundedCounterAbstract {
 public void inc() {// missing synchronization
  while (count >= MAX) {
   Thread.yield();
  }
  Thread.yield();    // race condition here
  count ++;
  checkInvariant(); // possible safety violation
 }
}
```

NB: wait() and notify() are invalid outside synchronized code!

`notify()` vs. `notifyAll()`

Careless use of `notify()` may lead to *race conditions.*



Now both A and C wait for nothing!

This scenario illustrates why issuing `notify` instead of `notifyAll` can lead to a race condition.

A and B both attempt a decrement and wait. C increments twice, waking up B and then waiting. Now B decrements and notifies A. A and C are now left waiting.

NB: It is always hard to create a test case that provoke a race condition. This is another reason why model-checking is more useful than testing to ensure safety and liveness.

Step: 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 may 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

Guards and assignments are encapsulated in helper methods:

```
public class BoundedCounterEncapsulatedAssigns
     extends BoundedCounterAbstract {
   ...
   public synchronized void inc() {
      awaitIncrementable();
      setCount(count + 1);
   }
   public synchronized void dec() {
      awaitDecrementable();
      setCount(count - 1);
   }
...
```

```
...
   protected synchronized void awaitIncrementable() {
      while (count >= MAX)
         try { wait(); }
         catch(InterruptedException ex) {};
   }
   protected synchronized void awaitDecrementable() {
      while (count <= MIN)
         try { wait(); }
         catch(InterruptedException ex) { };
   }
   protected synchronized void setCount(long newValue) {
      count = newValue;
      notifyAll();
   }
}
```

Encapsulating guards and assignments has two main consequences:

1. the code achieves a higher level of abstraction by hiding the synchronization policy;
2. the guards and assignment helper methods can potentially be reused across multiple client methods.

(In this example, just `setCount` is reused.)

Tracking State

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

```
public class BoundedCounterTrackingState
     extends BoundedCounterAbstract {
  ...
  public synchronized void inc() {
     while (count == MAX)
        try { wait(); }
        catch(InterruptedException ex) {};
     if (count++ == MIN)
        notifyAll();          // just left bottom state
  }
  ...
}
```

This pattern may be useful when only certain state changes could possibly wake a waiting thread. In the case of the bounded counter, threads can only be waiting if the counter was either in the maximum or the minimum state. In all other circumstances, there cannot possibly be any waiting threads.

As a consequence, we only need to issue a `notifyAll` when we leave either extreme state, i.e., when `inc` leaves the `MIN` state and when `dec` leaves the `MAX` state.

Tracking State Variables

```
public class BoundedCounterStateVariables
      extends BoundedCounterAbstract {
   protected enum State { BOTTOM, MIDDLE, TOP };
   protected State state = State.BOTTOM;

   public synchronized void inc() {
      while (state == State.TOP) { // consult logical state
         try { wait(); }
         catch(InterruptedException ex) {};
      }
      ++count;                     // modify actual state
      checkState();                // sync logical state
   }
...
```

```
...
   protected synchronized void checkState() {
      State oldState = state;
      if (count == MIN) { state = State.BOTTOM; }
      else if (count == MAX) { state = State.TOP; }
      else { state = State.MIDDLE; }

      if (leftOldState(oldState)) { notifyAll(); }
   }

   private boolean leftOldState(State oldState) {
      return state != oldState
            && ( oldState == State.TOP
               || oldState == State.BOTTOM);
   }
}
```

This pattern is pretty similar to the previous one, except it abstracts away from the concrete states.

Delegating 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!
      }
   }
}
```

In this pattern we have the variable holding the counter value itself issue the notifications. The design is slightly convoluted, as it separates the synchronized object from the observer that is notified.

Note that we must *synchronize with respect to the observer* before issuing `notifyAll`, or else an `IllegalMonitorException` will be raised.

Delegating notifications ...

Notification is delegated to the helper object:

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

The observer in this case is the bounded counter.

Note that threads wait with respect the bounder counter object(the observer), not the count variable. This explains why the count variable must issue `notifyAll` within a synchronized block on the *observer*, not itself, since threads are waiting on the observer.

What you should know!

- *What kinds of liveness problems can occur in concurrent programs?*
- *Why is progress a liveness rather than a safety issue?*
- *What is fair choice? Why do we need it?*
- *What is a terminal set of states?*
- *What are necessary and sufficient conditions for deadlock?*
- *How can you detect deadlock? How can you avoid it?*

Can you answer these questions?

- *How would you manually check a progress property?*
- *What is the difference between starvation and deadlock?*
- *How would you manually detect a waits-for cycle?*
- *What is fairness?*

What you should know!

- *When can you apply the Guarded Methods pattern?*
- *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?*
- *Why must you re-establish the class invariant before calling wait()?*
- *What should you do when you receive an InterruptedException?*
- *What is the difference between tracking state and using state-tracking variables?*

Can you answer these questions?

- *When are guarded methods better than balking?*
- *When should you use helper methods to implement guarded methods?*
- *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?*
- *When is the complexity of delegating notifications worthwhile?*