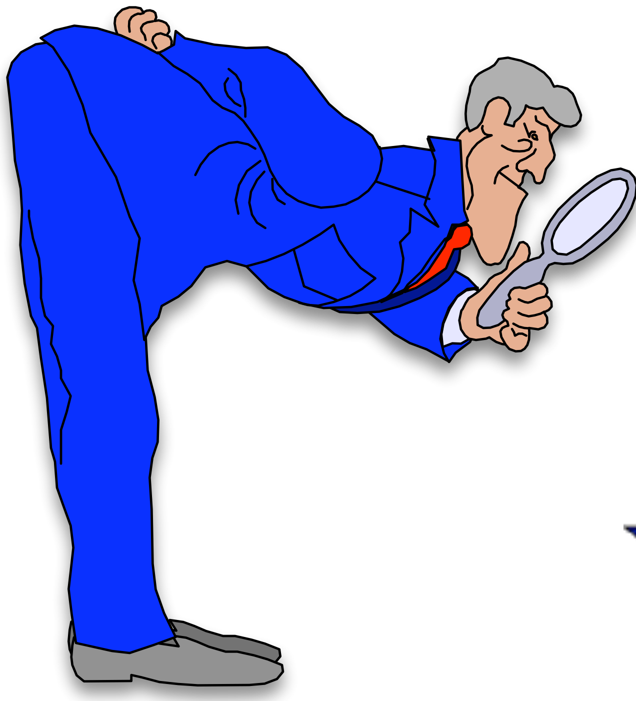
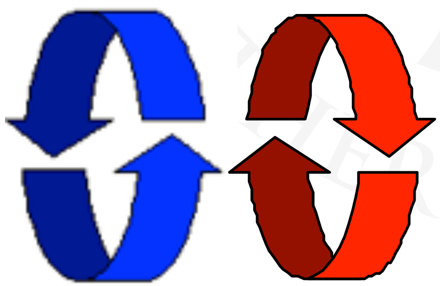
UNIVERSITY OF SOUTHERN DENMARK

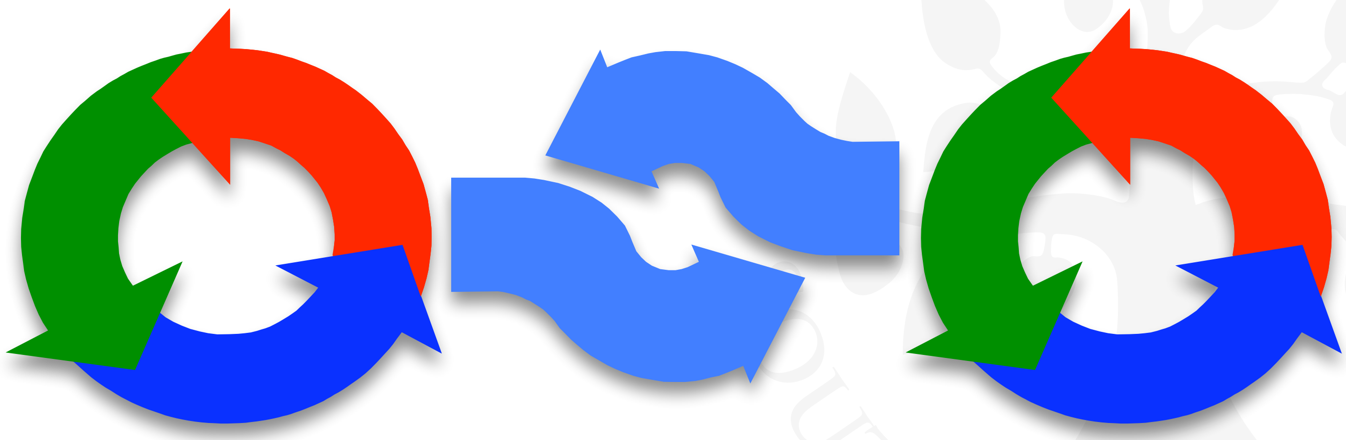
Safety & Liveness Properties





UNIVERSITY OF SOUTHERN DENMARK

Repetition: Deadlock



Concepts, Models, And Practice

- Concepts
  - deadlock (no further progress)
  - 4x necessary & sufficient conditions
- Models
  - no eligible actions (analysis gives shortest path trace)
- Practice
  - blocked threads

Aim - deadlock avoidance:

"Break at least one of the deadlock conditions".

Deadlock: 4 Necessary And Sufficient Conditions

1. **Mutual exclusion cond.** (aka. "Serially reusable resources"):

   the processes involved share resources which they use under mutual exclusion.

2. **Hold-and-wait condition** (aka. "Incremental acquisition"):

   processes hold on to resources already allocated to them while waiting to acquire additional resources.

3. **No pre-emption condition:**

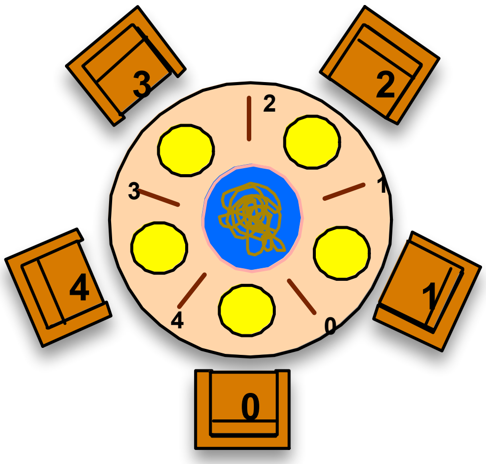
   once acquired by a process, resources cannot be "pre-empted" (forcibly withdrawn) but are only released voluntarily.

4. **Circular-wait condition** (aka. "Wait-for cycle"):

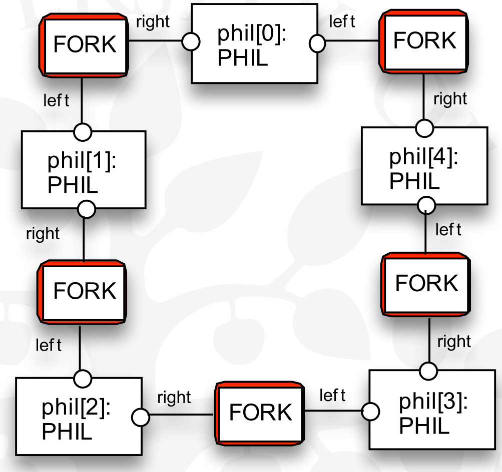
   a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Dining Philosophers (Concepts, Models And Practice)

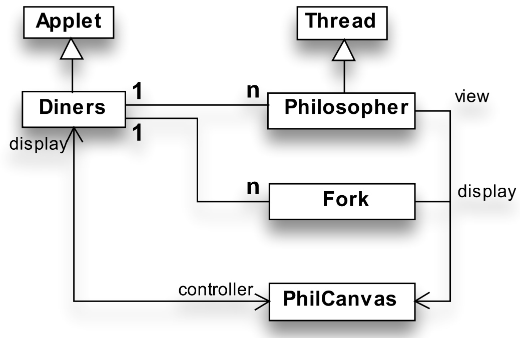
- Concepts

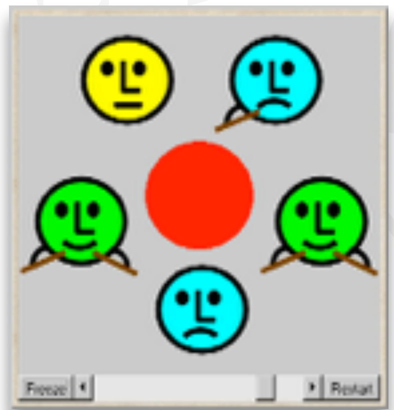


- Models

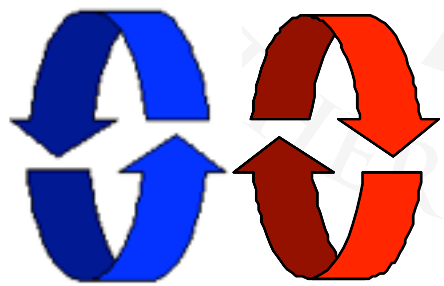


- Practice





Safety & Liveness Properties



Safety & Liveness Properties

Concepts:

- Properties: true for every possible execution
- Safety: nothing bad ever happens
- Liveness: something good eventually happens

Models:

- Safety: no reachable ERROR/STOP state
- Progress: an action is eventually executed (fair choice and action priority)

Practice:

Threads and monitors

Aim: property satisfaction.

Agenda

Part I / III

- Safety

Part II / III

- Liveness

Part III / III

- Example: Reader/Writer

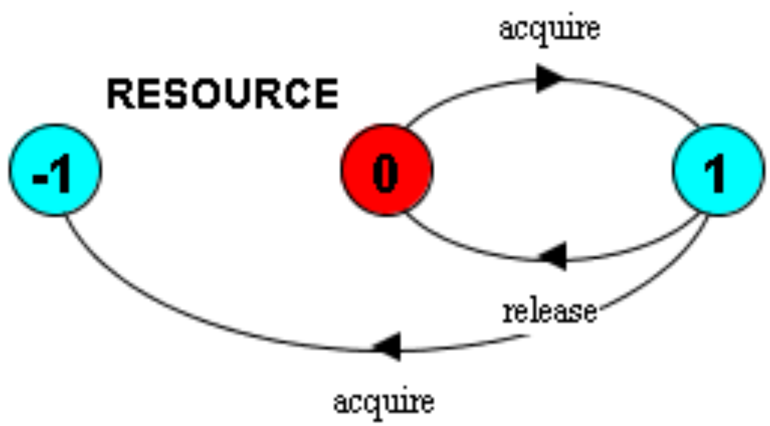
Safety

Part I / III

7.1 Safety

A **safety property** asserts that nothing **bad** happens.

- **STOP** or deadlocked state (no outgoing transitions)
- **ERROR** process (-1) to detect erroneous behaviour




```
RESOURCE =(acquire -> ACQUIRED),
ACQUIRED =(release -> RESOURCE
          |acquire -> ERROR).
```

- Analysis using LTSA:

  (shortest trace)

```
Trace to property violation in RESOURCE:
    acquire
    acquire
```

Stop Vs. Error

STOP:

```
P = (p->P | stop->STOP).
Q = (q->Q).

||SYSv1 = (P || Q).
```

Trace:

```
p
q
p
stop
q
q
...
```

LTSA:> **No deadlocks detected**

ERROR:

```
P = (p->P | error->ERROR).
Q = (q->Q).

||SYSv2 = (P || Q).
```

Trace:

```
p
q
p
error
```

SYSTEM DEADLOCKED

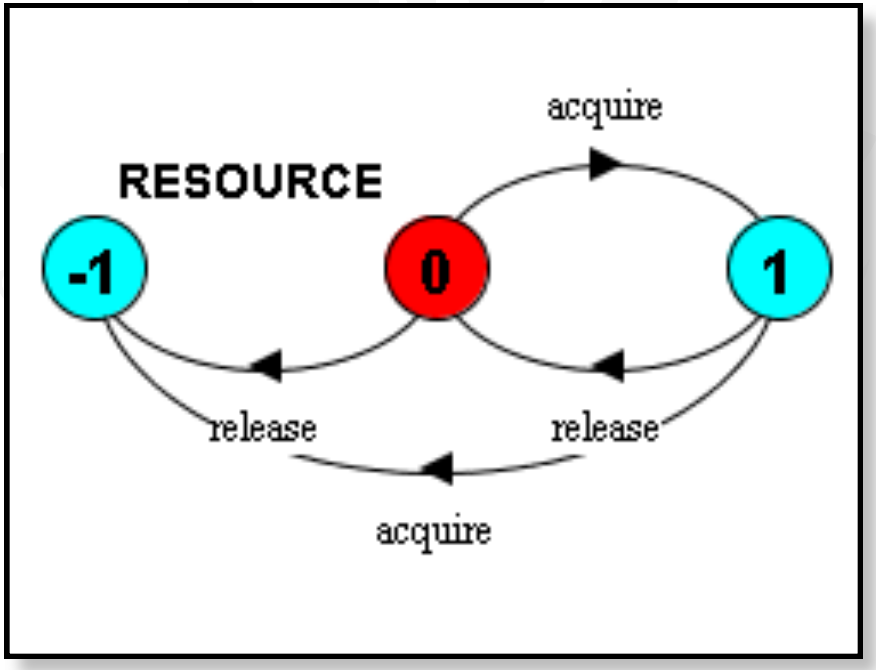
LTSA:> **Trace to property violation in P: error**

Safety - Property Specification

- ERROR conditions state what is **not** required (~ exceptions).
- In complex systems, it is usually better to specify **safety properties** by stating directly what **is** required.

```
property SAFE_RESOURCE =
    (acquire ->
        release ->
            SAFE_RESOURCE).
```

```
RESOURCE =
  (acquire ->
    (release -> RESOURCE
    |acquire -> ERROR)
  |release -> ERROR).
```



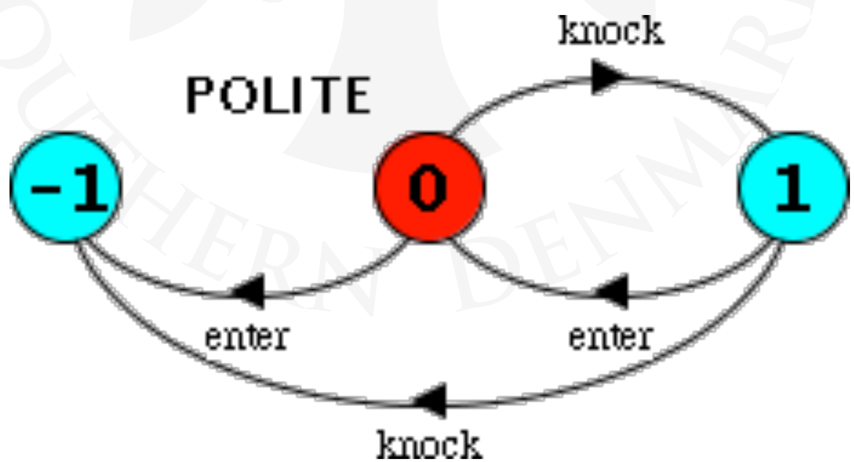
Safety Properties

Property that it is polite to knock before entering a room.

Traces:

```
knock->enter    ☺
enter           ☹
knock->knock    ☹
```

```
property POLITE
  = (knock -> enter -> POLITE).
```



Note: In **all** states, **all** the actions in the alphabet of a property are eligible choices.

Safety Properties

Safety `property P` defines a deterministic process that **asserts** that any trace including actions in the alphabet of `P`, is accepted by `P`.

Thus, if `S` is composed with `P`, then traces of actions in the alphabet $\alpha(S) \cap \alpha(P)$ must also be valid traces of `P`, otherwise **ERROR** is reachable.

Transparency of safety properties:

Since all actions in the alphabet of a property are eligible choices
=> composition with S does not affect its correct behaviour.

However, if a bad behaviour can occur (violating the safety property), then ***ERROR*** is reachable.

...and hence detectable through verification (using LTSA)!

Safety Properties

UNIVERSITY OF SOUTHERN DENMARK

- How can we specify that some action, **disaster**, never occurs?


```
NO_DISASTER = (disaster->ERROR).
```

...or...

```
property CALM = STOP + {disaster}.
```

A safety property must be specified so as to include all the acceptable, valid behaviours in its alphabet.

Models Vs. Properties: Implementation Vs. Specification

The model is for the implementation

The property is for the specification

- "The implementation is required to meet the specification"

Often:

- Operational model (**M**) ~ implementation
- **Declarative formula (ϕ)** ~ specification

```
∀t,t": acquire(t) ∧ acquire(t") ∧ t<t"  =>  ∃t': t<t'<t" ∧ release(t')
```

However, in FSP(/LTSA) both models and properties are described using the *same* language (namely FSP):

- Operational model: FSP process

```
property P = (acquire -> release -> P).
```

- **Operational** property: FSP property (process)

They will be similar (because they are using the same language), but they do not represent the same thing!

Safety - Mutual Exclusion


```
LOOP =
      (mutex.down->read->mod->write-> mutex.up->LOOP).

||SEMADEMO = (p[1..3]:LOOP ||
              {p[1..3]}::mutex:SEMAPHORE(1)).
```

How do we check that this does indeed ensure mutual exclusion in the critical section (**read/mod/write**)?

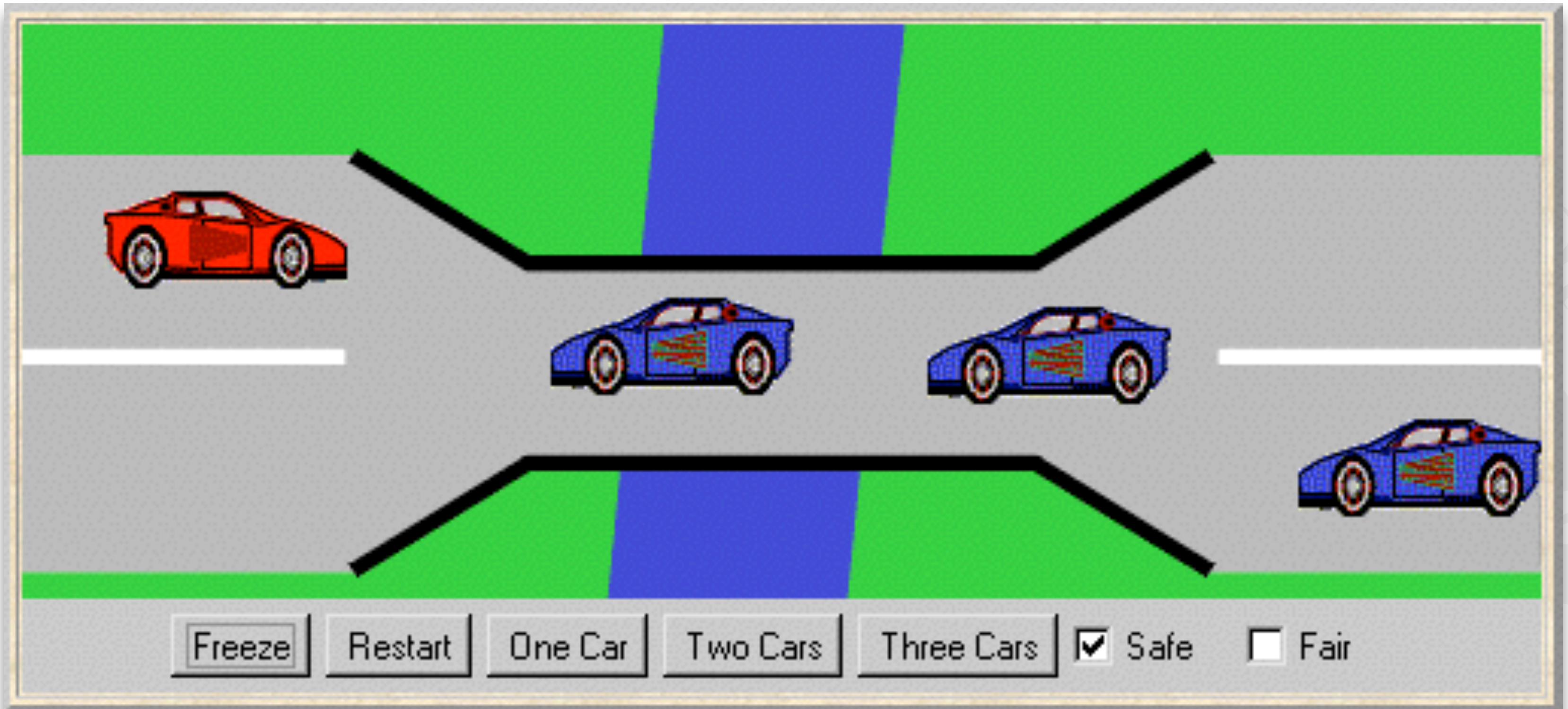
```
property MUTEX =
   (p[i:1..3].read -> p[i].write -> MUTEX).

||CHECK = (SEMADEMO || MUTEX).
```

Check safety using **LTSA**!

Is this safe with **SEMAPHORE(2)**?

$\forall t,t'': \mathbf{read}(t) \wedge \mathbf{read}(t'') \wedge t<t'' \Rightarrow \exists t': t<t'<t'' \wedge \mathbf{write}(t')$

7.2 Example: *Single Lane Bridge Problem*



A bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

Single Lane Bridge - Model

Using an appropriate **level of abstraction**!

- Events or actions of interest?

  enter and **exit** ~ Verbs

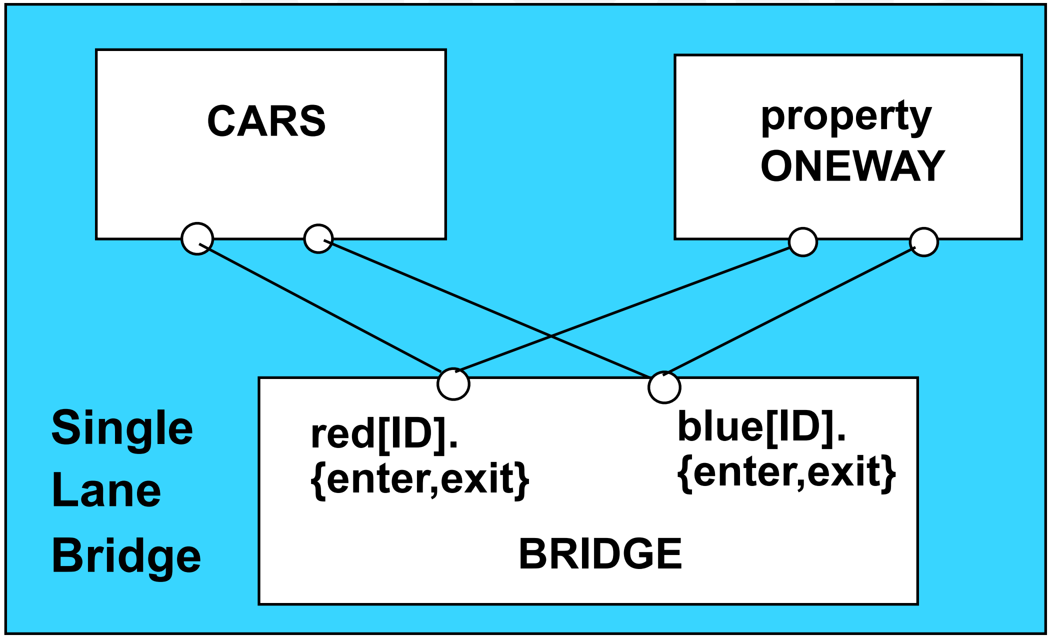
- Identify processes?

  car and **bridge** ~ Nouns

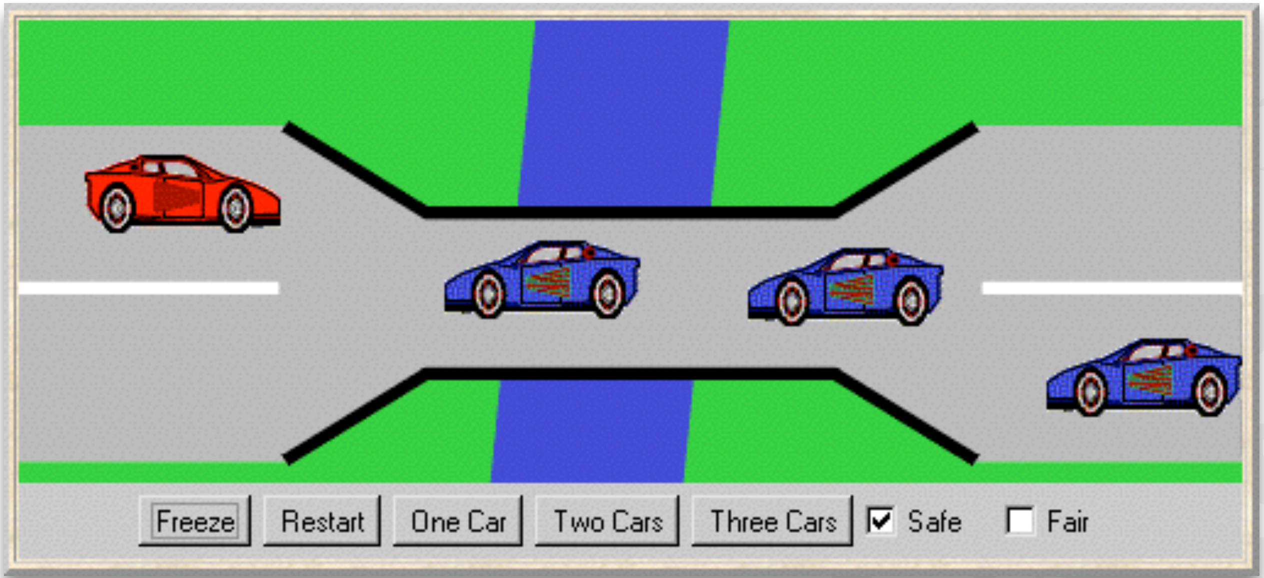
- Identify properties?

  "**oneway**" ~ Adjectives

Structure diagram:



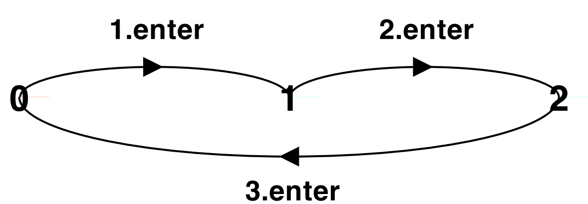
Single Lane Bridge - `Cars` Model

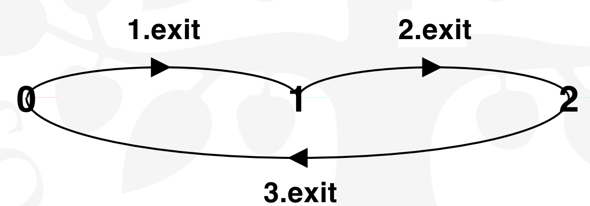



```
const N   = 3       // #cars (of each colour)
range ID = 1..N   // car identities

CAR = (enter->exit->CAR).   // car process
||N_CARS = ([ID]:CAR).      // N cars
```

Single Lane Bridge - Convoy Model

```
NOPASS_ENTER = SEQ[1],          // preserves entry order
SEQ[i:ID] = ([i].enter -> SEQ[i%N+1]).

NOPASS_EXIT  = SEQ[1],          // preserves exit order
SEQ[i:ID] = ([i].exit -> SEQ[i%N+1]).

||CONVOY = ([ID]:CAR || NOPASS_ENTER || NOPASS_EXIT).
```





Permits:

```
1.enter ; 1.exit ; 2.enter ; 2.exit
1.enter ; 2.enter ; 1.exit ; 2.exit
```

but not:

```
1.enter ; 2.enter ; 2.exit ; 1.exit
```

i.e. "no overtaking"

Single Lane Bridge - `Bridge` Model

Cars can move concurrently on bridge, but only as a oneway street (=> controller)!

How ; ideas?

The bridge maintains a count of blue and red cars on it.

Red cars are only allowed to enter when the blue count is 0

(and vice-versa).

```
range T = 0..N
```

```
BRIDGE = BRIDGE[0][0], // initially empty bridge
BRIDGE[nr:T][nb:T] =   // nr: #red; nb: #blue
   (when (nb==0) red[ID].enter -> BRIDGE[nr+1][nb]
   |             red[ID].exit  -> BRIDGE[nr-1][nb]
   |when (nr==0) blue[ID].enter-> BRIDGE[nr][nb+1]
   |             blue[ID].exit -> BRIDGE[nr][nb-1]
   ).
```

Single Lane Bridge - `Bridge` Model

```
Warning - BRIDGE.-1.0 defined to be ERROR
Warning - BRIDGE.0.-1 defined to be ERROR
Warning - BRIDGE.-1.1 defined to be ERROR
Warning - BRIDGE.-1.2 defined to be ERROR
Warning - BRIDGE.-1.3 defined to be ERROR
Warning - BRIDGE.0.4  defined to be ERROR
Warning - BRIDGE.1.-1 defined to be ERROR
Warning - BRIDGE.2.-1 defined to be ERROR
Warning - BRIDGE.4.0  defined to be ERROR
Warning - BRIDGE.3.-1 defined to be ERROR
Compiled: BRIDGE
```

"Sloppy controller":

Even when 0, `exit` actions permit the car counts to be decremented (i.e. unguarded exit actions) (similar with enter)

Recall that **LTSA** maps such undefined states to `ERROR`.

Is it a problem?

No, because cars are well-behaved (i.e. "they never `exit` before `enter`" and there are only three cars of each colour)

Single Lane Bridge - Safety Property "Oneway"

We now specify a **safety property** to check that cars only drive in one way at a time (i.e. no collisions occur)!:

```
property ONEWAY = EMPTY,
EMPTY =          (red[ID].enter  -> ONLY_RED[1]
                 |blue[ID].enter -> ONLY_BLUE[1]),

ONLY_RED[i:ID]  = (              red[ID].enter  -> RED[i+1]
                 |when (i==1) red[ID].exit   -> EMPTY
                 |when (i>1)  red[ID].exit   -> RED[i-1]),

ONLY_BLUE[j:ID] = (              blue[ID].enter -> BLUE[j+1]
                 |when (j==1) blue[ID].exit  -> EMPTY
                 |when (j>1)  blue[ID].exit  -> BLUE[j-1]).
```

When the bridge is empty, either a red or a blue car may enter.
While red cars are on the bridge only red cars can enter;
similarly for blue cars.

Model / Property: Implementation / Specification?

Model (~ implementation):

```
BRIDGE = BRIDGE[0][0], // initially empty bridge
BRIDGE[nr:T][nb:T] =   // nr: #red; nb: #blue
   (when  (nb==0) red[ID].enter   -> BRIDGE[nr+1][nb]
   |              red[ID].exit    -> BRIDGE[nr-1][nb]
   |when  (nr==0) blue[ID].enter -> BRIDGE[nr][nb+1]
   |              blue[ID].exit  -> BRIDGE[nr][nb-1]).
```

Property (~ specification):

```
property ONEWAY = EMPTY,
EMPTY =       (red[ID].enter  -> RED[1]
              |blue[ID].enter -> BLUE[1]),

RED[i:ID] = (            red[ID].enter  -> RED[i+1]
            |when (i==1) red[ID].exit   -> EMPTY
            |when (i>1)  red[ID].exit   -> RED[i-1]),

BLUE[j:ID]= (            blue[ID].enter -> BLUE[j+1]
            |when (j==1) blue[ID].exit  -> EMPTY
            |when (j>1)  blue[ID].exit  -> BLUE[j-1]).
```

Model / Property: Implementation / Specification?

UNIVERSITY OF SOUTHERN DENMARK

Controller model (~ implementation):

- Behaviour (which actions are permitted)

Property "observer" (~ specification):

- All *legal traces* over (often smaller) alphabet
- May be *many* properties checking different aspects of an impl.

Our controller meets its specification (i.e. "`no errors/deadlocks`").

- although "sloppy" (e.g. unguarded exits)

You cannot "cheat" here and use the controller as your specification (by prefixing it with `property`)

Single Lane Bridge - Model Analysis

A red and a blue convoy of N cars for each direction:

```
||CARS = (red:CONVOY || blue:CONVOY).
```

```
||SingleLaneBridge = (CARS||BRIDGE||ONEWAY).
```

Is the safety property "ONEWAY" violated?

```
No deadlocks/errors
```

...And **without** the BRIDGE (controller):

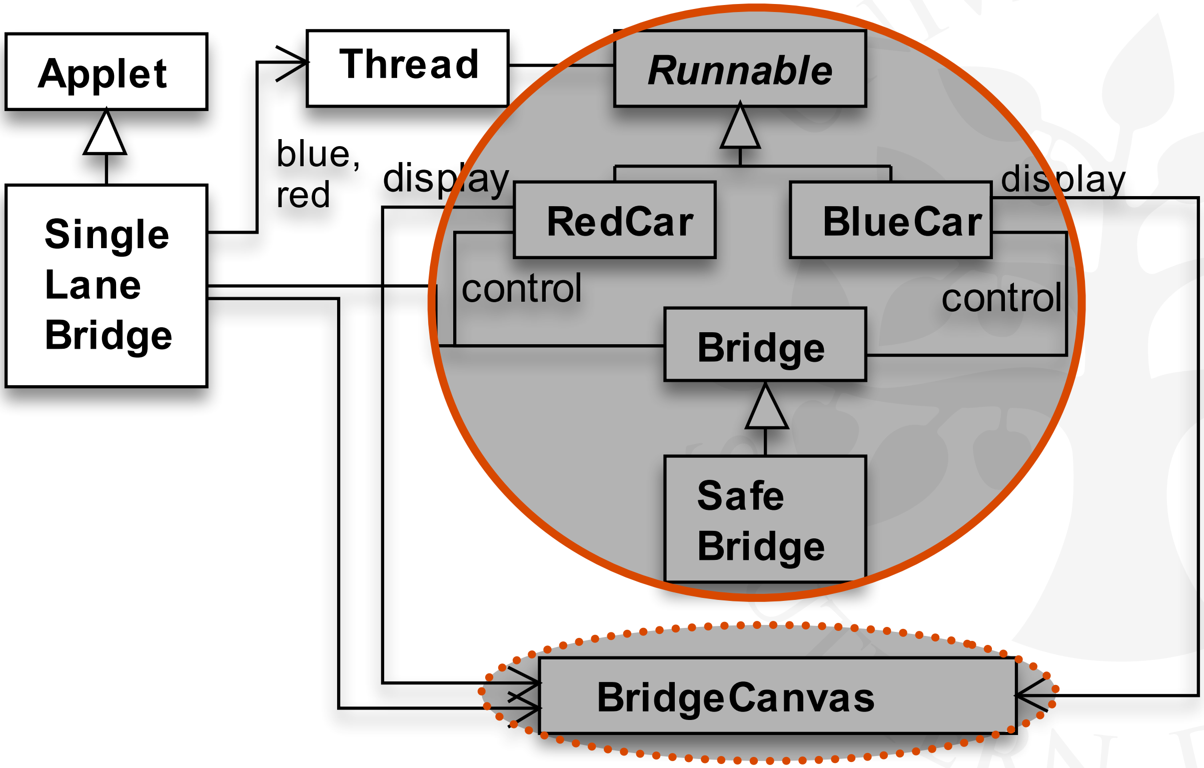
```
||SingleLaneBridge = (CARS||BRIDGE||ONEWAY).
```

Is the safety property "ONEWAY" violated?

```
Trace to property violation
                    in ONEWAY:
     red.1.enter
     blue.1.enter
```

Single Lane Bridge - Implementation In Java

CAR (active => thread) ; BRIDGE (passive => monitor)



BridgeCanvas enforces no overtaking (~ NOPASS_ENTER).

Single Lane Bridge - Bridgecanvas

An instance of **BridgeCanvas** class is created by the SingleLaneBridge applet.

```
class BridgeCanvas extends Canvas {
    public void init(int ncars) {…}  // set #cars

    public boolean moveRed(int i)   throws Int'Exc'{…}
    // moves red car #i a step  (if possible)
    // returns 'true' if on bridge

    public boolean moveBlue(int i) throws Int'Exc'{…}
    // moves blue car #i a step (if possible)
    // returns 'true' if on bridge
}
```

Each Car object is passed a reference to the BridgeCanvas.

Single Lane Bridge - Redcar

```
class RedCar implements Runnable {
    Bridge control; BridgeCanvas display; int id;

    RedCar(Bridge b, BridgeCanvas d, int i) {
        control = b; display = d; id = i;
    }

    public void run() {
        try {
            while (true) {
                while (!display.moveRed(id)) ; // not on br.
                control.redEnter(); // req access to br.
                while (display.moveRed(id)) ;  // move on br
                control.redExit(); // release access to br.
            }
        } catch (InterruptedException _) {}
    }
}
```

Similarly for the BlueCar...

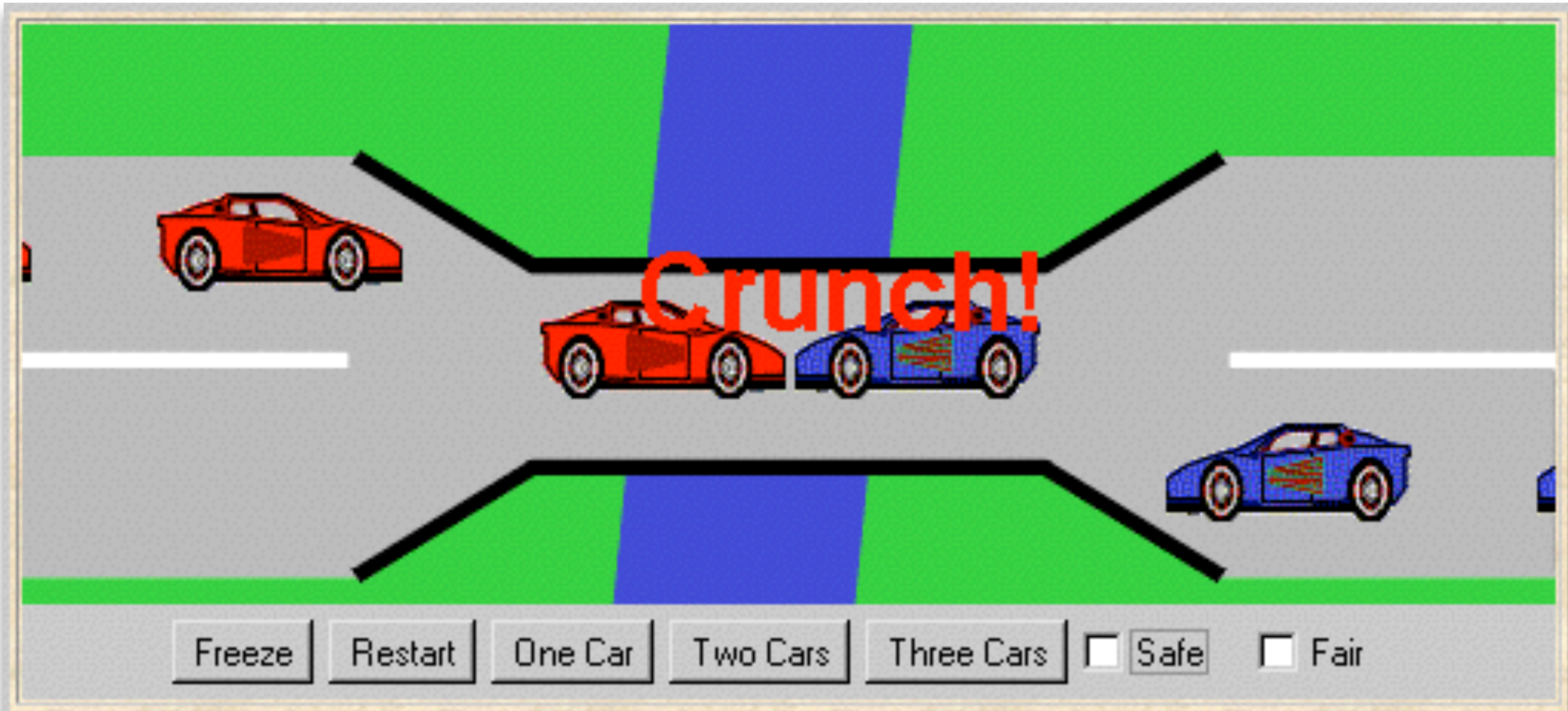
Single Lane Bridge - Class `Bridge`

```
class Bridge {
    synchronized void redEnter()  throws Int'Exc' {}
    synchronized void redExit()   {}
    synchronized void blueEnter() throws Int'Exc' {}
    synchronized void blueExit()  {}
}
```

Class `Bridge` provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

Result………… ?

Single Lane Bridge



8 people dead!

Single Lane Bridge - Safebridge

```
BRIDGE[nr:T][nb:T] =   // nr: #red; nb: #blue
… (when (nb==0) red[ID].enter  -> BRIDGE[nr+1][nb]
  |             red[ID].exit   -> BRIDGE[nr-1][nb]
```

```
class SafeBridge extends Bridge {
    protected int nred  = 0; // #red cars on br.
    protected int nblue = 0; // #blue cars on br.

    // monitor invariant: nred≥0 ∧ nblue≥0 ∧
    //                    ¬(nred>0 ∧ nblue>0)

    synchronized void redEnter() throws Int'Exc' {
        while (!(nblue==0)) wait();
        ++nred;
    }

    synchronized void redExit() {
        --nred;
         if (nred==0) notifyAll();
    }
}
```

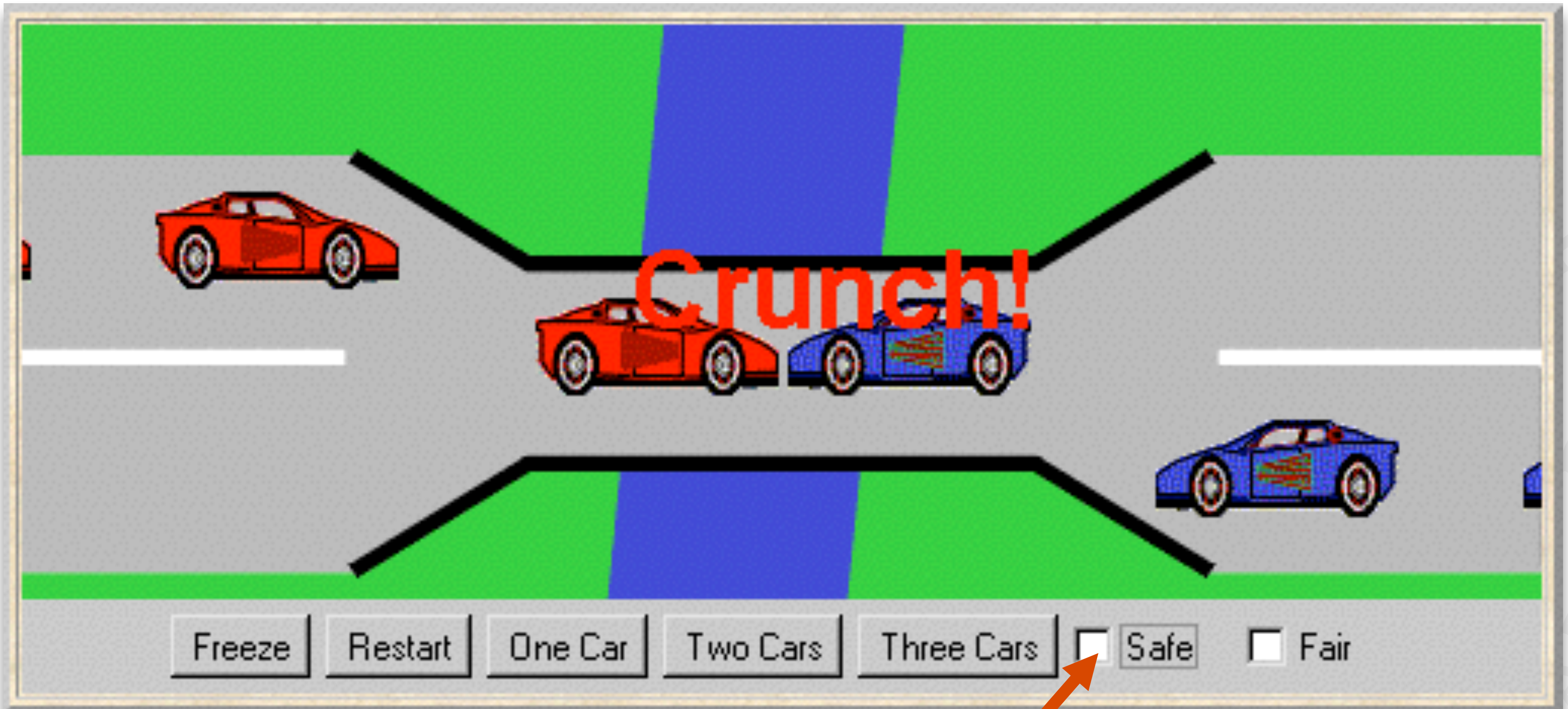
Single Lane Bridge – `Similarly For Blue`

```
synchronized void blueEnter() throws Int'Exc' {
    while (!(nred==0)) wait();
    ++nblue;
}

synchronized void blueExit() {
    --nblue;
    if (nblue==0) notifyAll();
}
```

To avoid (potentially) unnecessary thread switches, we use conditional notification to wake up waiting threads only when the number of cars on the bridge is zero (i.e., when the last car leaves the bridge).

But does every car **eventually** get an opportunity to cross the bridge...? This is a **liveness** property.

Single Lane Bridge



To ensure safety, the "safe" check box must be chosen in order to select the **SafeBridge** implementation.

Liveness

Part II / III

7.3 Liveness

A safety property **asserts** that nothing bad happens.

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

Does every car eventually get an opportunity to cross the bridge, i.e., make **progress**?

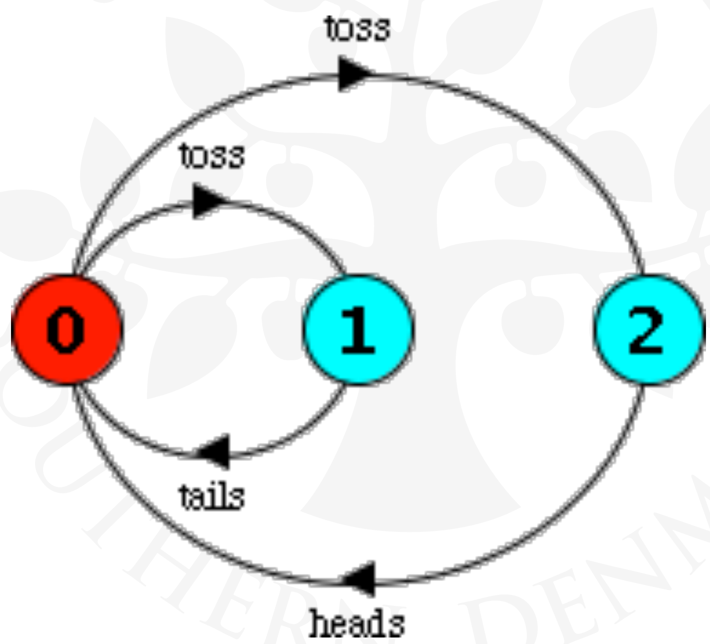
A progress property **asserts** that it is always the case that an action is eventually executed.

Progress is the opposite of starvation (= the name given to a concurrent programming situation in which an action is never executed).

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.

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



How about if we "choose":

toss(1) 100.000x; then
toss(2) 1x; then
toss(1) 100.000x; then
toss(2) 1x; then …

Fair?

Let's assume Fair Choice...

Progress Properties


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

This defines a **progress property**, `P`, which **asserts** that in an infinite execution, at least one of the actions

$a_1, a_2, \ldots, a_n$ will be executed infinitely often.

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

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

☺

☺

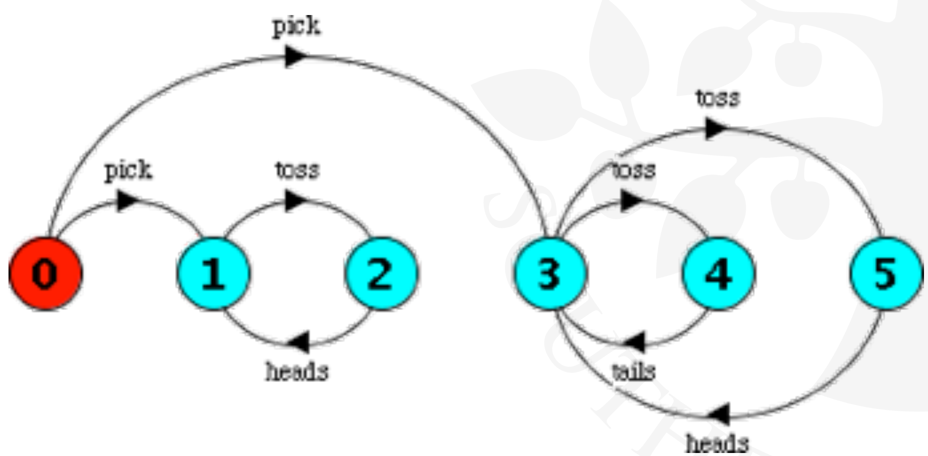
LTSA check progress:

```
No progress violations detected
```

Progress Properties

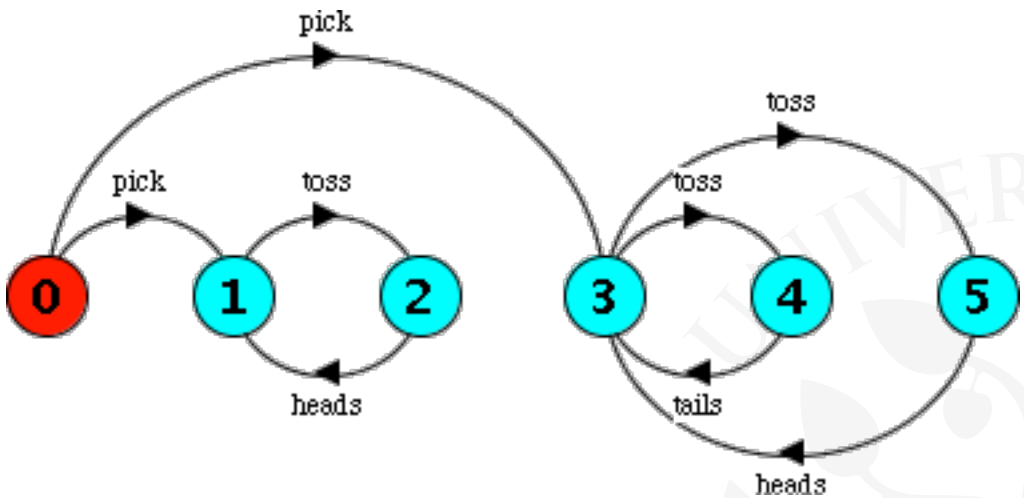
Suppose that there were **two** possible coins that could be picked up: a regular coin and a **trick** coin

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




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

Progress Properties




```
progress HEADS = {heads}

progress TAILS = {tails}
```

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

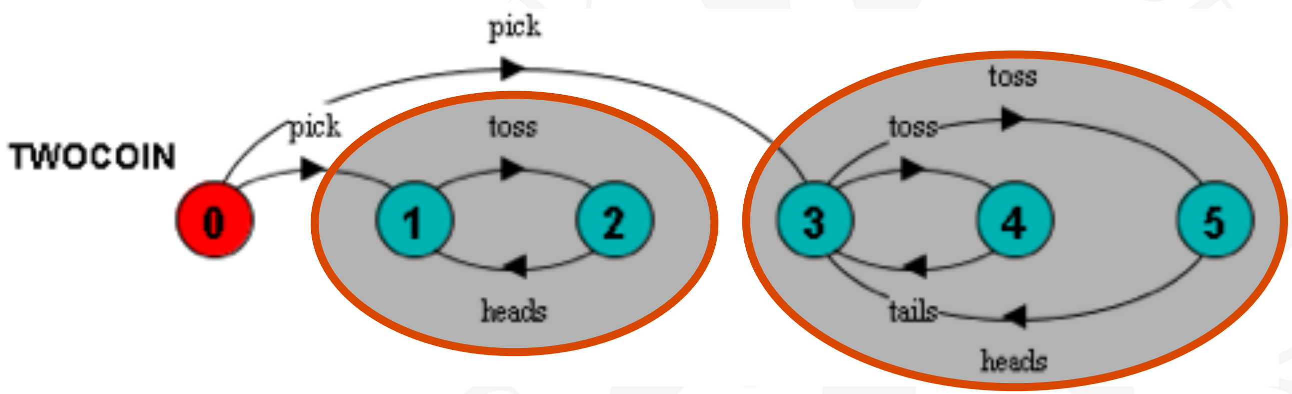
progress P = {heads,tails} ? ☺

Progress Analysis

A terminal set of states is one in which every state is reachable from every other state in the set via one or more transitions, and there is no transition from within the set to any state outside the set.

Terminal sets for TWOCOIN:

- {1,2} and
- {3,4,5}



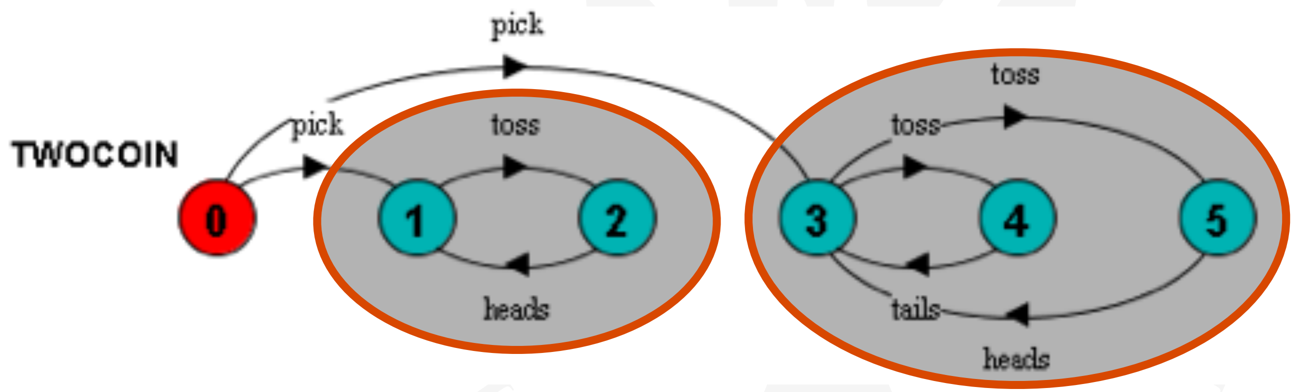
Given fair choice, each terminal set represents an execution in which each action used in a transition in the set is executed infinitely often.

Since there is no transition out of a terminal set, any action that is not used in the set cannot occur infinitely often in all executions of the system - and hence represents a potential progress violation!

Progress Analysis

A progress property is violated if analysis finds a terminal set of states in which **none** of the progress set actions appear.

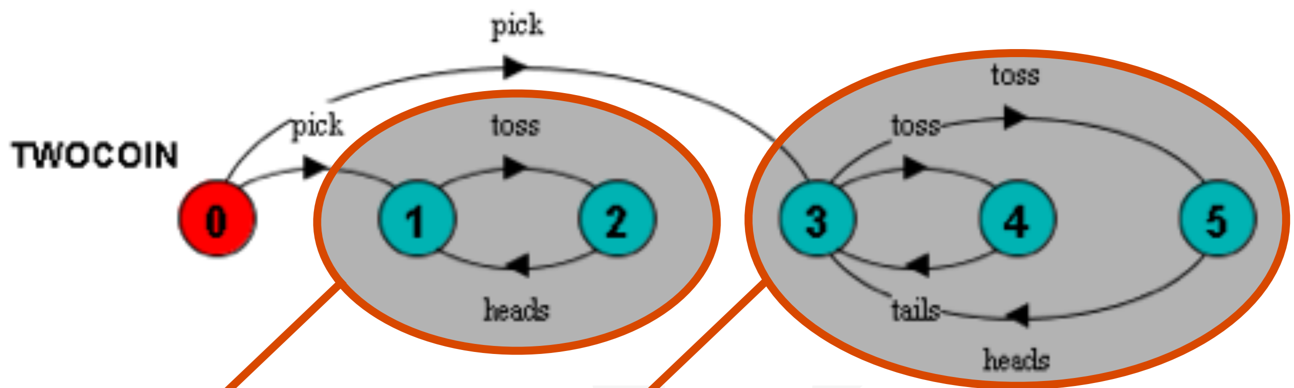
```
progress TAILS
  = {tails}
    in {1,2} ☹
```



Default progress: for every action in the alphabet, that action will be executed infinitely often. This is equivalent to specifying a separate progress property for every action.

Progress Analysis – Default Progress

Default progress:




```
Progress violation for actions:
      {pick, tails}
Path to terminal set of states:
      pick
Actions in terminal set:
      {toss, heads}
```

```
Progress violation for actions:
      {pick}
Path to terminal set of states:
      pick
Actions in terminal set:
      {toss, heads, tails}
```

Note: default holds => every other progress property holds (i.e., every action is executed infinitely often and the system consists of a single terminal set of states).

Progress - Action Priority

Action priority expressions describe scheduling properties:

High Priority ("<<")

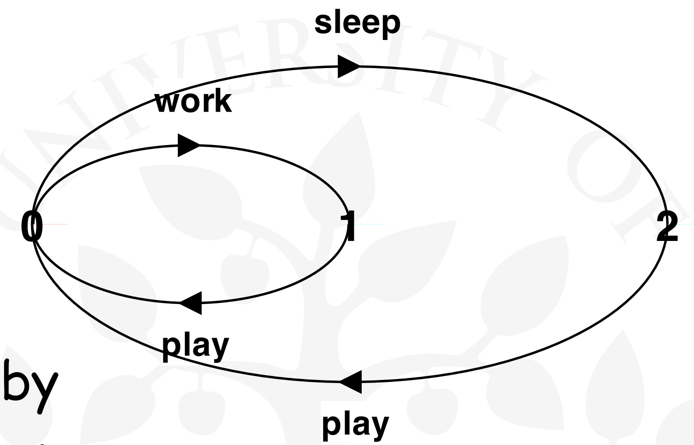
`||C = (P||Q)<<{a1,…,an}` specifies a composition in which the actions `a1,...,an` have higher priority than any other action in the alphabet of `P||Q` including the silent action `tau`. In any choice in this system which has one or more of the actions *`a1,...,an`* labelling a transition, the transitions labeled with lower priority actions are discarded.

Low Priority (">>")

`||C = (P||Q)>>{a1,…,an}` specifies a composition in which the actions `a1,...,an` have lower priority than any other action in the alphabet of `P||Q` including the silent action `tau`. In any choice in this system which has one or more transitions not labeled by *`a1,...,an`*, the transitions labeled by *`a1,...,an`* are discarded.

Progress - Action Priority Example

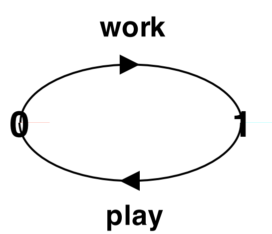
```
NORMAL =(work->play->NORMAL
        |sleep->play->NORMAL).
```

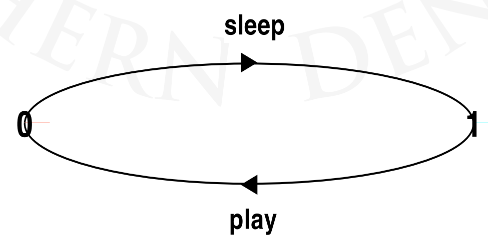


Action priority simplifies the resulting LTS by discarding lower priority actions from choices.

```
||HIGH =(NORMAL)<<{work}.
```

```
||LOW  =(NORMAL)>>{work}.
```





7.4 Congested Single Lane Bridge


```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

BLUECROSS - eventually one of the blue cars will be able to enter

REDCROSS - eventually one of the red cars will be able to enter

Congestion using action priority?

Could give red cars priority over blue (or vice versa) ?
In practice neither has priority over the other.

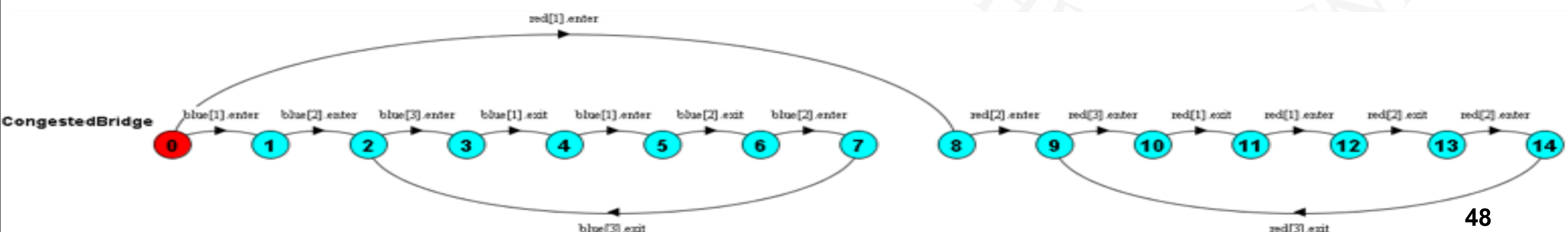
Instead we merely "encourage congestion" by lowering the priority of the exit actions of both cars from the bridge.

```
||CongestedBridge = (SingleLaneBridge)
                    >>{red[ID].exit,blue[ID].exit}.
```

Congested Single Lane Bridge Model

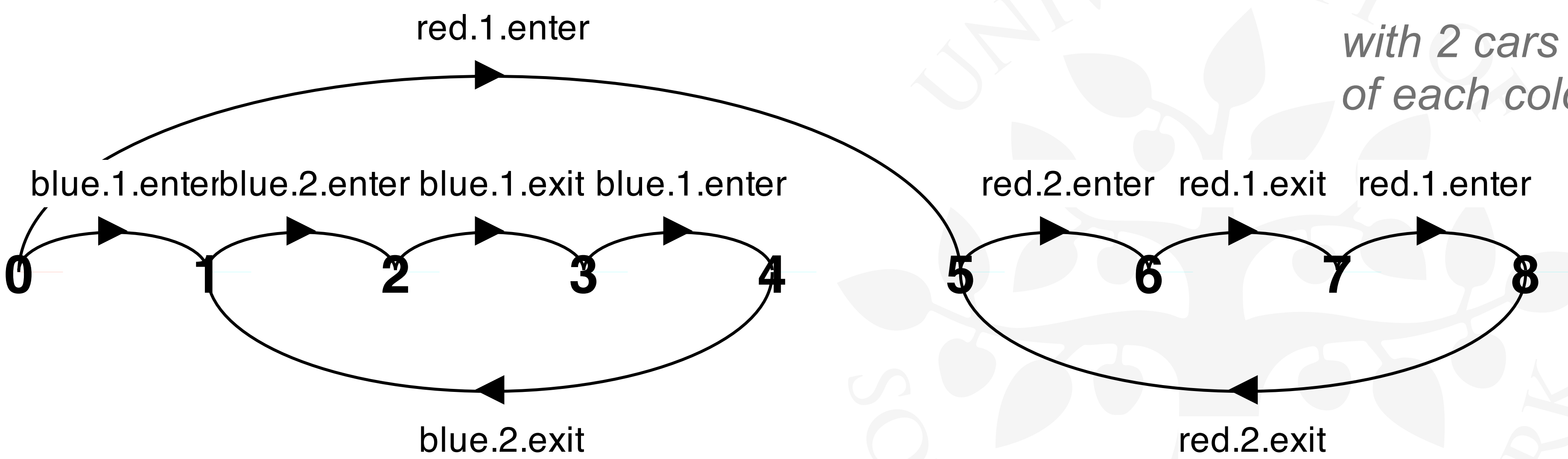
```
Progress violation: BLUECROSS
Path to terminal set of states:
     red.1.enter
     red.2.enter
Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter, red.
2.exit, red.3.enter, red.3.exit}

Progress violation: REDCROSS
Path to terminal set of states:
     blue.1.enter
     blue.2.enter
Actions in terminal set:
{blue.1.enter, blue.1.exit, blue.2.enter, blue.
2.exit, blue.3.enter, blue.3.exit}
```

This corresponds with the observation that, with **more than one car**, it is possible that whichever colour car enters the bridge first will continuously occupy the bridge preventing the other colour from ever crossing.



Congested Single Lane Bridge Model

```
||CongestedBridge = (SingleLaneBridge)
                     >>{red[ID].exit,blue[ID].exit}.
```



with 2 cars of each colour

Will the results be the same if we model congestion by giving car **entry** to the bridge **high** priority?

Can congestion occur if there is only one car moving in each direction?

Progress - Revised Single Lane Bridge Model

The bridge needs to know whether or not cars are **waiting** to cross.

Modify **CAR**:

```
CAR = (request -> enter -> exit -> CAR) .
```

The car "signals" bridge that it has arrived & wants to enter.

Modify **BRIDGE**:

> Red cars are only allowed to enter the bridge if there are no blue cars on the bridge **and** there are **no blue cars waiting** to enter the bridge.

> ...and vice-versa for blue cars.

Progress - Revised Single Lane Bridge Model

University of Southern Denmark

```
// nr: #red cars on br.;  wr: #red cars waiting to enter
// nb: #blue cars on br.; wb: #blue cars waiting to enter

BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] = (
      red[ID].request             -> BRIDGE[nr][nb][wr+1][wb]
     |when (nb==0 && wb==0)
              red[ID].enter       -> BRIDGE[nr+1][nb][wr-1][wb]
     |red[ID].exit                -> BRIDGE[nr-1][nb][wr][wb]
     |blue[ID].request            -> BRIDGE[nr][nb][wr][wb+1]
     |when (nr==0 && wr==0)
              blue[ID].enter      -> BRIDGE[nr][nb+1][wr][wb-1]
     |blue[ID].exit               -> BRIDGE[nr][nb-1][wr][wb]
).
```

OK now?

```
CAR = (request -> enter -> exit -> CAR).
```

Progress - Analysis Of Revised Single Lane Bridge Model


```
Trace to DEADLOCK:
    red.1.request
    red.2.request
    red.3.request
    blue.1.request
    blue.2.request
    blue.3.request
```

The trace is the scenario in which there are cars waiting at both ends, and consequently, the bridge does not allow either red or blue cars to enter.

Solution?

Acquire resources in the same global order! But how?

This takes the form of a boolean variable (`bt`) which breaks the deadlock by indicating whether it is the turn of blue cars or red cars to enter the bridge.

Arbitrarily initialise `bt` to true initially giving blue initial precedence.

Revised Single Lane Bridge Implementation - Fairbridge


```
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] = (
    red[ID].request       -> BRIDGE[nr][nb][wr+1][wb][bt]
   |when (nb==0 && (wb==0||!bt))
          red[ID].enter   -> BRIDGE[nr+1][nb][wr-1][wb][bt]
   |red[ID].exit          -> BRIDGE[nr-1][nb][wr][wb][True]
```

```
class FairBridge extends Bridge {

    …

    synchronized void redExit(){
        --nred;
        blueturn = true;
        if (nred==0) notifyAll();
    }
}
```

Progress - 2Nd Revision Of Single Lane Bridge Model


```
const True = 1    const False = 0    range B = False..True

    //  bt: true   ~ blue  turn;
    //      false  ~ red   turn

BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] = (
     red[ID].request       -> BRIDGE[nr][nb][wr+1][wb][bt]
    |when (nb==0 && (wb==0||!bt))
          red[ID].enter    -> BRIDGE[nr+1][nb][wr-1][wb][bt]
    |red[ID].exit          -> BRIDGE[nr-1][nb][wr][wb][True]
    |blue[ID].request      -> BRIDGE[nr][nb][wr][wb+1][bt]
    |when (nr==0 && (wr==0||bt))
          blue[ID].enter   -> BRIDGE[nr][nb+1][wr][wb-1][bt]
    |blue[ID].exit         -> BRIDGE[nr][nb-1][wr][wb][False]
).
```

Analysis ?

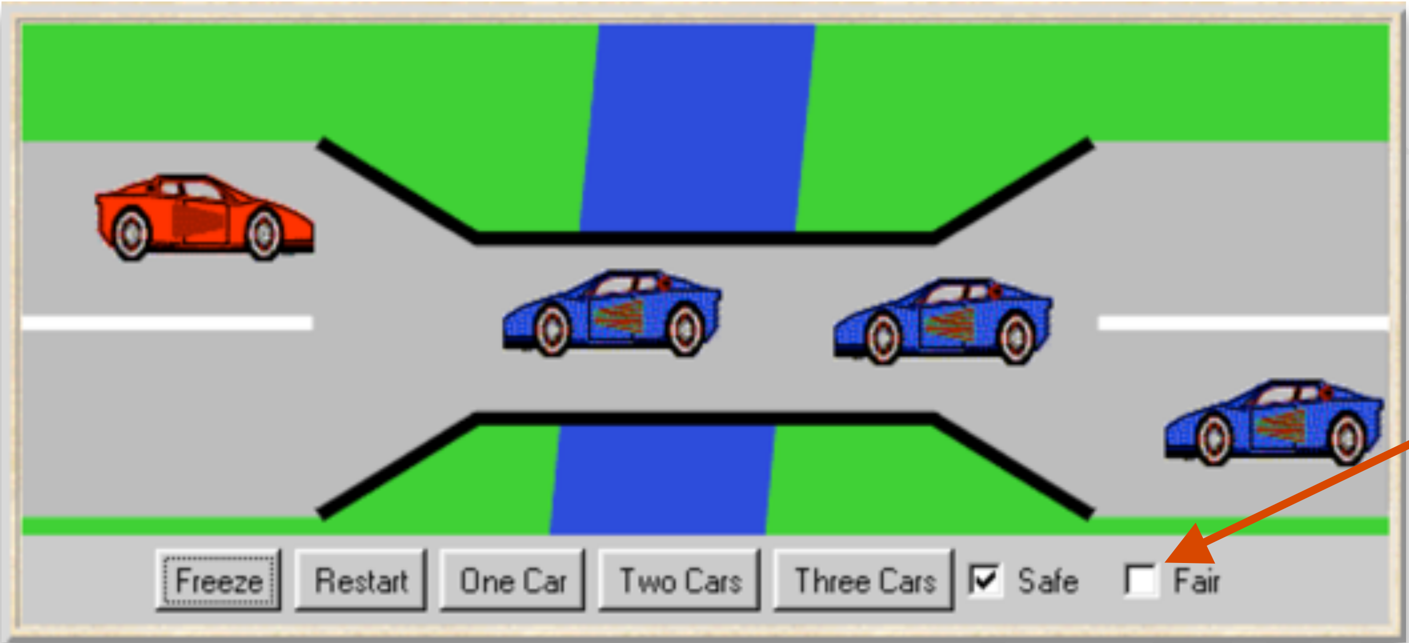
No progress violations detected. ☺

Revised Single Lane Bridge Implementation - Fairbridge


```
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] = (
    red[ID].request        -> BRIDGE[nr][nb][wr+1][wb][bt]
   |when (nb==0 && (wb==0||!bt))
         red[ID].enter     -> BRIDGE[nr+1][nb][wr-1][wb][bt]
```

```
class FairBridge extends Bridge {
    protected int nred, nblue, wblue, wred;
    protected boolean blueturn = true;

    synchronized void redRequest() {
        ++wred;
    }

    synchronized void redEnter() throws Int'Exc' {
        while (!(nblue==0 && (waitblue==0 || !blueturn)))
            wait();
        --wred;
        ++nred;
    }
```

Revised Single Lane Bridge Implementation - Fairbridge



Use **FairBridge** monitor

Note: we do not need to introduce a new request monitor method. The existing enter methods can be modified to increment a wait count before testing whether or not the caller can access the bridge... [see next slide]

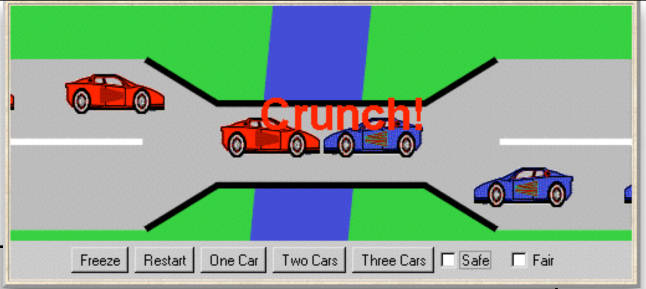
Implementation Short-Cut: Implicit “Request”

```
synchronized void redRequest() {
    ++wred;
}

synchronized void redEnter() throws Int'Exc' {
    while (!(nblue==0 && (waitblue==0 || !blueturn))) wait();
    --wred;
    ++nred;
}
```

...is equivalent to...: (for the problem at hand)

```
synchronized void redEnter() throws Int'Exc' {
    // request:
    ++wred;

    // enter:
    while (!(nblue==0 && (waitblue==0 || !blueturn))) wait();
    --wred;
    ++nred;
}
```

Repetition: Chapter 7
Safety & Liveness



A **safety** property **asserts** that nothing **bad** happens.

```
property ONEWAY = EMPTY,
EMPTY =      (red[ID].enter  -> RED[1]
             |blue[ID].enter -> BLUE[1]),

RED[i:ID] = (             red[ID].enter  -> RED[i+1]
            |when (i==1) red[ID].exit   -> EMPTY
            |when (i>1)  red[ID].exit   -> RED[i-1]),

BLUE[j:ID]= (             blue[ID].enter -> BLUE[j+1]
            |when (j==1) blue[ID].exit  -> EMPTY
            |when (j>1)  blue[ID].exit  -> BLUE[j-1]).
```

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

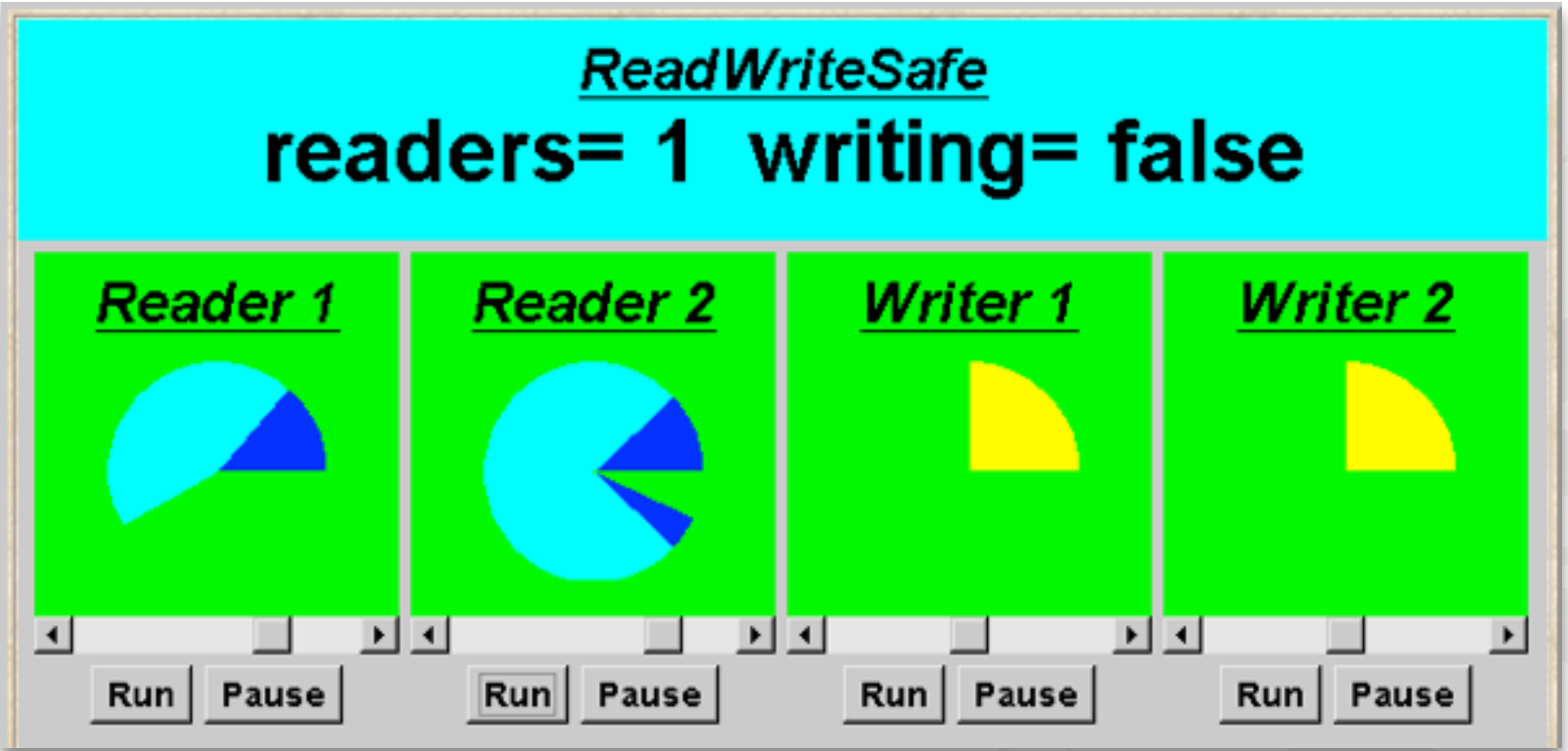
```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS =  {red[ID].enter}
```

Example: Readers/Writers

Part III / III

7.5 Readers And Writers

University of Southern Denmark



Light blue indicates database access.

A shared database is accessed by two kinds of processes. **Readers** execute transactions that examine the database while **Writers** both examine and update the database. A Writer must have exclusive access to the database; any number of Readers may concurrently access it.

Readers And Writers Model

- Events or actions of interest?

  acquireRead, releaseRead, acquireWrite, releaseWrite

- Identify processes.

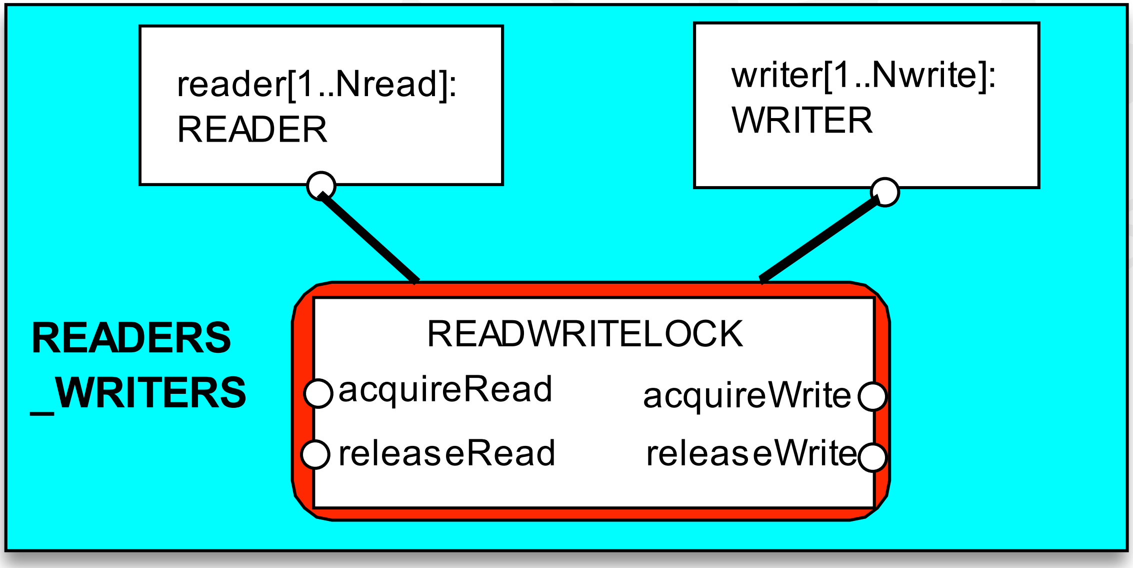
  Readers, Writers & the RW_Lock

- Identify properties.

  RW_Safe

  RW_Progress

- Structure diagram:



Readers/Writers Model - Reader & Writer


```
READER = (acquireRead ->
                 examine ->
                        releaseRead ->
                               READER)   \ {examine}.

WRITER = (acquireWrite ->
                 modify ->
                        releaseWrite ->
                               WRITER)   \ {modify}.
```

Action hiding is used as actions **examine** and **modify** are not relevant for access synchronisation.

Readers/Writers Model - `Rw_Lock`

The lock maintains a count of the number of readers, and a boolean for the writers.

```
const Nread = 2    // #readers
const Nwrite= 2    // #writers

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] = (
    when (!writing)
                acquireRead  -> RW[readers+1][writing]
   |            releaseRead  -> RW[readers-1][writing]
   |when (readers==0 && !writing)
                acquireWrite -> RW[readers][True]
   |            releaseWrite -> RW[readers][False]
).
```

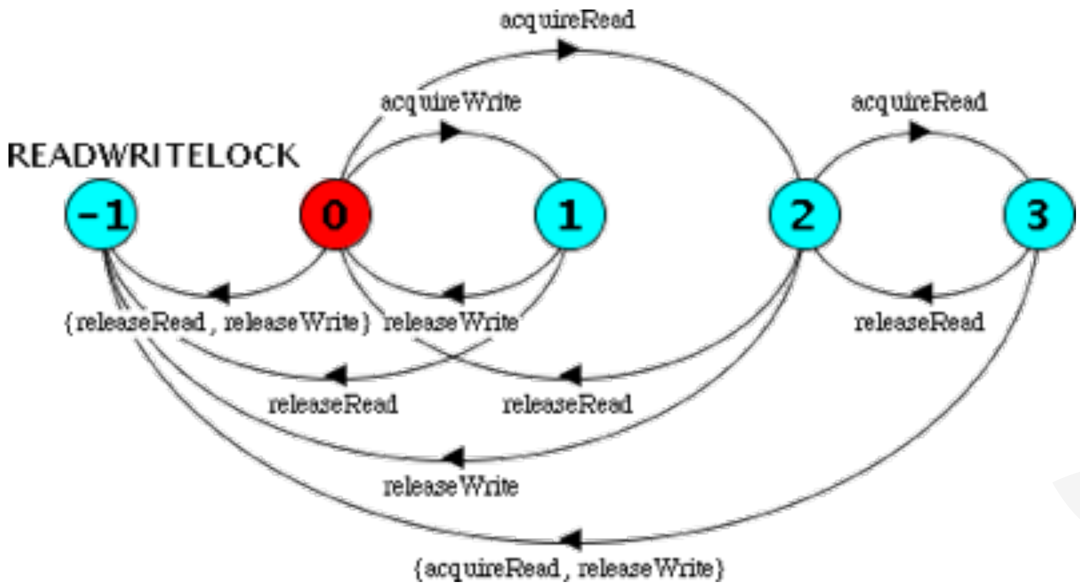
Readers/Writers Model - Safety

```
property SAFE_RW = NO_ONE,
NO_ONE = (acquireRead           -> ONLY_READERS[1]
         |acquireWrite          -> ONLY_WRITERS),

ONLY_READERS[i:1..Nread] =
     (acquireRead               -> ONLY_READERS[i+1]
     |when (i>1)  releaseRead   -> ONLY_READERS[i-1]
     |when (i==1) releaseRead   -> NO_ONE
     ),

ONLY_WRITERS = (releaseWrite -> NO_ONE).
```

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```

We can check that RW_LOCK satisfies the safety property……

Readers/Writers Model



We can now compose the **RW_LOCK** with **READER** and **WRITER** processes according to our structure…

```
||READERS_WRITERS
   = (reader[1..Nread]:READER
     || writer[1..Nwrite]:WRITER
     || {reader[1..Nread],
          writer[1..Nwrite]}::READWRITELOCK).
```

Safety and Progress Analysis ?

```
No deadlocks/errors. ☺
```

Readers/Writers Model - Progress


```
progress WRITE = {writer[1..Nwrite].acquireWrite}
progress READ  = {reader[1..Nread].acquireRead}
```

WRITE - eventually one of the writers will acquireWrite

READ – eventually one of the readers will acquireRead

```
No progress violations detected.
```
☺

Action priority (to "simulate intensive use")?

we lower the priority of the release actions for both readers and writers.

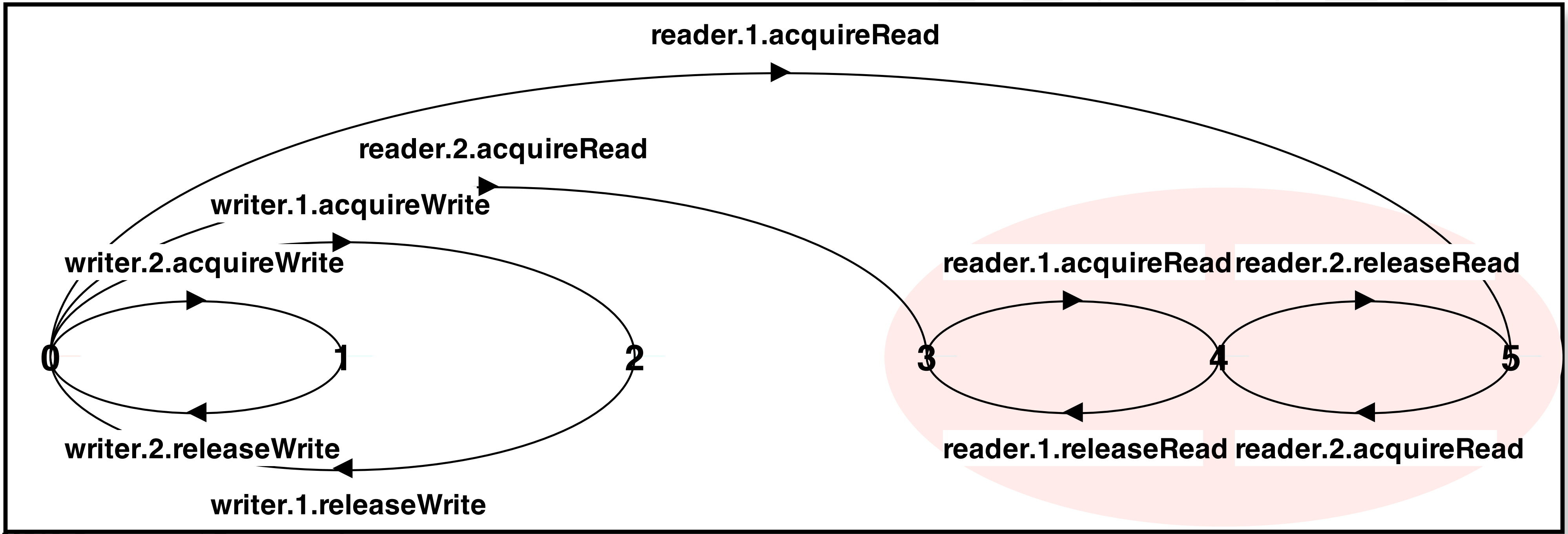
```
||RW_PROGRESS = READERS_WRITERS
               >>{reader[1..Nread].releaseRead,
                  writer[1..Nread].releaseWrite}.
```

Progress Analysis ? LTS?

Readers/Writers Model - Progress

```
Progress violation: WRITE
Path to terminal set of states:
     reader.1.acquireRead
Actions in terminal set:
{reader.1.acquireRead, reader.1.releaseRead,
 reader.2.acquireRead, reader.2.releaseRead}
```

Writer starvation: The number of readers never drops to zero.



Readers/Writers Implementation - Monitor Interface

We concentrate on the monitor implementation:

```
interface ReadWrite {
    void acquireRead()  throws Int'Exc';
    void releaseRead();
    void acquireWrite() throws Int'Exc';
    void releaseWrite();
}
```

We define an interface that identifies the monitor methods that must be implemented, and develop a number of alternative implementations of this interface.

Firstly, the **safe** **READWRITELOCK**.

Readers/Writers Implementation - Readwritesafe


```
class ReadWriteSafe implements ReadWrite {
    protected int readers = 0;
    protected boolean writing = false;

    synchronized void acquireRead() throws Int'Exc' {
        while (writing) wait();
        ++readers;
    }

    synchronized void releaseRead() {
        --readers;
        if(readers==0) notify();
    }
}
```

Unblock a single writer when no more readers.

```
when (!writing) acquireRead  -> RW[readers+1][writing]
|               releaseRead  -> RW[readers-1][writing]
```

Readers/Writers Implementation - Readwritesafe

UNIVERSITY OF SOUTHERN DENMARK

```
synchronized void acquireWrite() throws Int'Exc' {
    while (readers>0 || writing) wait();
    writing = true;
}

synchronized void releaseWrite() {
    writing = false;
    notifyAll();
}
```

Unblock **all** readers (and maybe other writers)

However, this monitor implementation suffers from the WRITE progress problem: possible **writer starvation** if the number of readers never drops to zero.

⇒ **Solution?**

```
|when (readers==0 && !writing) acquireWrite -> RW[readers][True]
|                              releaseWrite -> RW[readers][False]
```

Readers/Writers - Writer Priority

Strategy: **Block** readers if there is a writer waiting.

```
WRITER = (requestWrite ->
             acquireWrite ->
               modify ->
                 releaseWrite -> WRITER)   \{modify}.
```

Readers/Writers Model - Writer Priority

```
RW_LOCK = RW[0][False][0],
RW[readers:0..Nread][writing:Bool][waitingW:0..Nwrite] = (

    when (!writing && waitingW==0)
                acquireRead -> RW[readers+1][writing][waitingW]
   |releaseRead             -> RW[readers-1][writing][waitingW]

   |when (readers==0 && !writing)
               acquireWrite -> RW[readers][True][waitingW-1]
   |releaseWrite            -> RW[readers][False][waitingW]
   |requestWrite            -> RW[readers][writing][waitingW+1]
).
```

```
|| RW_P = R_W >>{*.release*}.  // simulate Intensive usage
```

Safety and Progress Analysis ?

Readers/Writers Model - Writer Priority

property RW_SAFE:

```
No deadlocks/errors
```

progress READ and WRITE:

```
Progress violation: READ
Path to terminal set of states:
     writer.1.requestWrite
     writer.2.requestWrite
Actions in terminal set:
{writer.1.requestWrite, writer.1.acquireWrite,
 writer.1.releaseWrite, writer.2.requestWrite,
 writer.2.acquireWrite, writer.2.releaseWrite}
```

Reader starvation: if always a writer waiting.

In practice: this may be satisfactory as is usually more read access than write, and readers generally want the most up to date information.

Readers/Writers Implementation - Readwritepriority

```
class ReadWritePriority implements ReadWrite {
    protected int readers = 0;
    protected boolean writing = false;
    protected int waitingW = 0; // #waiting writers

    synchronized void acquireRead() throws Int'Exc' {
        while (writing || waitingW>0) wait();
         ++readers;
    }

    synchronized void releaseRead() {
        --readers;
        if (readers==0) notify();
    }
}
```

Readers/Writers Implementation - Readwritepriority

```
synchronized void acquireWrite() throws Int'Exc' {
    // request write:
    ++waitingW;
    // acquire write:
    while (readers>0 || writing) wait();
    --waitingW;
    writing = true;
}

synchronized void releaseWrite() {
    writing = false;
    notifyAll();
}
```

Both **READ** and **WRITE** progress properties can be satisfied by introducing a **turn** variable as in the Single Lane Bridge.

Summary

UNIVERSITY OF SOUTHERN DENMARK

- Concepts
  - **properties**: true for every possible execution
  - **safety**: nothing bad ever happens
  - **liveness**: something good eventually happens
- Models
  - **safety**: no reachable ERROR/STOP state

    compose safety properties at appropriate stages
  - **progress**: an action is eventually executed

    fair choice and action priority

    apply progress check on the final target system model
- Practice
  - threads and monitors

Aim: property satisfaction