CSCI 5828: Foundations of Software Engineering

Lecture 18-19: Safety and Liveness Properties

Slides created by Magee and Kramer for the Concurrency textbook
Chapter 7

Safety & Liveness Properties

Concurrency: safety & liveness properties
## safety & liveness properties

<table>
<thead>
<tr>
<th>Concepts:</th>
<th>properties: true for every possible execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>safety: nothing bad happens</td>
</tr>
<tr>
<td></td>
<td>liveness: something good <em>eventually</em> happens</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Models:</th>
<th>safety: no reachable ERROR/STOP state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>progress: an action is <em>eventually</em> executed</td>
</tr>
<tr>
<td></td>
<td>fair choice and action priority</td>
</tr>
</tbody>
</table>

| Practice:          | threads and monitors                          |

**Aim:** property satisfaction.
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

![Diagram showing state transitions]

-1 \(\rightarrow\) 0 \(\rightarrow\) 1

**ACTUATOR**
= (command\(\rightarrow\)ACTION),

**ACTION**
= (respond\(\rightarrow\)ACTUATOR | command\(\rightarrow\)ERROR).

♦ **analysis using LTSA:**
(shortest trace)

**Trace to ERROR:**

command

command
Safety - property specification

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

```
property SAFE_ACTUATOR = (command
   -> respond
   -> SAFE_ACTUATOR).
```

- Analysis using **LTSA** as before.
Safety properties

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

Traces:  \( \text{knock} \rightarrow \text{enter} \)  \( \checkmark \)  \( \text{enter} \)  \( \times \)  \( \text{knock} \rightarrow \text{knock} \)

property POLITE

\[ = \text{(knock->enter->POLITE)}. \]

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 $P$ is composed with $S$, then traces of actions in the alphabet of $S \cap$ alphabet of $P$ 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, composing a property with a set of processes does not affect their correct behavior. However, if a behavior can occur which violates the safety property, then ERROR is reachable. Properties must be deterministic to be transparent.

Concurrency: safety & liveness properties
Safety properties

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

\[
\text{property } \text{CALM} = \text{STOP} + \{\text{disaster}\}.
\]

A safety property must be specified so as to include all the acceptable, valid behaviors in its alphabet.
Safety - mutual exclusion

\[
\text{LOOP} = (\text{mutex.down} \rightarrow \text{enter} \rightarrow \text{exit} \rightarrow \text{mutex.up} \rightarrow \text{LOOP}).
\]
\[
||\text{SEMADEMO} = (p[1..3] : \text{LOOP} \quad || \{p[1..3]\} :: \text{mutex:SEMAPHORE(1)}).
\]

How do we check that this does indeed ensure mutual exclusion in the critical section?

\[
\text{property} \quad \text{MUTEX} = (p[i:1..3].\text{enter} \rightarrow p[i].\text{exit} \rightarrow \text{MUTEX}).
\]
\[
||\text{CHECK} = (\text{SEMADEMO} || \text{MUTEX}).
\]

Check safety using LTS.

What happens if semaphore is initialized to 2?
7.2 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

♦ Events or actions of interest?
  enter and exit

♦ Identify processes.
  cars and bridge

♦ Identify properties.
  oneway

♦ Define each process and interactions (structure).

Concurrency: safety & liveness properties

©Magee/Kramer 2nd Edition
Single Lane Bridge - CARS model

\[
\text{const } N = 3 \quad \text{ // number of each type of car}
\]

\[
\text{range } T = 0..N \quad \text{ // type of car count}
\]

\[
\text{range } ID= 1..N \quad \text{ // car identities}
\]

\[
\text{CAR} = (\text{enter} \rightarrow \text{exit} \rightarrow \text{CAR}).
\]

To model the fact that cars cannot pass each other on the bridge, we model a CONVOY of cars in the same direction. We will have a red and a blue convoy of up to N cars for each direction:

\[
||\text{CARS} = (\text{red}:\text{CONVOY} \mid \mid \text{blue}:\text{CONVOY}).
\]
Single Lane Bridge - CONVOY model

NOPASS1 = C[1], //preserves entry order
C[i:ID] = ([i].enter-> C[i%N+1]).
NOPASS2 = C[1], //preserves exit order
C[i:ID] = ([i].exit-> C[i%N+1]).

||CONVOY = ([ID]:CAR|NOPASS1|NOPASS2).

Permits 1.enter→ 2.enter→ 1.exit→ 2.exit
but not 1.enter→ 2.enter→ 2.exit→ 1.exit

ie. no overtaking.
Single Lane Bridge - BRIDGE model

Cars can move concurrently on the bridge only if in the same direction. The bridge maintains counts of blue and red cars on the bridge. Red cars are only allowed to enter when the blue count is zero and vice-versa.

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

Even when 0, exit actions permit the car counts to be decremented. LTSA maps these undefined states to ERROR.
Single Lane Bridge - safety property ONEWAY

We now specify a safety property to check that cars do not collide! While red cars are on the bridge only red cars can enter; similarly for blue cars. When the bridge is empty, either a red or a blue car may enter.

\[
\text{property ONEWAY } = (\text{red}[\text{ID}].\text{enter } \rightarrow \text{RED}[1] \\
| \text{blue.}[\text{ID}].\text{enter } \rightarrow \text{BLUE}[1] \\
),
\]

\[
\text{RED}[i:ID] = (\text{red}[ID].\text{enter } \rightarrow \text{RED}[i+1] \\
| \text{when}(i==1)\text{red}[ID].\text{exit } \rightarrow \text{ONeway} \\
| \text{when}(i>1) \text{red}[ID].\text{exit } \rightarrow \text{RED}[i-1] \\
), \quad // i \text{ is a count of red cars on the bridge}
\]

\[
\text{BLUE}[i:ID] = (\text{blue}[ID].\text{enter } \rightarrow \text{BLUE}[i+1] \\
| \text{when}(i==1)\text{blue}[ID].\text{exit } \rightarrow \text{ONeway} \\
| \text{when}(i>1)\text{blue}[ID].\text{exit } \rightarrow \text{BLUE}[i-1] \\
). \quad // i \text{ is a count of blue cars on the bridge}
\]
Single Lane Bridge - model analysis

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

Is the safety property **ONeway** violated?

No deadlocks/errors

||SingleLaneBridge = (CARS||ONeway).

Without the **BRIDGE** constraints, is the safety property **ONeway** violated?

Trace to property violation in **ONeway**
red.1.enter
blue.1.enter

Concurrency: safety & liveness properties
Single Lane Bridge - implementation in Java

Active entities (cars) are implemented as threads.
Passive entity (bridge) is implemented as a monitor.
BridgeCanvas enforces no overtaking.

Concurrency: safety & liveness properties
An instance of BridgeCanvas class is created by SingleLaneBridge applet - ref is passed to each newly created RedCar and BlueCar object.

```java
class BridgeCanvas extends Canvas {
    public void init(int ncars) {...} //set number of cars
    //move red car with the identity i a step
    //returns true for the period on bridge, from just before until just after
    public boolean moveRed(int i)
    throws InterruptedException{...}

    //move blue car with the identity i a step
    //returns true for the period on bridge, from just before until just after
    public boolean moveBlue(int i)
    throws InterruptedException{...}

    public synchronized void freeze(){...} // freeze display
    public synchronized void thaw(){...} //unfreeze display
}
```
class RedCar implements Runnable {
    BridgeCanvas display; Bridge control; int id;
    RedCar(Bridge b, BridgeCanvas d, int id) {
        display = d; this.id = id; control = b;
    }
    public void run() {
        try {
            while (true) {
                while (!display.moveRed(id));  // not on bridge
                control.redEnter();           // request access to bridge
                while (display.moveRed(id));  // move over bridge
                control.redExit();            // release access to bridge
            }
        } catch (InterruptedException e) {} 
    }
}

Similarly for the BlueCar

Concurrency: safety & liveness properties
Class **Bridge** provides a null implementation of the access methods i.e. no constraints on the access to the bridge.

**Result**........... ?
To ensure safety, the “safe” check box must be chosen in order to select the SafeBridge implementation.
class SafeBridge extends Bridge {
    private int nred = 0; // number of red cars on bridge
    private int nblue = 0; // number of blue cars on bridge

    // Monitor Invariant: nred≥0 and nblue≥0 and
    // not (nred>0 and nblue>0)

    synchronized void redEnter() throws InterruptedException {
        while (nblue>0) wait();
        ++nred;
    }

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

    This is a direct translation from the BRIDGE model.
To avoid 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.
7.3 Liveness

A safety property asserts that nothing bad happens.

A liveness property asserts that something good eventually happens.

Single Lane Bridge: Does every car eventually get an opportunity to cross the bridge?

ie. 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.

If a coin were tossed an infinite number of times, we would expect that heads would be chosen infinitely often and that tails would be chosen infinitely often.

This requires **Fair Choice**!

\[
\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} | \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN})
\]
Progress properties

progress \( P = \{a_1, a_2, \ldots, a_n\} \) defines a progress property \( P \) which asserts that in an infinite execution of a target system, at least one of the actions \( a_1, a_2, \ldots, a_n \) will be executed infinitely often.

**COIN system:**

- 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 trick coin
- and a regular coin

\[
\text{TWO COIN} = (\text{pick} \rightarrow \text{COIN} | \text{pick} \rightarrow \text{TRICK}), \\
\text{TRICK} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}), \\
\text{COIN} = (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} | \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN})
\]

\[
\text{TWO COIN:} \\
\text{progress HEADS} = \{\text{heads}\} \\
\text{progress TAILS} = \{\text{tails}\}
\]
Progress properties

\[
\text{progress HEADS} = \{\text{heads}\}
\]

\[
\text{progress TAILS} = \{\text{tails}\}
\]

**LTSA** check progress

Progress violation: TAILS
Path to terminal set of states:
{toss, heads}

\[
\text{progress HEADSorTAILS} = \{\text{heads, tails}\}
\]
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\} \text{ in } \{1,2\}

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

Default analysis for TWOCOIN?
Progress analysis

Default analysis for TWOCOIN: separate progress property for every action.

Progress violation for actions: {pick}
Path to terminal set of states:
  - pick

Actions in terminal set:
  - {toss, heads}

Progress violation for actions: {pick, tails}
Path to terminal set of states:
  - pick

Actions in terminal set:
  - {toss, heads}

If the default holds, then every other progress property holds i.e. every action is executed infinitely often and system consists of a single terminal set of states.
Progress - single lane bridge

The Single Lane Bridge implementation can permit progress violations. However, if default progress analysis is applied to the model then no violations are detected!

Why not?

Fair choice means that eventually every possible execution occurs, including those in which cars do not starve. To detect progress problems we must check under adverse conditions. We superimpose some scheduling policy for actions, which models the situation in which the bridge is congested.

progress BLUECROSS = {blue[ID].enter}
progress REDCROSS = {red[ID].enter}
No progress violations detected.
Progress - action priority

Action priority expressions describe scheduling properties:

| ||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 τ. In any choice in this system which has one or more of the actions a1,...,an labeling a transition, the transitions labeled with lower priority actions are discarded. |
|---|
| ||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 τ. 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

**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}.

Concurrency: safety & liveness properties
7.4 Congested single lane bridge

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

**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.*

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

→ *Progress Analysis? LTS?*
**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 color car enters the bridge first will continuously occupy the bridge preventing the other color from ever crossing.
congested single lane bridge model

\[
\begin{align*}
| | \text{CongestedBridge} &= (\text{SingleLaneBridge}) \\
&\implies \{\text{red[ID].exit, blue[ID].exit}\}.
\end{align*}
\]

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

Modify **CAR**:

\[
\text{CAR} = (\text{request} \rightarrow \text{enter} \rightarrow \text{exit} \rightarrow \text{CAR}).
\]

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.

**Blue** cars are only allowed to enter the bridge if there are no **red** cars on the bridge and there are no **red cars waiting** to enter the bridge.
Concurrency: safety & liveness properties

Progress - revised single lane bridge model

```c
/* nr – number of red cars on the bridge  wr – number of red cars waiting to enter
   nb – number of blue cars on the bridge  wb – number of 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?
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?**

Introduce some *asymmetry* in the problem (cf. Dining philosophers).

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 set* *bt* *to true initially giving* *blue* initial precedence.
const True = 1
const False = 0
range B = False..True

/* bt - true indicates blue turn, false indicates 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]
  ).
Revised single lane bridge implementation - FairBridge

```java
class FairBridge extends Bridge {
    private int nred = 0;   // count of red cars on the bridge
    private int nblue = 0;  // count of blue cars on the bridge
    private int waitblue = 0; // count of waiting blue cars
    private int waitred = 0; // count of waiting red cars
    private boolean blueturn = true;

    synchronized void redEnter() throws InterruptedException {
        ++waitred;
        while (nblue>0 || (waitblue>0 && blueturn)) wait();
        --waitred;
        ++nred;
    }

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

This is a direct translation from the model.
Revised single lane bridge implementation - FairBridge

```java
synchronized void blueEnter(){
    throws InterruptedException {
    ++waitblue;
    while (nred>0 || (waitred>0 && !blueturn)) wait();
    --waitblue;
    ++nblue;
}

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

Note that we did 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.

The “fair” check box must be chosen in order to select the FairBridge implementation.
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/writers model

♦ Events or actions of interest?
  acquireRead, releaseRead, acquireWrite, releaseWrite

♦ Identify processes.
  Readers, Writers & the RW_Lock

♦ Identify properties.
  RW_Safe
  RW_Progress

♦ Define each process and interactions (structure).
set Actions =
   {acquireRead,releaseRead,acquireWrite,releaseWrite}

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

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

**Alphabet extension** is used to ensure that the other access actions cannot occur freely for any prefixed instance of the process (as before).

**Action hiding** is used as actions examine and modify are not relevant for access synchronisation.
Concurrent readers/writers model - RW_LOCK

const True = 1
const False = 0
range Bool = False..True
const Nread = 2 // Maximum readers
const Nwrite = 2 // Maximum 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]
  ).

The lock maintains a count of the number of readers, and a Boolean for the writers.
readers/writers model - safety

property SAFE_RW

\[
\begin{align*}
\text{READ}[i:1..Nread] &= (\text{acquireRead} \rightarrow \text{READ}[i+1] \\
&\quad | \text{when}(i>1) \text{ releaseRead} \rightarrow \text{READ}[i-1] \\
&\quad | \text{when}(i==1) \text{ releaseRead} \rightarrow \text{SAFE_RW} \\
),
\end{align*}
\]

\[\text{WRITING} = (\text{releaseWrite} \rightarrow \text{SAFE_RW}).\]

We can check that RW_LOCK satisfies the safety property......

\[\|\| \text{READWRITELOCK} = (\text{RW_LOCK} \| \| \text{SAFE_RW}).\]

Safety Analysis? LTS?
readers/writers model

An **ERROR** occurs if a reader or writer is badly behaved (release before acquire or more than two readers).

We can now compose the **READWRITELOCK** with **READER** and **WRITER** processes according to our structure... ...

---

|READERS_WRITERS

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

---

Concurrency: safety & liveness properties
Readers/writers - progress

\[
\text{progress } \text{WRITE} = \{\text{writer}[1..N\text{write}].\text{acquireWrite}\} \\
\text{progress } \text{READ} = \{\text{reader}[1..N\text{read}].\text{acquireRead}\}
\]

- **WRITE** - eventually one of the **writers** will acquireWrite
- **READ** - eventually one of the **readers** will acquireRead

→ **Adverse conditions using action priority?**

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

\[
\| \| \text{RW\_PROGRESS} = \text{READERS\_WRITERS} \\
\quad >>\{\text{reader}[1..N\text{read}].\text{releaseRead}, \\
\quad \text{writer}[1..N\text{write}].\text{releaseWrite}\}.
\]

→ **Progress Analysis? LTS?**
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.

Try the Applet!
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.
Concurrent access to shared data can lead to both safety and liveness issues. To address these, the readers/writers implementation - **ReadWriteSafe** - is introduced.

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

  public synchronized void acquireRead() throws InterruptedException {
    while (writing) wait();
    ++readers;
  }

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

  Unblock a single writer when no more readers.
}
```

This implementation allows for multiple readers to acquire reads simultaneously, while ensuring that a single writer is blocked when no more readers are present. This helps in maintaining both safety and liveness properties in concurrent systems.
readers/writers implementation - ReadWriteSafe

```java
public synchronized void acquireWrite() throws InterruptedException {
    while (readers > 0 || writing) wait();
    writing = true;
}

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

Unblock all readers

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

Solution?
readers/writers - writer priority

**ReadWriteSafe**
 readers= 1  writing= false

Strategy:
Block readers if there is a writer waiting.

\[
\text{set Actions} = \{\text{acquireRead, releaseRead, acquireWrite, releaseWrite, requestWrite} \}
\]

\[
\text{WRITER} = (\text{requestWrite} \rightarrow \text{acquireWrite} \rightarrow \text{modify}
\rightarrow \text{releaseWrite} \rightarrow \text{WRITER}) + \text{Actions} \{\text{modify}\}.
\]
readers/writers model - writer priority

```plaintext
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]
).
```

→ 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}

In practice, this may be satisfactory as is usually more read access than write, and readers generally want the most up to date information.
class ReadWritePriority implements ReadWrite{
    private int readers = 0;
    private boolean writing = false;
    private int waitingW = 0; // no of waiting Writers.

    public synchronized void acquireRead() throws InterruptedException {
        while (writing || waitingW > 0) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if (readers == 0) notifyAll();
    }

    May also be readers waiting
readers/writers implementation - **ReadWritePriority**

```java
synchronized public void acquireWrite()
    throws InterruptedException {
    ++waitingW;
    while (readers>0 || writing) wait();
    --waitingW;
    writing = true;
}

synchronized public 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

◆ Concepts
  ● properties: true for every possible execution
  ● safety: nothing bad 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