Safety and Liveness

Abhik Roychoudhury
CS 3211
National University of Singapore

Reading material: Chapter 7 of Textbook.

Safety & liveness properties

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

Models:
- Safety: no reachable ERROR/STOP state
  - Progress: an action is eventually executed

Practice: threads and monitors

Aim: property satisfaction.

Programs, Properties

- Programs, Properties
- Program
- Property
- Model Checker (SPIN)
- Witness violation of property – violating execution.
- A desired property is supposed to hold true for all the concurrent program executions.

Liveness properties

- Something “good” will eventually happen.
- The most common liveness property in seq. programs
  - The program will eventually terminate!
- For concurrent programs, liveness properties can be of the form
  - Request for shared resources are eventually granted.
- So, what is new in today’s lecture’s discussion?
  - We have been discussing properties like mutual exclusion all along. Today’s discussion makes it more systematic and shows mutual excl. as a special case of a larger class of properties!

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

Trace to ERROR:
- command
- ACTUATOR
- ACTION
- respond

-1
0
1

command
Safety Property Specification

- Either, mark the violation of the safety property as “Error” states in the concurrent system S.
- Checking the safety property then amounts to checking that the “error” states are never reached.
- Instead, the safety property itself could be described as a process P (using our process equations).
- Checking for property violation amounts to finding a trace of S that does not satisfy P.

Checking the safety property then amounts to checking that the “error” states are never reached.

Instead, the safety property itself could be described as a process P (using our process equations).

Checking for property violation amounts to finding a trace of S that does not satisfy P.

Safety properties

Property: that it is polite to knock before entering a room.
Traces: knock→enter × enter × knock→knock

property POLITE = (knock→enter→POLITE).

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

Safety properties

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

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

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.

A violation of this property is a trace which will lead to -1. We have to check whether such a trace is possible in the concurrent system being checked.

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 ∩ 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.

Safety properties

Safety - mutual exclusion

property SAFE_ACTUATOR = (command
→ respond
→ SAFE_ACTUATOR).

A violation of this property is a trace which will lead to -1. We have to check whether such a trace is possible in the concurrent system being checked.

property MUXEX = (p[1..3].enter
→ p[i].exit
→ MUXEX).

CHECK = (SEMADEMO || MUXEX).
Safety - mutual exclusion

LOOP = (mutex.down -> enter -> exit
- > mutex.up -> LOOP).

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

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

property MUTEX = (p[i]:enter
- > p[i].exit
- > MUTEX).

CHECK = (SEMADEMO | | MUTEX).

What happens if semaphore is initialized to 2?

Violation of MUTEX
p.1.mutex.down,p.1.enter, p.2.mutex.down, p.2.enter

Semaphore - continued

Semaphore initialized to 2.

LOOP = (mutex.down -> enter -> exit
- > mutex.up -> LOOP).

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

SEMA(v) = (up -> SEMA[v+1]
| when (v > 0) down ->SEMA[v-1] )

Violation of MUTEX
p.1.mutex.down,p.1.enter,p.2.mutex.down,p.2.enter

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).

Single Lane Bridge - CARS model

const N = 3 // number of each type of car
range T = 0..N // type of car count
range ID= 1..N // car identities

CAR = (enter->exit->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:

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

Single Lane Bridge - CONVOY model

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

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] = \)
       \( (\text{when}(nb=0) \)
       \( \text{red[ID].enter \rightarrow BRIDGE}[nr+1][nb]) \)
       \( \text{when}(nr=0) \)
       \( \text{blue[ID].enter \rightarrow BRIDGE}[nr][nb+1]) \).

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.

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.

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

RED[i-ID] = (red[ID].enter \rightarrow RED[i+1] | when(i==1)red[ID].exit \rightarrow ONEWAY | when(i>1)red[ID].exit \rightarrow RED[i-1])

\( /i/ \) is a count of red cars on the bridge

BLUE[i-ID] = (blue[ID].enter \rightarrow BLUE[i+1] | when(i==1)blue[ID].exit \rightarrow ONEWAY | when(i>1)blue[ID].exit \rightarrow BLUE[i-1])

\( /i/ \) 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

SingleLaneBridge1 = (CARS | ONEWAY).

Without the BRIDGE constraints, is the safety property ONEWAY violated?

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

Single Lane Bridge - BridgeCanvas

An instance of BridgeCanvas class is created by SingleLaneBridge applet - ref is passed to each newly created RedCar and BlueCar object.

class BridgeCanvas extends Canvas {
  public void init(int ncars) {...} //set number of cars
  //move red car with the identity i a step
  public boolean moveRed(int i) throws InterruptedException{...}
  //move blue car with the identity i a step
  public boolean moveBlue(int i) throws InterruptedException{...}
  public synchronized void freeze(){...} // freeze display
  public synchronized void thaw(){...} //unfreeze display
}

Single Lane Bridge - implementation in Java

Single Lane Bridge - RedCar

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)) { // on bridge
          control.redEnter(); // request access to bridge
        }
        while (display.moveRed(id)) { // move over bridge
          //release access to bridge
        }
      }
    } catch (InterruptedException e) {} 
  }
}

Similarly for the BlueCar
Single Lane Bridge - class Bridge

```java
class Bridge {
    synchronized void redEnter() throws InterruptedException {}
    synchronized void redExit() {}
    synchronized void blueEnter() throws InterruptedException {}
    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 - SafeBridge

```java
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() {
        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? i.e. make PROGRESS?

A progress property asserts that it is always the case that a specific action is eventually executed. Progress is the opposite of starvation, the name given to a concurrent programming situation in which a specific 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.
Progress properties

\[ \text{progress } P = \{a_1, a_2, \ldots, a_n\} \]
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.

\[ \text{COIN system: } \]
\[ \text{progress HEADS} = \{\text{heads}\} \quad \checkmark \]
\[ \text{progress TAILS} = \{\text{tails}\} \quad ? \]

No progress violations detected (assuming fair choice).

Further explanation

TWOCOIN:

\[ \text{progress HEADS} = \{\text{heads}\} \quad \text{YES} \]
\[ \text{progress TAILS} = \{\text{tails}\} \quad \text{NO} \]

progress (heads)

case 1: if the trick coin is picked, only heads is executed

progress (tails)

case 2: if the normal coin is picked, heads is still executed infinitely often assuming fair choice being exerted on the coin toss.

Note that we consider both possibilities of trick coin or normal coin being picked --- this is a choice which is made only once, not infinitely many times.

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 — hence represents a potential progress violation.

Progress properties

Suppose that there were two possible coins that could be picked up:

- a trick coin
- and a regular coin....

TWOCOIN = (pick->COIN|pick->TRICK),

TRICK = (toss->heads->TRICK),

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

TWOCOIN:

\[ \text{progress HEADS} = \{\text{heads}\} \quad \checkmark \]
\[ \text{progress TAILS} = \{\text{tails}\} \quad ? \]

Progress violation: TAILS

Path to terminal set of states:

pick

Actions in terminal set:

\{toss, heads\}

progress HEADS or TAILS = \{heads, tails\} \quad ? \checkmark

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: 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 HEADS = \{heads\} in \{1,2\}

progress TAILS = \{tails\} in \{1,2\}

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, tails}

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.

Why no progress violation is detected?

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.

So, for every execution trace, there does not exist a progress violation. However, under certain scenarios, such as for a heavily congested bridge, there exists a violation of progress.

So, we need some mechanism to model "heavily congested bridge" by:
- prioritize car entry over car exit
- need some mechanism for prioritizing outgoing actions from a state.

Progress - action priority

Action priority expressions describe scheduling properties:

- \( C = (P \| Q) \ll \{a_1, \ldots, a_n\} \) specifies a composition in which the actions \( a_1, \ldots, a_n \) have higher priority than any other action in the alphabet of \( P \| Q \) including the silent action \( \tau \). In any choice in this system with one or more transitions labeled by \( a_1, \ldots, a_n \), the transitions labeled with lower priority actions are discarded.

- \( C = (P \| Q) \gg \{a_1, \ldots, a_n\} \) specifies a composition in which the actions \( a_1, \ldots, a_n \) 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 \( a_1, \ldots, a_n \), the transitions labeled by \( a_1, \ldots, a_n \) are discarded.

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?

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) \ll \{work\}.

LOW = (NORMAL) \gg \{work\}.

Progress - single lane bridge

progress BLUECROSS = \{blue[1D].enter\}
progress REDCROSS = \{red[1D].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) \gg \{red[1D].exit, blue[1D].exit\}.

Progress Analysis? LTS?
**Congested bridge model**

![Congested bridge model diagram](image)

**Congested single lane bridge model**

![Congested single lane bridge model diagram](image)

**Progress violation: BLUECROSS**

Path to terminal set of states:
- red.1.enter
- red.2.enter
- red.2.exit
- red.3.enter
- red.3.exit

Actions in terminal set:
- red.1.enter
- red.2.enter
- red.2.exit
- red.3.enter
- red.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.

**Progress violation: REDCROSS**

Path to terminal set of states:
- blue.1.enter
- blue.2.enter
- blue.2.exit
- blue.3.enter
- blue.3.exit

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.

**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).

**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.

**Progress - analysis of revised single lane bridge model**

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.

**Trace to DEADLOCK:**

- red.1.request
- red.2.request
- red.3.request
- blue.1.request
- blue.2.request
- blue.3.request

**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.

**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?**

**OK now?**
Progress - 2nd revision of single lane bridge model

```
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+1][nb][wr][wb-1][bt]
blue[ID].exit -> BRIDGE[nr][nb][wr][wb][False])
```

```
Analysis ?
```

Revised single lane bridge implementation - FairBridge

```
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 waitred = 0; // count of waiting red cars
private int waitblue = 0; // count of waiting blue 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.
```

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

```
Note that we did not need to introduce a new request method inside the
monitor representing the bridge. The existing enter methods can be
modified to increment a "waiting count" before testing whether or not
the caller can access the bridge.
```

Results from 2nd Revision of Bridge

- No deadlocks.
- Progress requirements met in presence of congestion
  - Both red and blue cars waiting to enter the single lane bridge.

7.5 Readers and Writers

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 - R/W Lock**

- **Actions**
  
  \( \text{READER} = (\text{acquireRead} \rightarrow \text{examine} \rightarrow \text{releaseRead} \rightarrow \text{READER}) + \text{Actions} \backslash (\text{examine}) \)

- **READER**
  - Acquire read
  - Examine
  - Release read
  - Reader

- **WRITER**
  - Acquire write
  - Modify
  - Release write
  - Writer

**Alphabet Extension**

- Alphabet extension is used to ensure that the other access actions are known to the process.

**Action Hiding**

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

**CS3211 2012-13 by Abhik55**

---

**Readers/Writers Model - Safety**

- **Property**
  
  \( \text{SAFE_RW} = (\text{acquireRead} \rightarrow \text{READING}[1]) \)

- **READING[i..Nread]**
  
  \( \text{acquireRead} \rightarrow \text{READING}[i+1] \)

- **WRITING**
  
  \( \text{releaseWrite} \rightarrow \text{SAFE_RW} \)

**We can check that R/W_LOCK satisfies the safety property...**

**READWRITELOCK**

- Reader\[1..Nread\] : R/W_LOCK | SAFE_RW.

**Safety Analysis? LTS?**

---

**Readers/Writers - Progress**

- **Progress WRITE**
  
  \( \text{acquireWrite} \)

- **Progress READ**
  
  \( \text{acquireRead} \)

**Adverse Conditions using Action Priority?**

- Lower the priority of the release actions for both readers and writers.

**RW_PROGRESS = READERS_WRITERS**

- Reader\[1..Nread\] : Reader
- Reader\[1..Nwrite\] : Writer

**Safety and Progress Analysis?**

---

**RECAP: Progress - action priority**

- **High Priority (<<)**
  
  \( C = (P \mid Q) << (a_1...a_n) \)
  
  Specifies a composition in which the actions \( a_1...a_n \) have higher priority than any other action in the alphabet of \( P \mid Q \) including the silent action \( \tau \). In any choice in this system which has one or more transitions not labeled by \( a_1...a_n \), the transitions labeled with lower priority actions are discarded.

- **Low Priority (>>)**
  
  \( C = (P \mid Q) >> (a_1...a_n) \)
  
  Specifies a composition in which the actions \( a_1...a_n \) have lower priority than any other action in the alphabet of \( P \mid Q \) including the silent action \( \tau \). In any choice in this system which has one or more transitions not labeled by \( a_1...a_n \), the transitions labeled by \( a_1...a_n \) are discarded.
Readers/Writers Model - Progress

Progress violation: WRITE
Path to terminal set of states:
- 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 {
    public void acquireRead()
        throws InterruptedException;
    public void releaseRead();
    public void acquireWrite()
        throws InterruptedException;
    public synchronized 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 {
    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();
    }
    public synchronized void acquireWrite()
        throws InterruptedException {
        while (readers>0 || writing) wait();
        writing = true;
    }
    public synchronized void releaseWrite() {
        writing = false;
        notifyAll();
    }
}
```

Unblock to a single writer when there are no more readers.

Readers/Writers - Writer Priority

Strategy: Block readers if there is a writer waiting.

```
set Actions = (acquireRead,releaseRead,acquireWrite, releaseWrite,requestWrite)
WRITER =\{requestWrite->acquireWrite->modify
    ->releaseWrite->WRITER
\}+Actions\{modify\}.
```

Solution?
Readers/Writers model - writer priority

```java
// 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][writing][waitingW+1]
```

Safety and Progress Analysis?

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

Readers/Writers implementation - ReadWritePriority

```java
class ReadWritePriority implements ReadWrite{
    private int readers =0;
    private boolean writing = false;
    private int waitingW = 0;
}
```

May also be readers waiting

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
  - progress: all safety properties at appropriate stages
- Practice:
  - threads and monitors

Follow-up questions

Abhik Roychoudhury
1. Question from post-it note

Why do we need properties if existing modeling techniques (those taught previously) can guarantee mutual exclusion?

Answer

Mutual exclusion is only one class of safety property. Deadlock is another popular class of safety properties. Safety properties are a general class of properties which state that certain "bad" events should never happen in the concurrent system being designed. Now, what is bad, and what is good - depends on the application in question. The no-deadlock property is a special kind of safety property which is always "bad" - irrespective of the application.

However, we have already seen simple examples where a property p may be a desired safety property in one application, but it may not need to be enforced in another application.

2. Question asked during lecture break

For the Promela modeling language is it possible for a condition to be put anywhere in the program?

Answer:

The answer is yes. Promela allows for the program to have statements like

\[ x > 0;\ y > 0;\ z > 0;\]

So, when the control reaches the statement \( x > 0 \) - the execution will check whether \( x \) is greater than 0. If \( x \) is greater than 0, the execution of this condition behaves like a skip statement. If \( x \) is not greater than 0, the execution will be blocked (this may be unblocked by another process modifying the value of \( x \), if \( x \) is a shared variable).

3. Another question asked

You discussed about starvation properties today. What if the scheduler for my concurrent program introduces starvation?

Answer

Once again, we must ensure that we do not confuse the levels of abstraction. A concurrent program is running with the help of an underlying scheduler. When we reason about progress/no-starvation properties the concurrent program, we are assuming an underlying "fair" scheduler - at least fair to the extent that it does not ignore one of the program threads forever. Now, exactly how this "fairness" is implemented - that is up to the systems software writer who will write the scheduler. As an application programmer, you want to be sure that your program will not run into starvation scenarios even when the scheduler is guaranteed to have "fair choice".