Safety & Liveness Properties

INF2140 Parallel Programming: Lecture 7 PART II

March 21, 2012

Plan

- repetition of main concepts
- repetition of **Single Lane Bridge example**
- final implementation of Single Lane Bridge example
- **Readers Writers example**
  - safety and progress
  - fairness
  - analysis and implementation

Repetition

Safety: "A safety property asserts that nothing bad happens."

- test if any path to ERROR/STOP (use LTSA)
- *property specification*
  - limit valid behaviors, by (explicit or implicit) transitions to ERROR
  - program like a normal process (but stop is considered ERROR)
  - compose with system like a normal process

  - **transparent** – may not add behavior
  - **deterministic**

Repetition

Liveness: "A liveness property asserts that something good eventually happens."

- **Progress:** guarantee that certain events will eventually happen.
- The progress property **progress P = {a1,a2..an}** asserts that in an infinite execution of the system, at least **one** of the actions `a1,a2..an` will be executed *infinitely often*.
- Example:

  `COIN` system: `progress HEADS = {heads}` ?
  `progress TAILS = {tails}` ?

  `progress TAILS = {heads,tails}` ?

  LTSA: `No progress violations detected.`

Repetition

Fairness

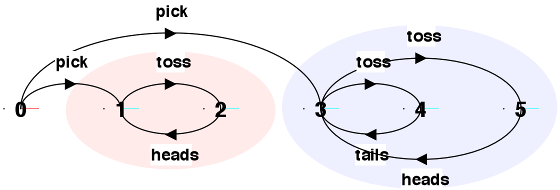
- **Fair Choice** (all possible actions chosen infinitely often)
  - **FSP** assumes underlying fair choice **(a1->P1 | a2->P2 | ...)**
  - **Java** has no underlying fairness (*notify need not be fair*)
- **Action priority** can express scheduling policy
  - high priority («)
  - low priority (»)

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 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!

Repetition: Single Lane Bridge

Model (FSP) first version

- CONVOY – no overtaking
- BRIDGE with counters **nr** (number of red cars on bridge) + **nb**
- property ONEWAY – to check against collision. OK!

Implementation

- direct implementation in Java
- conditional notifyAll to be efficient. Progress not violated?

Repetition: Single Lane Bridge: Fairness

```
progress BLUECROSS = {blue[ID].enter}
progress REDCROSS = {red[ID].enter}
// No progress violations detected.
```

- Progress in FSP assuming fair choice.
- Progress also without fair choice?

Model scheduling in FSP, to use in Java. Consider the problematic issues: congestion of cars. Will blue cars be able to enter? Red?

- **Revision 1:** use priorities and identify waiting lines for blue and red cars, using counters **wr** and **wb** respectively.
  - gives a deadlock (cars at both ends – no one can enter)
- **Revision 2:** introduce a switch to enforce change in direction: Boolean **bt** – blue cars have turn.

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]
// and similarly for blue cars:
  |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?

Revised single lane bridge implementation - FairBridge

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

```
  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();
  }
}
```

The "fair" check box must be chosen to select the `FairBridge` implementation.

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.

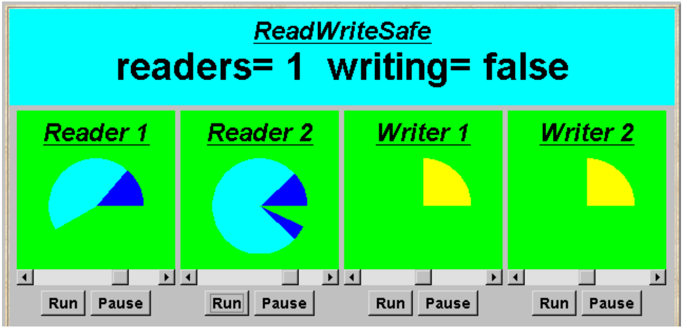
Summary: Single Lane Bridge

- **FSP model**
  - find deadlock
  - specify and check safety properties (ONEWAY)
  - revise model
  - specify and check progress properties (BLUECROSS, REDCROSS)
  - use priorities to reflect Java's (lack of) underlying progress control
- **Java implementation**
  - use FSP, almost direct translation to Java
  - use conditional notification (for better efficiency)
  - use FSP to re-analyse implementation choices

Aim: property satisfaction!

Classic example: Readers Writers

- look at deadlock
- safety
- liveness
- fairness
- analysis and implementation

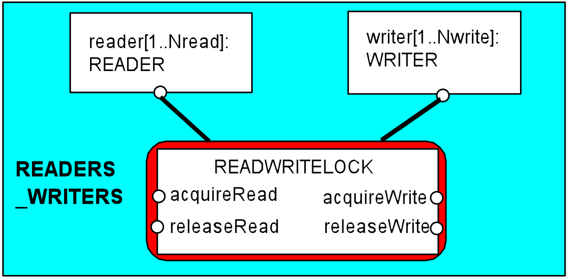


Light blue indicates database access.

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/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).



readers/writers model: READER & WRITER

```
set Actions =
  {acquireRead,releaseRead,acquireWrite,releaseWrite}

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

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

The alphabet extension ensures 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.

readers/writers model: RW_LOCK

```
const False = 0 const True = 1
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]
 ).
```

RW_LOCK maintains a count of the number of readers, and a Boolean for the writers.

readers/writers model: safety

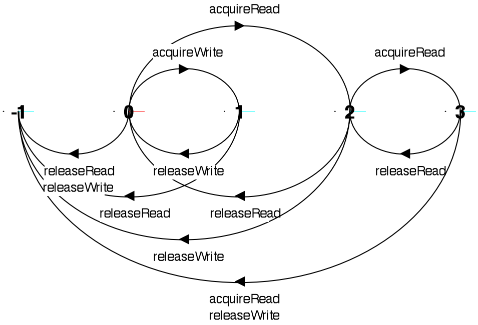
```
property SAFE_RW
  = ( acquireRead -> READING[1]
    | acquireWrite -> WRITING
    ),
READING[i:1..Nread]
  = ( acquireRead -> READING[i+1]
    | when (i>1) releaseRead -> READING[i-1]
    | when (i==1) releaseRead -> SAFE_RW
    ),
WRITING = (releaseWrite -> SAFE_RW).
```

We can check that `RW_LOCK` satisfies the property:

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

Safety and Progress Analysis ?

readers/writers - 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

Adverse conditions using action priority?

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

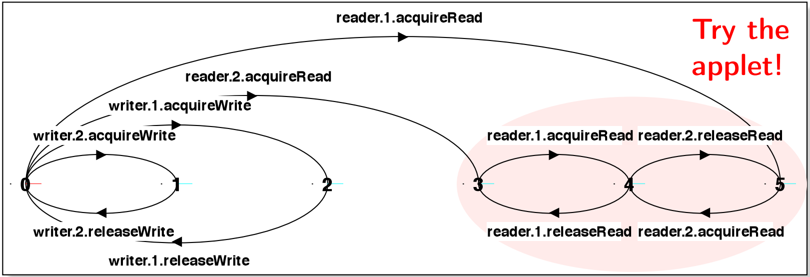
```
||RW_PROGRESS = READERS_WRITERS
              >>{reader[1..Nread].releaseRead,
                writer[1..Nwrite].releaseWrite}.
```

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

```
interface ReadWrite {
    public void acquireRead()
        throws InterruptedException;
    public void releaseRead();
    public void acquireWrite()
        throws InterruptedException;
    public 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(); }
```

Unblock *a single writer* when no more readers.

readers/writers implementation - ReadWriteSafe

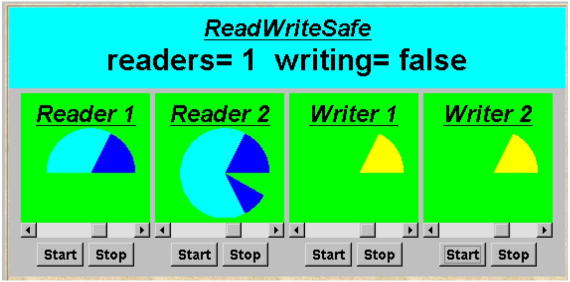
```java
public synchronized void acquireWrite()
             throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }

  public synchronized void releaseWrite() {
    writing = false;
    notifyAll(); // Unblock all readers and writers
  }
}
```

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



Strategy:
Block readers if there is a writer waiting.

```
set Actions = {acquireRead,releaseRead,acquireWrite,
               releaseWrite,requestWrite}

WRITER =(requestWrite ->acquireWrite->modify
                 ->releaseWrite->WRITER)
    + Actions \ {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]
  ).
```

waitingW number of writer requests.

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 there are usually more read accesses than writes, and readers generally want the most up to date information.

readers/writers implementation - ReadWritePriority

```
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

```
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.

readers/writers model - fair model

```
RW_LOCK = RW[0][False][0][False],

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

- **rt** “readers turn” used for fairness.
- waitW are the waiting writers, as before.

Safety and Progress Analysis ?

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