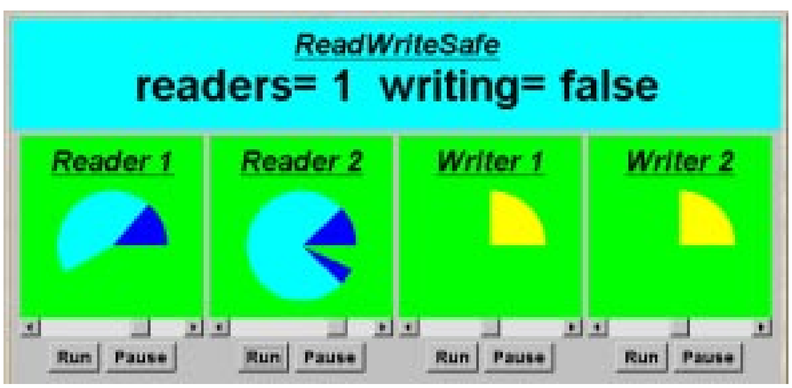
Concurrent Programming
19530-V (WS01)

Lecture 10: Readers and Writers

Dr. Richard S. Hall
rickhall@inf.fu-berlin.de

Readers-Writers Example

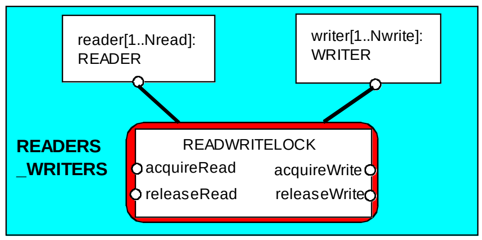


Light blue indicates database access.

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 the database.

Readers-Writers Model

- Events or actions of interest?
  - **acquireRead**, **releaseRead**, **acquireWrite**, **releaseWrite**
- Identify processes
  - **Reader**, **Writer**, and **RW_Lock**
- Identify properties.
  - **RW_Safe**
  - **RW_Progress**
- Define each process
  - Interactions and structure



Readers-Writers Model

```
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 since the actions **examine** and **modify** are not relevant for access synchronization.

Readers-Writers Lock Model

```
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]
      ).
```

The lock maintains a count of the number of readers and a Boolean for a single writer.

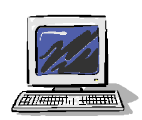
Readers-Writers Safety Property

```
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).
```

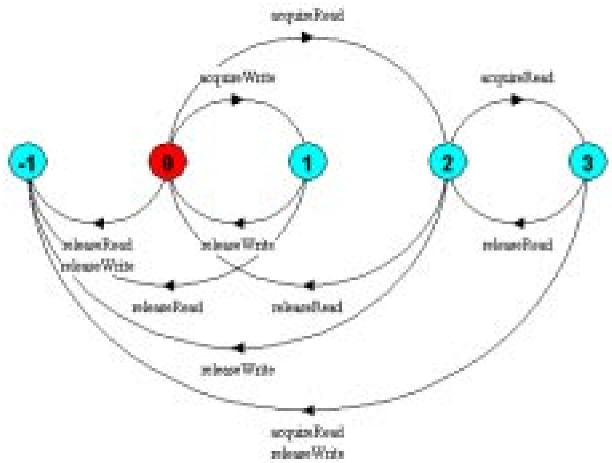
We can check that **RW_LOCK** satisfies the safety property…

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

Safety analysis? LTS?



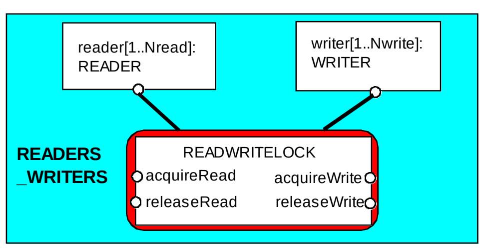
Readers-Writers Safety Property



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 Composition




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

Safety and progress analysis?

Readers-Writers Safety Property

```
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**

How do we model adverse conditions using action priority?
We lower the priority of the release actions for both readers and writers.

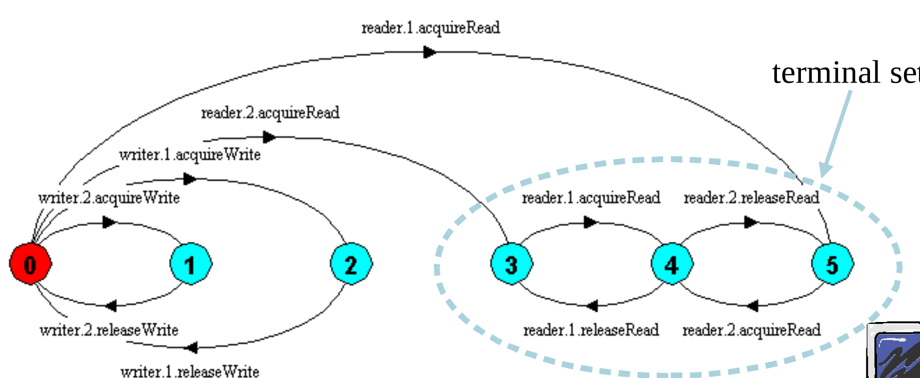
```
||RW_PROGRESS = READERS_WRITERS
                >>{reader[1..Nread].releaseRead,
                   writer[1..Nwrite].releaseWrite}.
```

Progress analysis? LTS?

Readers-Writers 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 Safety Property

We will concentrate on the monitor implementation

```
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.

First, the ***safe*** *implementation...*

Readers-Writers Safety Property

```
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();
  }

// continued...
```

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

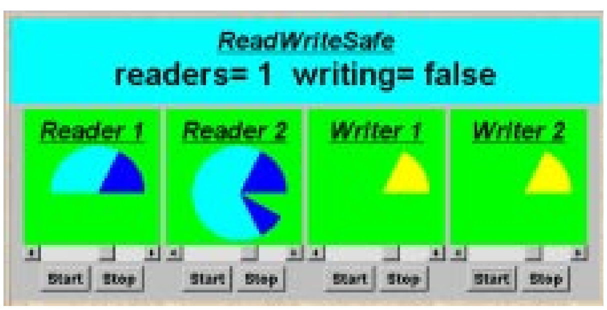
Readers-Writers Safety Property

```
// ...continued from previous slide

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

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

Unblock ***all readers***

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

Readers-Writers with Writer Priority



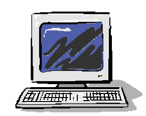
Strategy:
Block readers if there is a writer waiting.

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

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

Readers-Writers with 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]
  ).
```

Safety and progress analysis?



Readers-Writers with Writer Priority

Property RW_SAFE

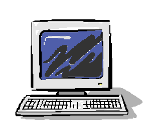
```
No deadlocks/errors
```

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: readers might always wait for writers.

In practice, this may be satisfactory because there might be more read access than write and readers generally want the most up to date information.



Readers-Writers with Writer Priority

```
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();
  }

// continued...
```


Readers-Writers with Writer Priority

```
// ...continued from previous slide

  public synchronized void acquireWrite() {
    ++waitingW;
    while (readers>0 || writing) try { wait();}
          catch(InterruptedException e){}
    --waitingW;
    writing = true;
  }

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

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