Final Repetition and Questions

INF2140 Parallel Programming

May 23, 2012

Plan

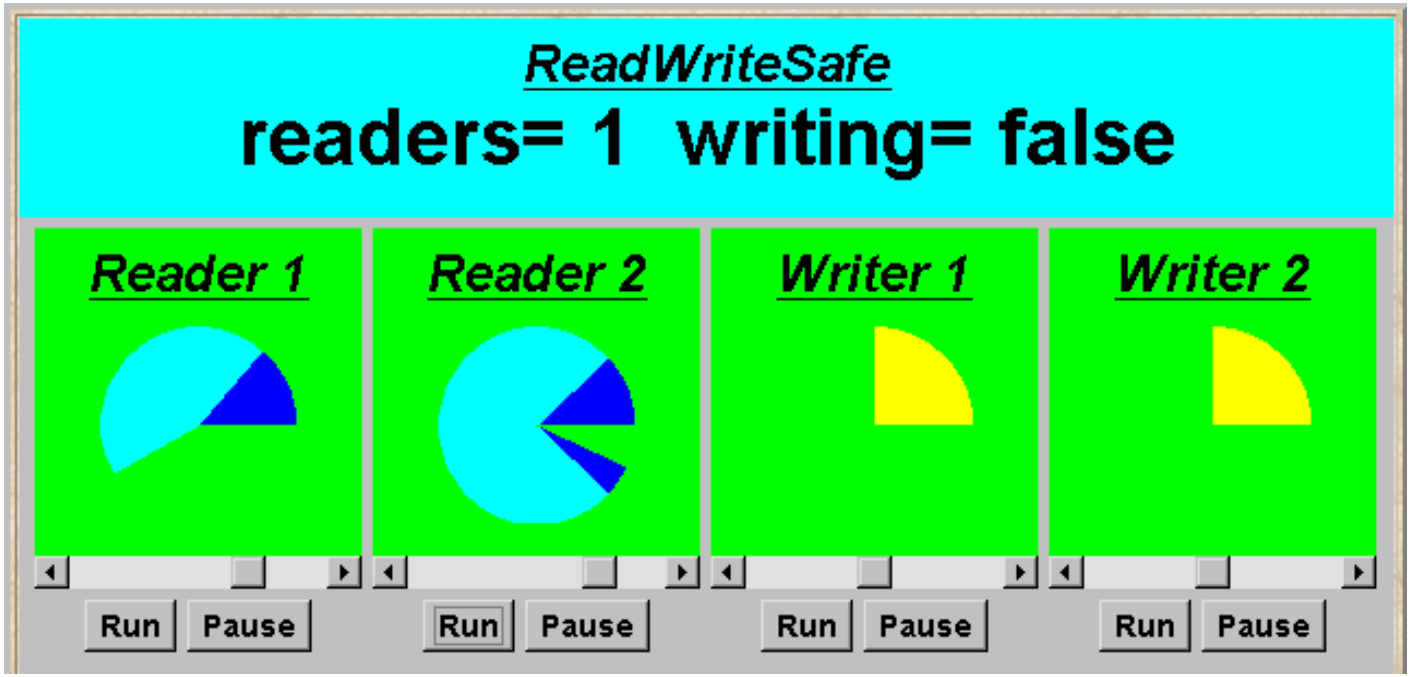
- repetition of FSP
- main concepts
- a major example
  - **Readers Writers example**
  - FSP things
  - safety and progress
  - fairness
  - analysis and implementation

Rough overview FSP to Java

FSP	Java
(main) process	thread
process after comma	part of thread
process[index]	thread with variable
process definition	run method
action	method
indexed action	method with parameter
recursion	while/recursion in run
alphabet	interface
labeling/prefixing (:)	instantiation/creation
process sharing (::)	not relevant (caller/callee)
high priority (>>)	no fairness mechanisms
low priority (<<)	no fairness mechanisms
hiding	non-visible methods

Classic example: Readers Writers

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

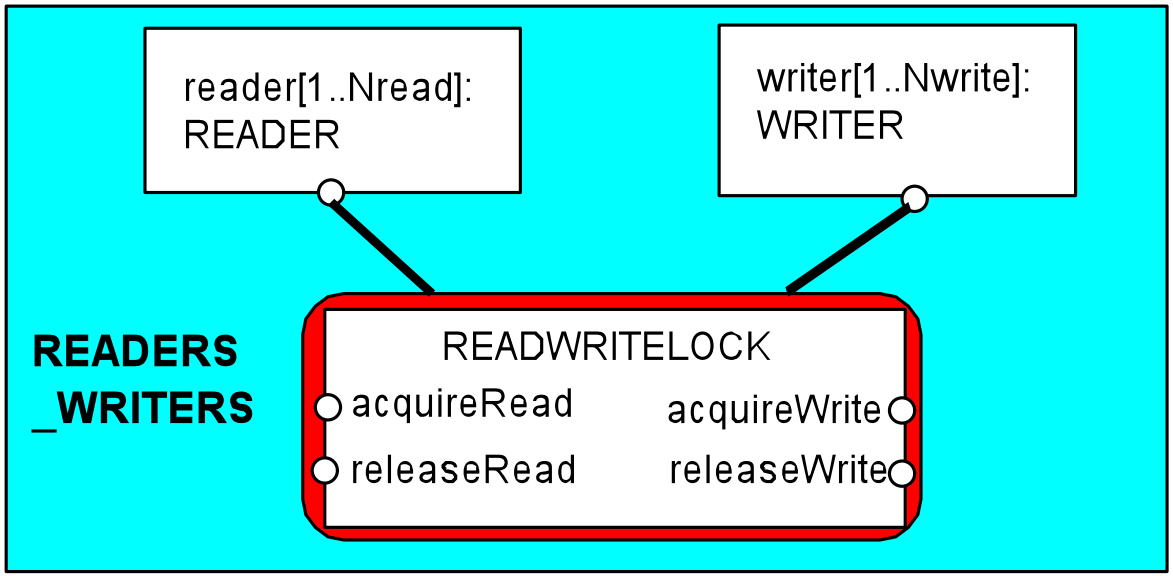


Light blue indicates database access.

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

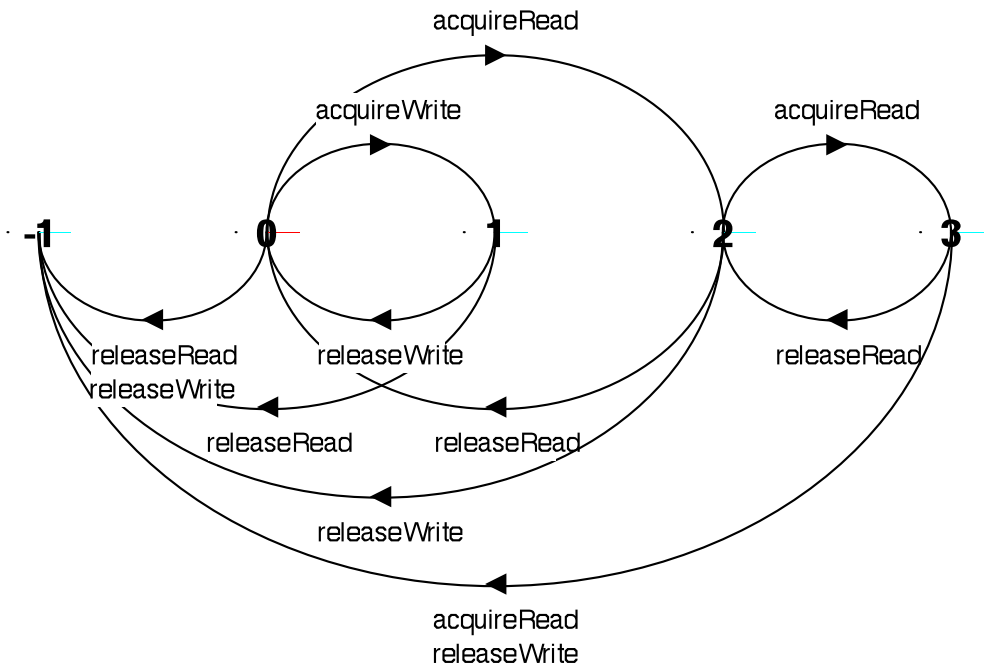
```
property SAFE_RW
  = ( acquireRead -> READING[1]
    | acquireWrite -> WRITING
    ),
READING[i:1..Nread]
  = ( acquireRead -> READING[i+1]
    | when (i>1) releaseRead -> READING[i-1]
    | when (i==1) releaseRead -> SAFE_RW
    ),
WRITING = (releaseWrite -> SAFE_RW).
```

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.

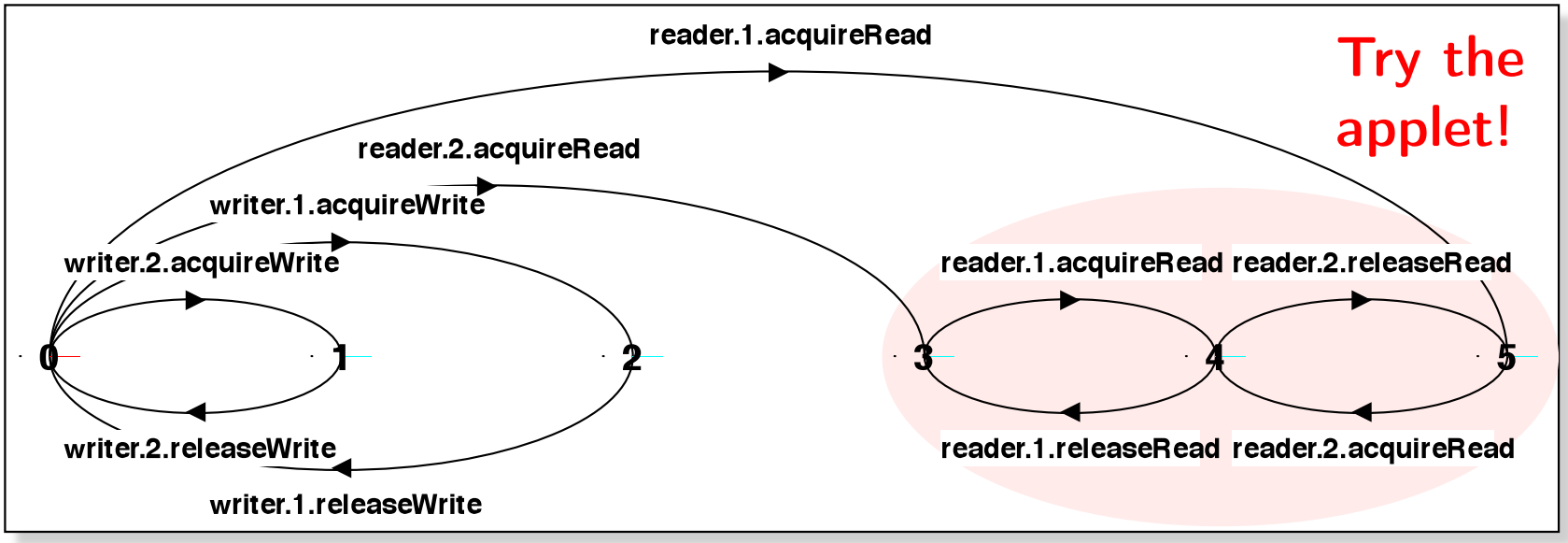
```
||RW_PROGRESS = READERS_WRITERS
              >>{reader[1..Nread].releaseRead,
                 writer[1..Nwrite].releaseWrite}.
```

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

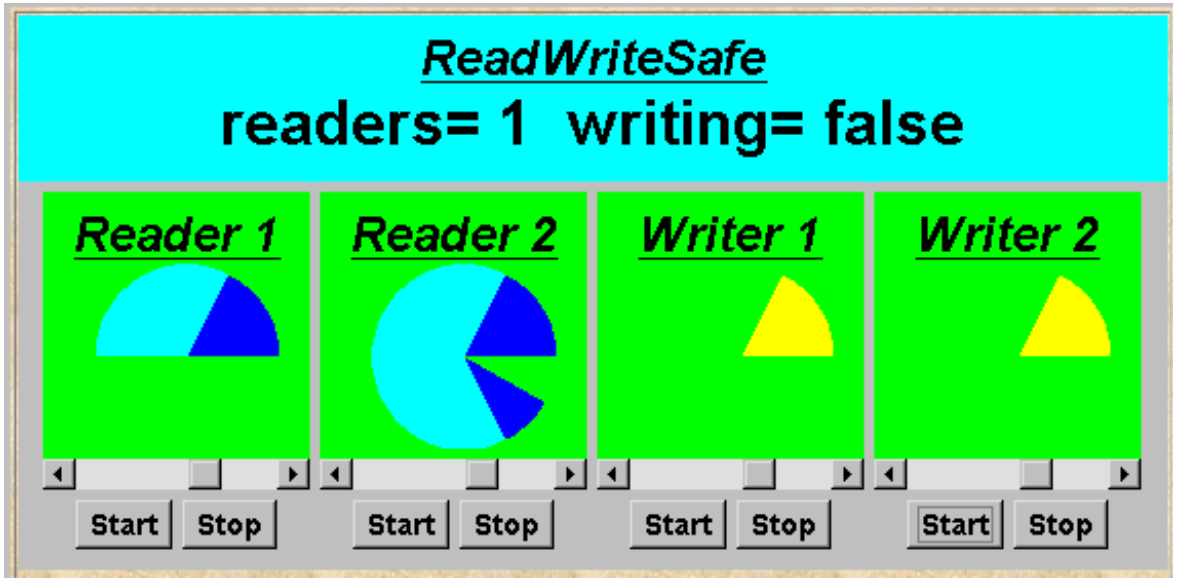
```
public synchronized void acquireWrite()
                throws InterruptedException {
    while (readers>0 || writing) wait();
    writing = true;
  }

  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 ?