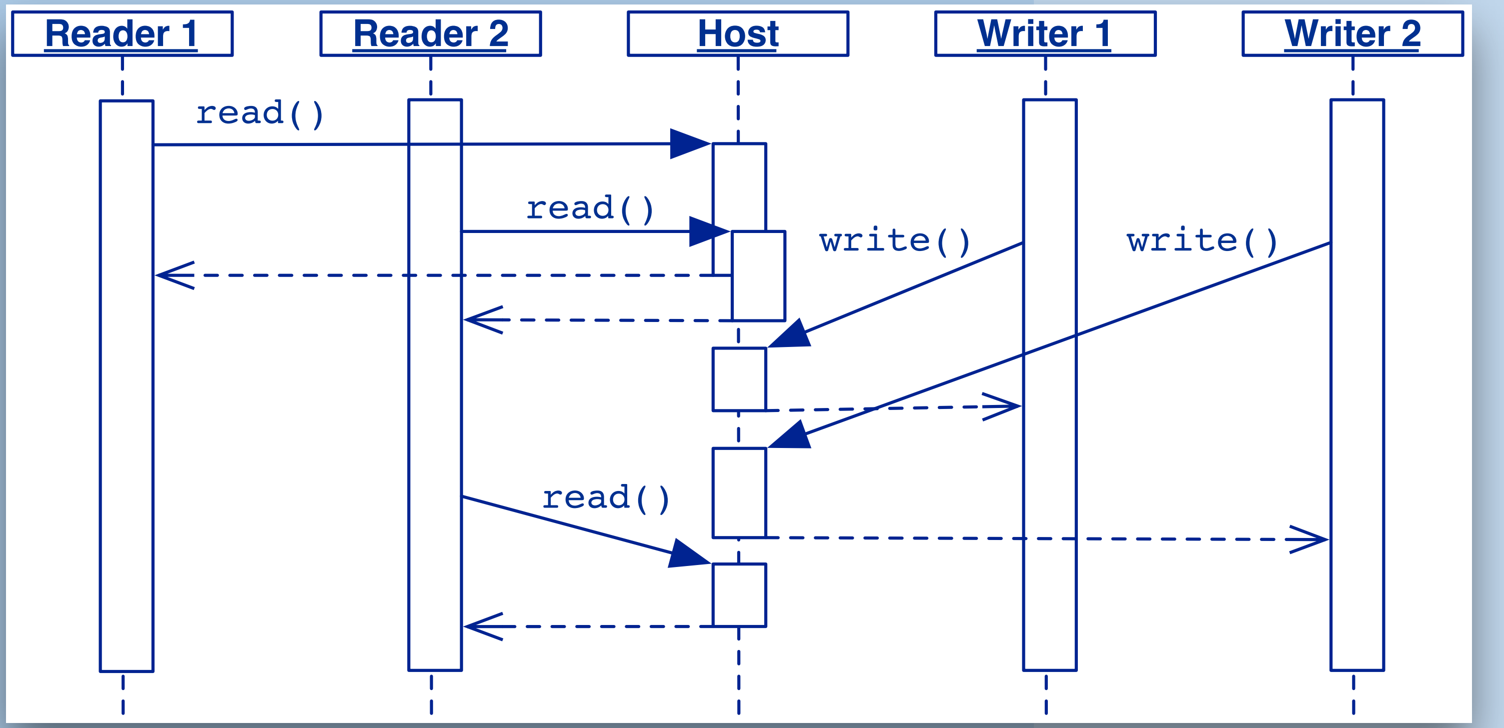
9. Fairness and Optimism

Oscar Nierstrasz



Selected material © Magee and Kramer

Roadmap

> Concurrently available methods
    —Priority, Fairness and Interception
> Readers and Writers
    —Readers and Writers policies
> Optimistic methods

Roadmap

> **Concurrently available methods**
>   —Priority, Fairness and Interception
> Readers and Writers
>   —Readers and Writers policies
> Optimistic methods

Pattern: Concurrently Available Methods

Intent: Non-interfering methods are made concurrently available by implementing policies to *enable and disable methods* based on the current state and running methods.

Applicability

> Host objects are accessed by many different threads.

> Host services are not completely interdependent, so need not be performed under mutual exclusion.

> You need to improve throughput for some methods by eliminating nonessential blocking.

> You want to prevent various accidental or malicious starvation due to some client forever holding its lock.

> Full synchronization would needlessly make host objects prone to deadlock or other liveness problems.

A classical example is that of objects with many readers and few writers. Readers may concurrently access the object. Only a writer should lock out other readers or writers.

Another example is an object with complex state. Finer grained concurrency control over parts of the object's state can improve liveness. (Imagine a database where the entire database is locked for each transaction, rather than individual tables or tuples.)

Concurrent Methods — design steps

Layer concurrency control policy over mechanism by:

Policy Definition:

- When may methods run concurrently?
- What happens when a disabled method is invoked?
- What priority is assigned to waiting tasks?

Instrumentation:

- Define state variables to detect and enforce policy.

Interception:

- Have the host object intercept public messages and then relay them under the appropriate conditions to protected methods that actually perform the actions.

Roadmap

- Concurrently available methods
  - —**Priority, Fairness and Interception**
- Readers and Writers
  - —Readers and Writers policies
- Optimistic methods

Priority

Priority may depend on any of:

- Intrinsic attributes of tasks (class & instance variables)
- Representations of task priority, cost, price, or urgency
- The number of tasks waiting for some condition
- The time at which each task is added to a queue
- Fairness guarantees that each waiting task will eventually run
- Expected duration or time to completion of each task
- The desired completion time of each task
- Termination dependencies among tasks
- The number of tasks that have completed
- The current time

Fairness

There are subtle differences between definitions of fairness:

- Weak fairness: If a process *continuously* makes a request, *eventually* it will be granted. (Dog begging for food.)

- Strong fairness: If a process makes a request *infinitely often*, *eventually* it will be granted. (Cat checking for food in its bowl.)

- Linear waiting: If a process makes a request, it will be granted *before* any other process is granted the request *more than once*. (Buying one-per-customer tickets.)

- FIFO (first-in first out): If a process makes a request, it will be granted *before* that of any process *making a later request*. (Stand in queue at post office.)

Weak fairness may allow a process to starve. Linear waiting and FIFO are easy to implement, though “later” may not be well-defined in a distributed environment.

Interception

Interception strategies include:

> ***Pass-Throughs***: The host maintains a set of *immutable references to helper objects* and simply relays all messages to them within unsynchronized methods.

> ***Lock-Splitting***: Instead of splitting the class, *split the synchronization locks* associated with subsets of the state.

> ***Before/After methods***: Public methods contain *before/after processing* surrounding calls to non-public methods in the host that perform the services.

With a pass-through, the host is immutable, so requires no synchronization. The helper objects are individually synchronized, but there may be several of them that are concurrently available. In this case, the host resembles a Facade (design pattern).

With lock splitting, the state of the object (a set of instance variables) is partitioned into several subsets, each with its own lock. ConcurrentHashMap uses an array of 16 locks, each of which guards one of the 16 hash buckets.

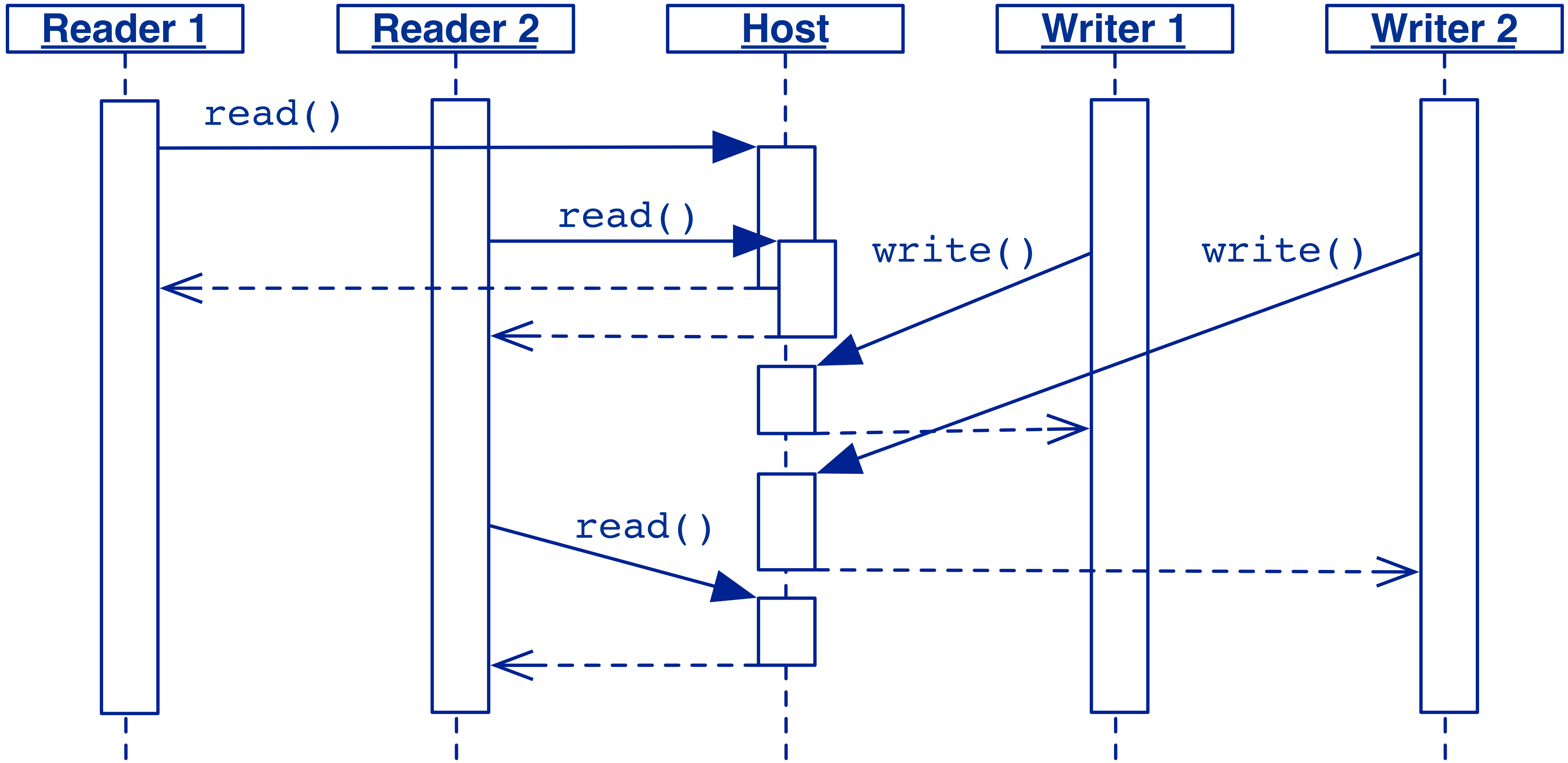
Before and after methods may contain finer grained synchronization policies to guard the actual private (or protected) methods that do the work. We will see examples later of Readers and Writers that use such a strategy.

Roadmap

> Concurrently available methods
  — Priority, Fairness and Interception
> **Readers and Writers**
  — Readers and Writers policies
> Optimistic methods

Concurrent Reader and Writers

“Readers and Writers” is a family of concurrency control designs in which “Readers” (non-mutating accessors) may concurrently access resources while “Writers” (mutative, state-changing operations) require exclusive access.



Many applications make use of some kind of “database” or persistent store that is frequently accessed by “readers” but only periodically updated by “writers”. Think of network configuration tables, or a customer database. Readers do not interfere with each other, since they treat the data as though they were immutable. Only writers conflict with readers or other writers. If most processes only need read access, it is wasteful to impose strict mutual exclusion on readers.

In the scenario, Readers 1 and 2 can access the Host concurrently. Only when Writer 1 requests access, it must wait until it can gain exclusive access. Similarly Writer 2 and Reader 2 must wait to gain access.

“Readers and Writers” refers to a family of solutions to the problem of safely (and fairly) enabling concurrent access to resources shared by reader and writer processes.

https://en.wikipedia.org/wiki/Readers–writers_problem

Readers/Writers Model

We are interested only in capturing who gets access:

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

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

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

These finite state processes capture the essence of the Readers/ Writers model. Synchronization policies hook into the before and after events that acquire and release read or write access. Examine and modify are internal actions. The others are visible, and may be synchronized.

A Simple RW Protocol

```
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]
   ).
```

This simple mutual exclusion protocol does not impose any priority. Multiple readers are allowed. Readers lock out writers; writers lock out readers and writers. Note that some actions don’t make sense in certain states (e.g., release in start state).

Safety properties

We specify the safe interactions:

```
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 ).
```

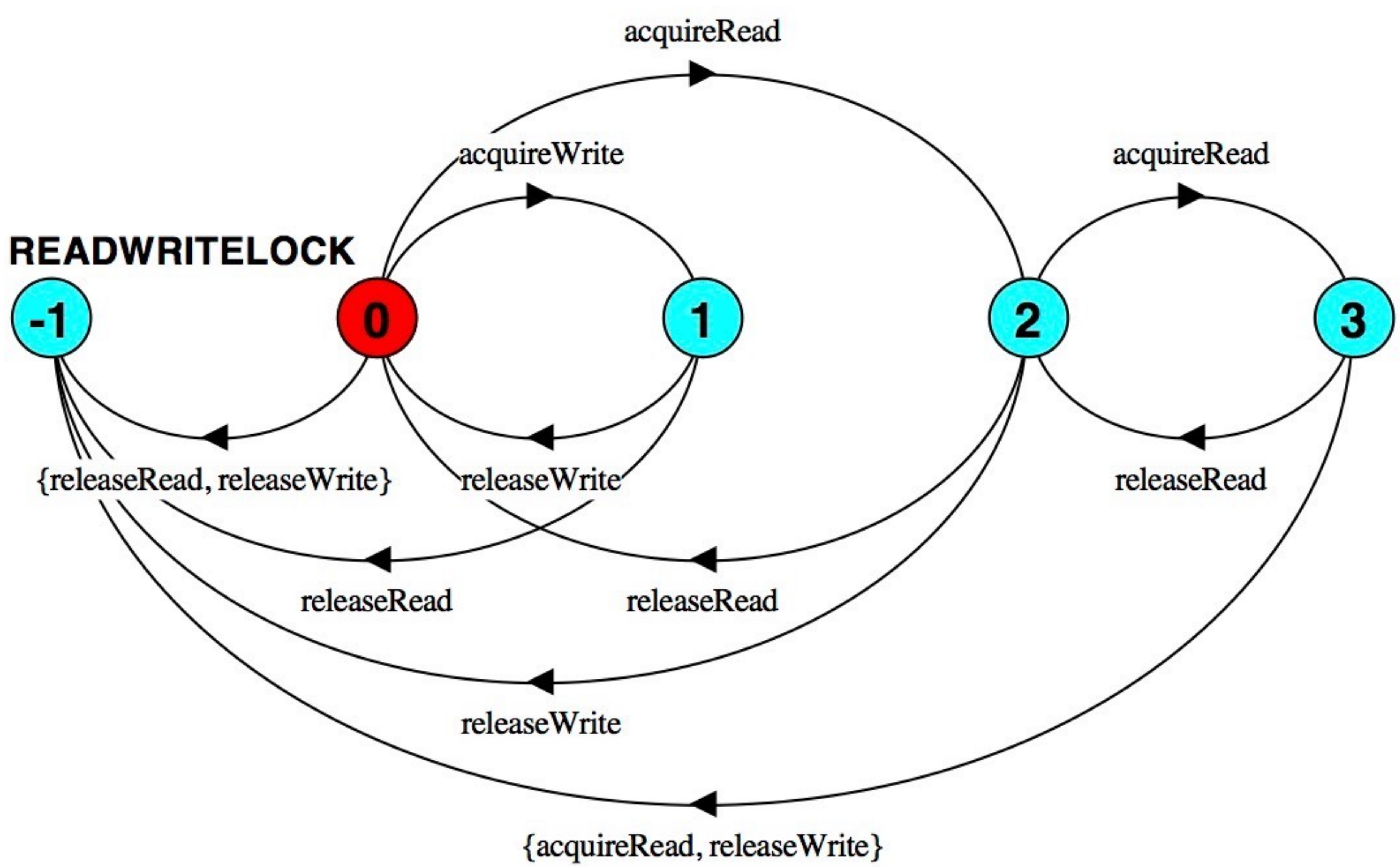
We specify here that multiple read locks may be acquired, but only one write lock.

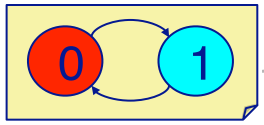
The system may be in one of three states: ready for a reader or writer to get in; reading (any number); writing (one writer). this also specifies that we must acquire before releasing.

Safety properties ...

And compose them with RW_LOCK:

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





Here we see that RW_LOCK permits several unsafe traces. Actual Readers and Writers may be badly behaved and violate the safety property, for example, by acquiring a read lock and then releasing a write lock.

Composing the Readers and Writers

We compose the READERS and WRITERS with the protocol and check for safety violations:

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

```
No deadlocks/errors
```

Each READER/WRITER is forced to sync with the lock.

We see that READER and WRITER are well-behaved with respect to the specified safety property.

Progress properties

We similarly specify liveness properties:

```
progress WRITE[i:1..Nwrite] = writer[i].acquireWrite
progress READ[i:1..Nwrite] = reader[i].acquireRead
```

Assuming fair choice, we have no liveness problems

```
Progress Check...
No progress violations detected.
```

Our liveness properties simply state that both readers and writers are all guaranteed to eventually acquire the resource. Under fair choice, there are no progress violations.

Priority

If we give priority to acquiring locks, we may starve out writers!

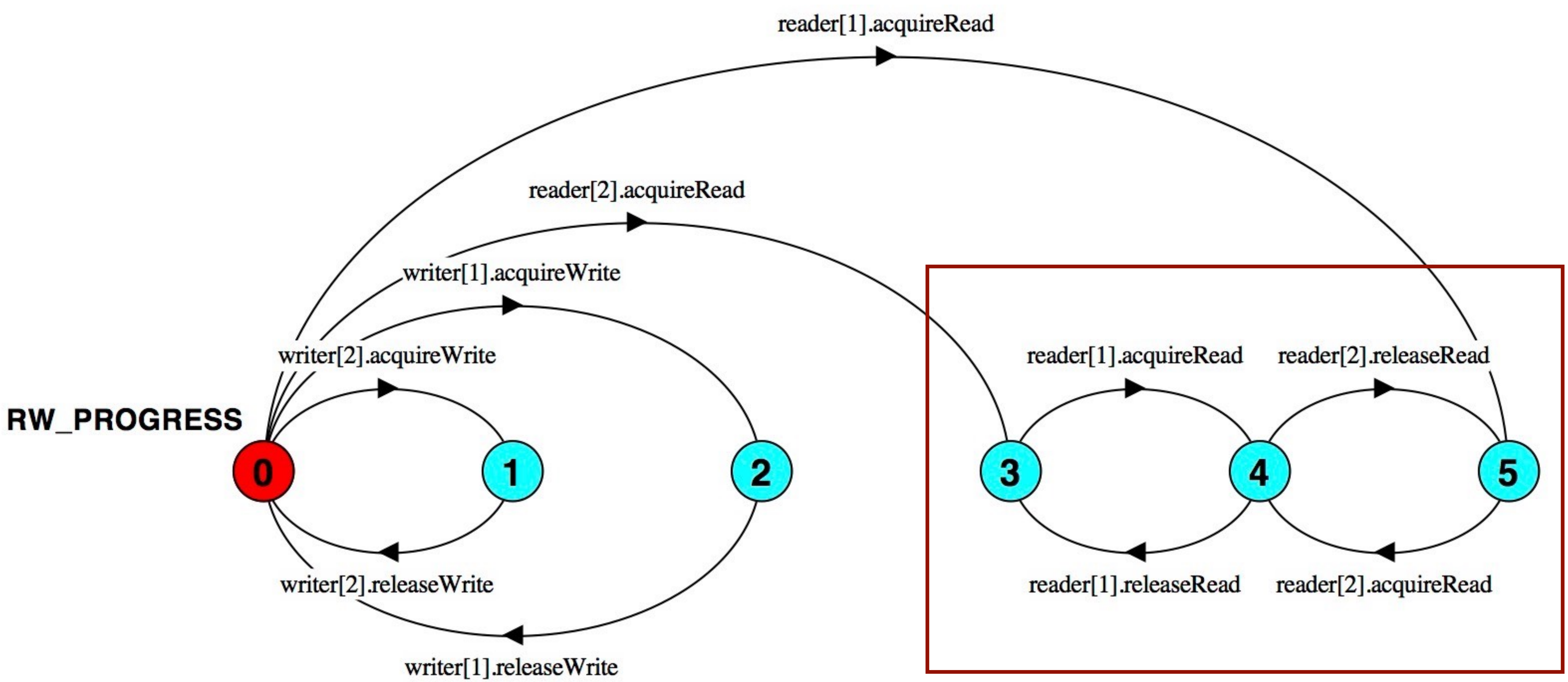
```
||RW_PROGRESS =
        READERS_WRITERS
              >>{reader[1..Nread].releaseRead,
              writer[1..Nread].releaseWrite}.
```

```
Progress violation: WRITE.1 WRITE.2
Trace to terminal set of states:
       reader.1.acquireRead tau
Actions in terminal set:
       reader[1..2].{acquireRead, releaseRead}
```

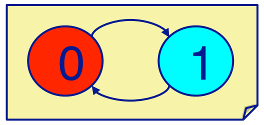
If we give priority to acquiring over releasing (`READERS_WRITERS >>{…}`), then the writer may be starved out. (In `P>>B`, the actions in `B` have lowest priority; in `P<<B` they have highest priority.)

The LTSA model checker shows us that two readers can overlap in acquiring and releasing the read locks, thus starving out any writers.

Starvation



NB: minimize to eliminate tau actions



In this scenario, there is always some reader holding the read lock, so writers are starved out.

Roadmap

- Concurrently available methods
  - Priority, Fairness and Interception
- Readers and Writers
  - **Readers and Writers policies**
- Optimistic methods

Readers and Writers Policies

Individual policies must address:

- Can new *Readers join already active Readers* even if a Writer is waiting?
  - —if yes, *Writers may starve*
  - —if not, the *throughput of Readers decreases*

- If both Readers and Writers are waiting for a Writer to finish, *which should you let in first*?
  - —Readers? A Writer? FCFS? Random? Alternate?
  - —Similar choices exist after Readers finish.

- Can *Readers upgrade to Writers* without having to give up access?

Policies ...

> ***A typical set of choices:***
>
> —Block incoming Readers if there are waiting Writers.
>
> —“Randomly” choose among incoming threads (i.e., let the scheduler choose).
>
> —No upgrade mechanisms.

Before/after methods are the simplest way to implement Readers and Writers policies.

Readers and Writers example

Implement state tracking variables

```
public abstract class ReadersWritersStateTracking {
   protected int activeReaders = 0;       // zero or more
   protected int activeWriters = 0;       // always zero or one
   protected int waitingReaders = 0;
   protected int waitingWriters = 0;
   protected abstract void doRead();      // defined by subclass
   protected abstract void doWrite();
...
```

ReadersWriters

Readers and Writers example

Public methods call protected before/after methods

```
...
   public void read() {          // unsynchronized
      beforeRead();              // obtain access
      doRead();
      afterRead();               // release access
   }
   public void write() {
      beforeWrite();
      doWrite();
      afterWrite();
   }
...
```

Readers and Writers example

Synchronized before/after methods maintain state variables

```
...
   protected synchronized void beforeRead() {
      ++waitingReaders;           // available to subclasses
      while (!allowReader()) {
         try { wait(); }
         catch (InterruptedException ex) {}
      }
      --waitingReaders;
      ++activeReaders;
   }
   protected synchronized void afterRead() {
      --activeReaders;
      notifyAll();
   }
...
```

Readers and Writers example

Different policies can use the same state variables …

```
...
   protected boolean allowReader() {          // default policy
      return waitingWriters == 0 && activeWriters == 0;
   }
...
```

Can you define suitable before/after methods for Writers?

Notice how this design allows us to implement a variety of policies in the helper methods, especially `allowReader()` and `allowWriter()`.

Exercise: What (simple) policy might you define to ensure that neither readers nor writers can be starved out?

Readers and Writers demo

```
class ReadWriteDemo extends ReadersWritersStateTracking {
...
   public void doit() {
      new Reader(this).start();
      ...
   }
...
   protected void doRead() {
      System.out.print("(");
      Thread.yield();
      System.out.print(")");
   }
   protected void doWrite() {
      System.out.print("[");
      ...
    }
}
```

```
(()())[][][][][][][]
[][][][](()()()()() ()
()()()()()())[][] [][]
[][][][][][][] (()()()
()()))
```

Roadmap

- Concurrently available methods
  - —Priority, Fairness and Interception
- Readers and Writers
  - —Readers and Writers policies
- **Optimistic methods**

Pattern: Optimistic Methods

Intent: Optimistic methods attempt actions, but *rollback state in case of interference*. After rollback, they either throw failure exceptions or retry the actions.

Applicability

> Clients can tolerate either failure or retries.

—If not, consider using guarded methods .

> You can avoid or cope with livelock.

> You can undo actions performed before failure checks

—***Rollback/Recovery:*** undo effects of each performed action. If messages are sent to other objects, they must be undone with “anti-messages”

—***Provisional action:*** “pretend” to act, delaying commitment until interference is ruled out.

Note that if the likelihood of failure is high, this will lead to busy-waiting of clients repeatedly attempting to complete their actions.

See also:

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

Optimistic Methods — design steps

Collect and encapsulate all mutable state so that it can be tracked as a unit:

> Define an immutable helper class holding values of all instance variables.

> Define a representation class, but make it mutable (allow instance variables to change), and additionally include a version number (or transaction identifier) field or even a sufficiently precise time stamp.

> Embed all instance variables, plus a version number, in the host class, but define `commit` to take as arguments all assumed values and all new values of these variables.

> Maintain a serialized copy of object state.

> Various combinations of the above ...

All of these approaches offer a way to update the state, but then to delay the decision to either commit that state or roll back to the previous state.

“Maintain a serialized copy of object state.” — e.g., a backup copy of the state that is updated as an atomic action.

Detect failure ...

Provide an operation that simultaneously detects version conflicts and performs updates via a method of the form:

```
class Optimistic {                          // code sketch
   private State currentState;              // immutable values
   synchronized boolean commit(State assumed, State next)
   {
      boolean success = (currentState.equals(assumed)) ;
      if (success)
         currentState = next;
      return success;
   }
}
```

An Optimistic Bounded Counter

```
public class BoundedCounterOptimistic
   extends BoundedCounterAbstract {

   protected synchronized boolean commit(Long oldc, Long newc) {
      boolean success = (count == oldc);
      if (success) {
         count = newc;
      } else {
         System.err.println("COMMIT FAILED -- RETRYING");
      }
      return success;
   }
```

Counter

The optimistic bounded counter checks that the state has not been changed by another process. If it has, it aborts the transactions, otherwise it updates the state.

Detect failure ...

Structure the main actions of each public method as follows:

```
State assumed = currentState() ;
State next = ...                        // compute optimistically
if (!commit(assumed, next))
   rollback();
else
   otherActionsDependingOnNewStateButNotChangingIt();
```

An Optimistic Bounded Counter

```
    ...
    public synchronized long value() {
        return count;
    }
    public void inc() {
        for (;;) {                             // thinly disguised busy-wait!
            long prev = this.value();
            long val = prev;
            if (val < MAX && commit(prev, val+1)) {
                break;
            }
            Thread.yield();                    // is there another thread?!
        }
    }
    ...
}
```

Here we can see that optimism may lead to busy-waiting if there is a lot of contention. In the worst case, a process may be starved out (livelock).

Handle conflicts ...

Choose and implement a policy for dealing with commit failures:

> *Throw an exception* upon commit failure that tells a client that it may retry.

> *Internally retry* the action until it succeeds.

> *Retry some bounded number of times*, or until a timeout occurs, finally throwing an exception.

> *Pessimistically synchronize* selected methods which should not fail.

Ensure progress ...

Ensure progress in case of internal retries

- *Immediately retrying* may be counterproductive!
- *Yielding* may only be effective if all threads have reasonable priorities and the Java scheduler at least approximates *fair choice* among waiting tasks (which it is not guaranteed to do)!
- *Limit retries* to avoid livelock

What you should know!

> *What criteria might you use to prioritize threads?*
> *What are different possible definitions of fairness?*
> *What are readers and writers problems?*
> *What difficulties do readers and writers pose?*
> *When should you consider using optimistic methods?*
> *How can an optimistic method fail? How do you detect failure?*

Can you answer these questions?

- *When does it make sense to split locks? How does it work?*
- *When should you provide a policy for upgrading readers to writers?*
- *What are the dangers in letting the (Java) scheduler choose which writer may enter a critical section?*
- *What are advantages and disadvantages of encapsulating synchronization conditions as helper methods?*
- *How can optimistic methods livelock?*