9. Fairness and Optimism

Prof. O. 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.
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 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.
There are subtle differences between definitions of fairness:

> **Weak fairness**: If a process *continuously* makes a request, *eventually* it will be granted.

> **Strong fairness**: If a process makes a request *infinitely often*, *eventually* it will be granted.

> **Linear waiting**: If a process makes a request, it will be granted *before* any other process is granted the request *more than once*.

> **FIFO** (first-in first out): If a process makes a request, it will be granted *before* that of any process *making a later request*. 
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.
Roadmap

- Concurrently available methods
  - Priority, Fairness and Interception
- Readers and Writers
  - Readers and Writers policies
- Optimistic methods
“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.
Readers/Writers Model

We are interested only in capturing who gets access:

\[
\text{set } \text{Actions} = \{\text{acquireRead, releaseRead, acquireWrite, releaseWrite}\}
\]

\[
\text{READER} = (\text{acquireRead} \rightarrow \text{examine} \rightarrow \text{releaseRead} \rightarrow \text{READER}) \\
+\text{Actions \{examine\}}.
\]

\[
\text{WRITER} = (\text{acquireWrite} \rightarrow \text{modify} \rightarrow \text{releaseWrite} \rightarrow \text{WRITER}) \\
+\text{Actions \{modify\}}.
\]
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]
  ).
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 ).
And compose them with RW_LOCK:

\[ | | \text{READWRITELOCK} = (\text{RW\_LOCK} \ || \ \text{SAFE\_RW}). \]
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
Progress properties

We similarly specify liveness properties:

progress \text{WRITE}[i:1..N\text{write}] = \text{writer}[i].\text{acquireWrite}
progress \text{READ}[i:1..N\text{write}] = \text{reader}[i].\text{acquireRead}

Assuming \textit{fair choice}, we have no liveness problems

Progress Check...
No progress violations detected.
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\}
Starvation
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?*
  — Similar choices exist after Readers finish.

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

```java
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

```java
public void read() {
    beforeRead(); // unsynchronized
    doRead(); // obtain access
    afterRead(); // release access
}

public void write() {
    beforeWrite();
    doWrite();
    afterWrite();
}
...
Readers and Writers example

Synchronized before/after methods maintain state variables

```java
...  
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?
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.
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 ...
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_ == assumed) ;
        if (success)
            currentState_ = next;
        return success;
    }
}
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;
    }
}
Detect failure ...

Structure the main actions of each public method as follows:

```java
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 c = count; long v = c.longValue();
    if (v < MAX && commit(c, new Long(v+1))) break;
    Thread.yield(); // is there another thread?!
  }
}
...
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 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?
License

http://creativecommons.org/licenses/by-sa/2.5/

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.