5. Liveness and Guarded Methods

Prof. O. Nierstrasz

Selected material © Magee and Kramer
Roadmap

- Liveness
  - Progress Properties

- Deadlock
  - The Dining Philosophers problem
  - Detecting and avoiding deadlock

- Guarded Methods
  - Checking guard conditions
  - Handling interrupts
  - Structuring notification
Roadmap

- **Liveness**
  - Progress Properties

- **Deadlock**
  - The Dining Philosophers problem
  - Detecting and avoiding deadlock

- **Guarded Methods**
  - Checking guard conditions
  - Handling interrupts
  - Structuring notification
Liveness

> A liveness property asserts that something good eventually happens

> A progress property asserts that it is always the case that an action is eventually executed

> Progress is the opposite of starvation, the name given to a concurrent programming situation in which an action is never executed
Liveness and Guarded Methods

Liveness Problems

A program may be “safe”, yet suffer from various kinds of liveness problems:

**Starvation:** (AKA “indefinite postponement”)
> The system as a whole makes progress, but some individual processes don’t

**Dormancy:**
> A waiting process fails to be woken up

**Premature termination:**
> A process is killed before it should be

**Deadlock:**
> Two or more processes are blocked, each waiting for resources held by another
Fair Choice: If a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

If a coin were tossed an infinite number of times, we would expect that both heads and tails would each be chosen infinitely often. This assumes fair choice!

COIN = ( toss -> heads -> COIN | toss -> tails -> COIN ).
Safety vs. Liveness

Consider:

```markdown
property SAFE = ( heads -> SAFE | tails -> SAFE ).
```

The safety properties of COIN are not very interesting.

*How do we express what *must* happen?*
Progress properties

\textbf{progress } P = \{a_1, a_2 \ldots a_n\}

asserts that in an \textit{infinite execution} of a target system, \textit{at least one} of the actions \textit{a1, a2...an} will be executed \textit{infinitely often}.

\textbf{progress } HEADS = \{\text{heads}\}
\textbf{progress } TAILS = \{\text{tails}\}

\textbf{No progress violations detected.}
Suppose we have both a normal coin and a trick coin

\[
\begin{align*}
\text{TWO\text{COIN}} &= (\text{pick} \rightarrow \text{COIN} \mid \text{pick} \rightarrow \text{TRICK}), \\
\text{TRICK} &= (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}), \\
\text{COIN} &= (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \mid \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}).
\end{align*}
\]
A **terminal set** of states is one in which *every state is mutually reachable but no transitions leads out of the set.*

The terminal set \{1, 2\} violates progress property TAILS

\[
\begin{align*}
\text{progress } \text{HEADS} &= \{\text{heads}\} \\
\text{progress } \text{TAILS} &= \{\text{tails}\} \\
\text{progress } \text{HEADSorTAILS} &= \{\text{heads, tails}\}
\end{align*}
\]

Progress violation: TAILS
Trace to terminal set of states: pick
Actions in terminal set: \{toss, heads\}
Consider:

property REPICK = ( pick -> toss -> REPICK ).

Trace to property violation in REPICK:
- pick
- toss
- heads
- toss

How does this safety property expose the flaw in the system?
How would you fix the TWOCOIN to have this property pass?
Roadmap

> Liveness
  — Progress Properties

> Deadlock
  — The Dining Philosophers problem
  — Detecting and avoiding deadlock

> Guarded Methods
  — Checking guard conditions
  — Handling interrupts
  — Structuring notification
Deadlock

Four necessary and sufficient conditions for deadlock:

1. **Serially reusable resources:**
   - the deadlocked processes share resources under mutual exclusion.

2. **Incremental acquisition:**
   - processes hold on to acquired resources while waiting to obtain additional ones.

3. **No pre-emption:**
   - once acquired by a process, resources cannot be pre-empted but only released voluntarily.

4. **Wait-for cycle:**
   - a cycle of processes exists in which each process holds a resource which its successor in the cycle is waiting to acquire.
Waits-for cycle

Has A awaits B

Has E awaits A

Has B awaits C

Has D awaits E

Has C awaits D
A **deadlocked state** is one with no outgoing transitions

In FSP: STOP process

\[
\text{MOVE} = ( \text{north} \rightarrow \\
\quad ( \text{south} \rightarrow \text{MOVE} \\
\quad \quad | \text{north} \rightarrow \text{STOP} \\
\quad ) \\
\quad )
\]

Progress violation for actions: \{north, south\}
Trace to terminal set of states: north north
Actions in terminal set set: {}
The Dining Philosophers Problem

> Philosophers alternate between *thinking* and *eating*.
> A philosopher needs *two forks* to eat.
> No two philosophers may hold the same fork simultaneously.
> There must be *no deadlock and no starvation*.
> Want efficient behaviour under absence of contention.
Deadlocked diners

A deadlock occurs if a \textit{waits-for cycle} arises in which each philosopher grabs one fork and waits for the other.
Dining Philosophers illustrate many classical safety and liveness issues:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mutual Exclusion</strong></td>
<td>Each fork can be used by one philosopher at a time</td>
</tr>
<tr>
<td><strong>Condition synchronization</strong></td>
<td>A philosopher needs two forks to eat</td>
</tr>
<tr>
<td><strong>Shared variable communication</strong></td>
<td>Philosophers share forks ...</td>
</tr>
<tr>
<td><strong>Message-based communication</strong></td>
<td>... or they can pass forks to each other</td>
</tr>
<tr>
<td><strong>Busy-waiting</strong></td>
<td>A philosopher can poll for forks ...</td>
</tr>
<tr>
<td><strong>Blocked waiting</strong></td>
<td>... or can sleep till woken by a neighbour</td>
</tr>
<tr>
<td><strong>Liveloak</strong></td>
<td>All philosophers can grab the left fork and busy-wait for the right ...</td>
</tr>
<tr>
<td><strong>Deadlock</strong></td>
<td>... or grab the left one and wait (sleep) for the right</td>
</tr>
<tr>
<td><strong>Starvation</strong></td>
<td>A philosopher may starve if the left and right neighbours are always faster at grabbing the forks</td>
</tr>
<tr>
<td><strong>Race conditions</strong></td>
<td>Anomalous behaviour depends on timing</td>
</tr>
</tbody>
</table>
Liveness and Guarded Methods

Modeling Dining Philosophers

PHIL = ( sitdown
    -> right.get -> left.get
    -> eat -> left.put -> right.put
    -> arise -> PHIL ).

FORK = ( get -> put -> FORK).

||DINERS(N=5)= forall [i:0..N-1]
    ( phil[i]:PHIL || {phil[i].left,phil[((i-1)+N)%N].right}::FORK ).

Is this system safe? Is it live?
Dining Philosophers Analysis

Trace to terminal set of states:
- phil.0.sitdown
- phil.0.right.get
- phil.1.sitdown
- phil.1.right.get
- phil.2.sitdown
- phil.2.right.get
- phil.3.sitdown
- phil.3.right.get
- phil.4.sitdown
- phil.4.right.get

Actions in terminal set: {}

No further progress is possible due to the waits-for cycle
There are two fundamentally different approaches to eliminating deadlock.

**Deadlock detection:**
> Repeatedly check for waits-for cycles. When detected, choose a victim and force it to release its resources.
  —Common in transactional systems; the victim should “roll-back” and try again

**Deadlock avoidance:**
> Design the system so that a waits-for cycle cannot possibly arise.
There are many solutions offering varying degrees of liveness guarantees:

**Break the cycle**
- Number the forks. Philosophers grab the *lowest numbered fork* first.
- One philosopher grabs forks in the reverse order.

**Philosophers queue to sit down**
- allow *no more than four* at a time to sit down

Do these solutions avoid deadlock? What about starvation? Are they “fair”?
Roadmap

- Liveness
  - Progress Properties
- Deadlock
  - The Dining Philosophers problem
  - Detecting and avoiding deadlock
- Guarded Methods
  - Checking guard conditions
  - Handling interrupts
  - Structuring notification
There are various strategies and techniques to ensure liveness:

> Start with safe design and *selectively remove* synchronization
> Start with live design and selectively *add safety*
> Adopt *design patterns* that limit the need for synchronization
> Adopt *standard architectures* that avoid cyclic dependencies
Pattern: Guarded Methods

**Intent:** Temporarily suspend an incoming thread when an object is not in the right state to fulfil a request, and wait for the state to change rather than balking (raising an exception).
Guarded Methods — applicability

> Clients can *tolerate indefinite postponement*. (Otherwise, use a balking design.)

> You can guarantee that the *required states are eventually reached* (via other requests), or if not, that it is acceptable to block forever.

> You can arrange that *notifications occur after all relevant state changes*. (Otherwise consider a design based on a busy-wait spin loop.)

> …
Guarded Methods — applicability …

…

> You can *avoid or cope with liveness problems* due to waiting threads retaining all synchronization locks.

> You can *construct computable predicates* describing the state in which actions will succeed. (Otherwise consider an optimistic design.)

> Conditions and actions are *managed within a single object*. (Otherwise consider a transactional form.)
The basic recipe is to use `wait` in a conditional loop to block until it is safe to proceed, and use `notifyAll` to wake up blocked threads.

```java
public synchronized Object service() {
    while (wrong State) {
        try { wait(); } 
        catch (InterruptedException e) { }
    }
    // fill request and change state ...
    notifyAll();
    return result;
}
```
Define *interfaces* for the methods, so that classes can implement guarded methods according to *different policies*.

```java
public interface BoundedCounter {
    public static final long MIN = 0; // min value
    public static final long MAX = 10; // max value
    public long value(); // inv't: MIN <= value() <= MAX
                          // init: value() == MIN
    public void inc();    // pre: value() < MAX
    public void dec();    // pre: value() > MIN
}
```

*Counter*
Step: Check guard conditions

> Define a *predicate* that precisely describes the conditions under which actions may proceed. (This can be encapsulated as a helper method.)

> Precede the conditional actions with a *guarded wait loop* of the form:

```java
while (!condition) {
    try { wait(); }
    catch (InterruptedException ex) { ... } }
```

> Optionally, encapsulate this code as a helper method.
If there is only one possible condition to check in this class (and all plausible subclasses), and notifications are issued only when the condition is true, then there is no need to re-check the condition after returning from wait().

Ensure that the object is in a consistent state (i.e., the class invariant holds) before entering any wait (since wait releases the synchronization lock).

— The easiest way to do this is to perform the guards before taking any actions.
Step: Handle interrupts

> Establish a *policy* to deal with InterruptedExceptions. Possibilities include::

— **Ignore interrupts** (i.e., an empty catch clause), which preserves safety at the possible expense of liveness.

— **Terminate** the current thread (stop). This preserves safety, though brutally! (Not recommended.)

— **Exit** the method, possibly raising an exception. This preserves liveness but may require the caller to take special action to preserve safety.

— **Cleanup** and restart.

— **Ask** for user intervention before proceeding.

*Interrupts can be useful to signal that the guard can never become true because, for example, the collaborating threads have terminated.*
Step: Signal state changes

> Add *notification code* to each method of the class that changes state in any way that can affect the value of a guard condition. Some options are:

— Use `notifyAll` to *wake up all threads* that are blocked in waits for the host object.

— Use `notify` to wake up only one thread (if any exist). This is best treated as an *optimization* where:

  – *all blocked threads are necessarily waiting for conditions signalled by the same notifications,*
  – *only one of them can be enabled by any given notification,* and
  – *it does not matter which one of them becomes enabled.*

— You build your own special-purpose notification methods using `notify` and `notifyAll`. (For example, to selectively notify threads, or to provide certain *fairness guarantees.*)
Testing for safety violations

public abstract class BoundedCounterAbstract
    implements BoundedCounter {
    protected long count = MIN;
    private int errors = 0;

    protected void checkInvariant() {
        if (! (count >= BoundedCounter.MIN && count <= BoundedCounter.MAX) ) {
            errors++;
        }
    }

    public int errors() {
        return errors;
    }
}

Common behaviour to help us test for safety violations
public class BoundedCounterBasic
    extends BoundedCounterAbstract {...
public synchronized void inc() {
    while (count >= MAX) {
        try { wait(); }
        catch(InterruptedException ex) { }
    }
    count ++;
    notifyAll();
    checkInvariant(); // record safety violations
}
...
}
Race conditions

```
public class BoundedCounterNoSyncBAD
    extends BoundedCounterAbstract {
public void inc() { // missing synchronization
    while (count >= MAX) {
        Thread.yield();
    }
    Thread.yield(); // race condition here
    count ++;
    checkInvariant(); // possible safety violation
}
}
```

NB: wait() and notify() are invalid outside synchronized code!
Careless use of `notify()` may lead to *race conditions*.

Now both A and C wait for nothing!
Ensure that each wait is balanced by at least one notification. Options include:

| **Blanket Notifications** | Place a *notification at the end of every method* that can cause any state change (i.e., assigns any instance variable). Simple and reliable, but may cause performance problems ...

| **Encapsulating Assignment** | *Encapsulate assignment to each variable* mentioned in any guard condition in a helper method that performs the notification after updating the variable.

| **Tracking State** | Only issue notifications for the *particular state changes* that could actually unblock waiting threads. May improve performance, at the cost of flexibility (i.e., subclassing becomes harder.)

| **Tracking State Variables** | Maintain an *instance variable that represents control state*. Whenever the object changes state, invoke a helper method that re-evaluates the control state and will issue notifications if guard conditions are affected.

| **Delegating Notifications** | Use *helper objects to maintain aspects of state* and have these helpers issue the notifications. |
Encapsulating assignment

Guards and assignments are encapsulated in helper methods:

```java
public class BoundedCounterEncapsulatedAssigns extends BoundedCounterAbstract {
    ...
    public synchronized void inc() {
        awaitIncrementable();
        setCount(count + 1);
    }
    public synchronized void dec() {
        awaitDecrementable();
        setCount(count - 1);
    }
    ...
```
protected synchronized void awaitIncrementable() {
    while (count >= MAX)
        try {
            wait();
        }
        catch(InterruptedException ex) {};
}

protected synchronized void awaitDecrementable() {
    while (count <= MIN)
        try {
            wait();
        }
        catch(InterruptedException ex) {};
}

protected synchronized void setCount(long newValue) {
    count = newValue;
    notifyAll();
}
The only transitions that can possibly affect waiting threads are those that step away from logical states top and bottom:

```java
public class BoundedCounterTrackingState
    extends BoundedCounterAbstract {

    ... 
    public synchronized void inc() {
        while (count == MAX) {
            try { wait(); } 
            catch(InterruptedException ex) {}; 
            if (count++ == MIN) 
                notifyAll(); // just left bottom state 
        }
    }
    ... 
}
```
public class BoundedCounterStateVariables extends BoundedCounterAbstract {
    protected enum State { BOTTOM, MIDDLE, TOP }
    protected State state = State.BOTTOM;

    public synchronized void inc() {
        while (state == State.TOP) { // consult logical state
            try { wait(); } 
            catch(InterruptedException ex) {};
        }
        ++count; // modify actual state
        checkState(); // sync logical state
    }
...

    }
... protected synchronized void checkState() {
    State oldState = state;
    if (count == MIN) { state = State.BOTTOM; }
    else if (count == MAX) { state = State.TOP; }
    else { state = State.MIDDLE; }

    if (leftOldState(oldState)) { notifyAll(); }
}

private boolean leftOldState(State oldState) {
    return state != oldState
        && (oldState == State.TOP
            || oldState == State.BOTTOM);
}
Delegating notifications

```java
public class NotifyingLong {
  private long value;
  private Object observer;
  public NotifyingLong(Object o, long v) {
    observer = o; value = v;
  }
  public synchronized long value() { return value; }
  public void setValue(long v) {
    synchronized(this) { // NB: partial synchronization
      value = v;
    }
    synchronized(observer) { // NB: must be synchronized!
      observer.notifyAll();
    }
  }
}
```
Delegating notifications ...

*Notification is delegated to the helper object:*

```java
public class BoundedCounterNotifyingLong implements BoundedCounter {
    private NotifyingLong count = new NotifyingLong(this, MIN);
    public synchronized long value() { return count.value(); }
    public synchronized void inc() {
        while (count.value() >= MAX) {
            try { wait(); }
            catch(InterruptedException ex) {};
        }
        count.setValue(count.value()+1); // issues notification
    }
    ...
}
```
What kinds of liveness problems can occur in concurrent programs?

Why is progress a liveness rather than a safety issue?

What is fair choice? Why do we need it?

What is a terminal set of states?

What are necessary and sufficient conditions for deadlock?

How can you detect deadlock? How can you avoid it?
Can you answer these questions?

> How would you manually check a progress property?
> What is the difference between starvation and deadlock?
> How would you manually detect a waits-for cycle?
> What is fairness?
What you should know!

> When can you apply the Guarded Methods pattern?
> When should methods recheck guard conditions after waking from a wait()?
> Why should you usually prefer notifyAll() to notify()?
> When and where should you issue notification?
> Why must you re-establish the class invariant before calling wait()?
> What should you do when you receive an InterruptedException?
> What is the difference between tracking state and using state-tracking variables?
Can you answer these questions?

> When are guarded methods better than balking?
> When should you use helper methods to implement guarded methods?
> What is the best way to structure guarded methods for a class if you would like it to be easy for others to define correctly functioning subclasses?
> When is the complexity of delegating notifications worthwhile?
License

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.
Nothing in this license impairs or restricts the author's moral rights.

http://creativecommons.org/licenses/by-sa/3.0/

© Oscar Nierstrasz