COMP31212: Concurrency

Topic 5.3: Liveness and Topic 5.4 Fairness
Outline

Topic 5.3: Liveness Properties
  Progress Properties
  Priority
  Example - Single Lane Bridge again
  Java Implementation for Fair Bridge

Topic 5.4: Fairness and Starvation
  Readers and Writers Problem
  Properties for Reader/Writers
  Java implementation
Outline

**Topic 5.3: Liveness Properties**

- Progress Properties
  - Priority
  - Example - Single Lane Bridge again
  - Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**

- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
Making Progress

- **Liveness**: something good *eventually* happens
- **Progress**: an action always *eventually* gets executed
- **Fair Choice**: if a choice over a set of transitions is made infinitely often, then every transition in the set will be chosen infinitely often

FSP uses *Fair Choice* by default.

\[
\text{progress } P = \{a_1, \ldots, a_n\}
\]

defines Progress Property \( P \):

*At least one of the actions \( a_1, \ldots, a_n \) will be executed infinitely often.*
Example: Coins

TWOCOIN = ( pick \rightarrow COIN
        \mid pick \rightarrow TRICK ),

TRICK = ( toss \rightarrow heads \rightarrow TRICK ),

COIN = ( toss \rightarrow heads \rightarrow COIN
        \mid toss \rightarrow tails \rightarrow COIN ).

progress HEADS = \{heads\}
progress TAILS = \{tails\}
progress HEADSorTAILS = \{heads, tails\}

Which of these progress properties hold for COIN or for TWOCOIN?
LTS for TWOCOIN
Graph theory: Strongly connected components

From graph theory:

**Definition.** Two nodes $m$ and $n$ in a graph are **strongly connected** if there is a path from $m$ to $n$ and a path from $n$ to $m$.

A set of nodes in a graph is strongly connected if every pair of nodes in the set is strongly connected.

**Definition.** A **strongly connected component** of a graph is a maximal strongly connected set (i.e. a set that cannot be extended with other nodes and still remain strongly connected).

A strongly connected component is **terminal** if there are no edges out of it.
Terminal Sets

A terminal set of states is one where every state is reachable from every other, and no transition to any state outside terminal set.

i.e. terminal set = terminal strongly connected component

Key idea: A progress property is violated if there is a terminal set in which none of the progress set actions appear.

Note: In LTSA tool, if no progress properties are stated, the default progress property is each action as a separate progress property.
Terminal strongly connected components and fairness

There are three strongly connected components: (a) the node 0, (b) the nodes 1 and 2, and (c) the nodes 3, 4 and 5. (b) and (c) are terminal strongly connected components.

**Question:** Is TWOCOIN fair for tails?

**NO** - the system can enter a state in which all infinite traces don’t contain tails - the terminal strongly connected component (b).
Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
  - Example - Single Lane Bridge again
  - Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
Adding Priority to Actions

**High Priority:** $P \ll \{a_1, \ldots, a_n\}$

If they occur in a choice, do $a_1, \ldots, a_n$ in preference to other actions.

**Low Priority:** $P \gg \{a_1, \ldots, a_n\}$

If they occur in a choice, do other actions in preference to $a_1, \ldots, a_n$

i.e. discard lower priority actions within a choice.

*Utilise to uncover potential progress problems by selecting traces which may be problematic...*
Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
- Example - Single Lane Bridge again
- Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
Congestion on the Single-Lane Bridge

progress BLUECROSS = { blue[ID].enter }
progress REDCROSS = { red[ID].enter }

||CongestedBridge = SingleLaneBridge
    >> { red[ID].exit, blue[ID].exit }.

We thus make exit of cars from the bridge a low priority, so that in a choice of entering and exiting, cars enter.

★ What is result? ★ ★ How can we fix it? ★
First Attempt

CAR = (request->enter->exit->CAR).

BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
(red[ID].request → BRIDGE[nr][nb][wr+1][wb]
|when (nb==0 && wb==0)
        red[ID].enter  → BRIDGE[nr+1][nb][wr-1][wb]
|red[ID].exit    → BRIDGE[nr-1][nb][wr][wb]
|blue[ID].request → BRIDGE[nr][nb][wr][wb+1]
|when (nr==0 && wr==0)
        blue[ID].enter  → BRIDGE[nr][nb+1][wr][wb-1]
|blue[ID].exit      → BRIDGE[nr][nb-1][wr][wb] )

★ Why does this not work? ★
Answer

The system deadlocks. If there are cars waiting at both ends then no cars enter the bridge!
Revised Version

BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =

(red[ID].request -> BRIDGE[nr][nb][wr+1][wb][bt]
|when (nb==0 && (wb==0 || !bt))
  red[ID].enter -> BRIDGE[nr+1][nb][wr-1][wb][bt]
  red[ID].exit -> BRIDGE[nr-1][nb][wr][wb][True]
|blue[ID].request->BRIDGE[nr][nb][wr][wb+1][bt]
|when (nr==0 && (wr==0 || bt))
  blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1][bt]
  blue[ID].exit -> BRIDGE[nr][nb-1][wr][wb][False]
).

Here we introduce an asymmetry to break the deadlock. There is a boolean variable bt which indicates whether it is the turn of blue cars or red cars to enter the bridge. Initially bt is set to true to indicate it is a blue car’s turn.
Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
- Example - Single Lane Bridge again
- Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
Java for a Fair Bridge

class FairBridge extends Bridge {

    private int nred = 0; private int nblue=0;
    private int waitblue=0; private int waitred=0;
    private boolean blueturn = true;

    synchronized void redEnter()
        throws InterruptedException {
        ++waitred;
        while (nblue>0
                || (waitblue>0 && blueturn)) wait();
        --waitred;
        ++nred; }

    synchronized void redExit(){
    --nred;
    blueturn = true;
    if (nred==0)
        notifyAll(); }

synchronized void blueEnter() throws InterruptedException {
    ++waitblue;
    while (nred>0 || (waitred>0 && !blueturn)) wait();
    --waitblue;
    ++nblue;
}

synchronized void blueExit(){
    --nblue;
    blueturn = false;
    if (nblue==0)
        notifyAll();
}
Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
- Example - Single Lane Bridge again
- Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
A Readers/Writers System: Fairness and Starvation

- Database access and update
- Several reader and writer processes
- Simultaneous access where possible: multiple read, exclusive write
- Avoid interference
- Fairness, progress
FSP Modelling aspects

Processes:
- Readers
- Writers
- Database

Properties:
- **Safety**: No Readers have access when a Writer has access
- **Safety**: Only one Writer has access at a time
- **Progress**: Any Reader (waiting for access) will eventually gain access
- **Progress**: Any Writer (waiting for access) will eventually gain access
An Abstract Model

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

READER = ( acquireRead \rightarrow releaseRead \rightarrow READER ) + Actions.
WRITER = ( acquireWrite \rightarrow releaseWrite \rightarrow WRITER ) + Actions.

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] =
  ( when ( !writing )
    acquireRead \rightarrow RW[readers+1][writing]
  | releaseRead \rightarrow RW[readers-1][writing]
  | when ( readers==0 && !writing )
    acquireWrite \rightarrow RW[readers][True]
  | releaseWrite \rightarrow RW[readers][False]
).

Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
- Example - Single Lane Bridge again
- Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
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 ).

progress WRITE = {writer[i..Nwrite].acquireWrite}
progress READ = {reader[i..Nread].acquireRead}
Checking Safety

||READWRITELOCK = (RW_LOCK || SAFE_RW).

We now consider the parallel composition of a number of READERS, a number of WRITERS and the above process:

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

\[
\text{progress WRITE} = \{\text{writer[i..Nwrite].acquireWrite}\}
\]
\[
\text{progress READ} = \{\text{reader[i..Nread].acquireRead}\}
\]
\[
||\text{RW_PROGRESS} = \text{READERS_WRITERS}
\]
\[
\gg \{ \text{reader[1..Nread].releaseRead, writer[1..Nread].releaseWrite} \}.
\]

Under this priority, we find that the progress properties do not hold. For example, the readers can gain exclusive priority to the resource and writers cannot then gain access.
No Writer-Starvation Read/Write Lock

One solution: Readers cannot gain access if writers are waiting. To model this, add requestWrite action:

```plaintext
READER = (acquireRead -> releaseRead -> READER) + Actions.
WRITER = (requestWrite -> acquireWrite -> releaseWrite -> WRITER) + Actions.
```

```plaintext
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][False][waitingW+1]).
```

This ensures writers never starve, but they may exclude readers.
Fair Read/Write Lock

To ensure that both progress properties hold, we may use a scheduling algorithm. For example, we could add a boolean variable to indicate whose turn it is and then readers may still defer to writers, but only if it is not their turn:

```plaintext
RW_LOCK = RW[0][False][0][False],
RW[readers:0..Nread][writing:Bool]
[waitingW:0..Nwrite][readersturn:Bool] =
  (when (!writing && (waitingW==0||readersturn))
    acquireRead -> RW[readers+1][writing][waitingW][readersturn]
    | releaseRead -> RW[readers-1][writing][waitingW][False]
    | when ( readers==0 && !writing )
    acquireWrite -> RW[readers][True ][waitingW-1][readersturn]
    | releaseWrite -> RW[readers][False][waitingW][True]
    | requestWrite -> RW[readers][writing][waitingW+1][readersturn]
  ).
```
Outline

**Topic 5.3: Liveness Properties**
- Progress Properties
- Priority
- Example - Single Lane Bridge again
- Java Implementation for Fair Bridge

**Topic 5.4: Fairness and Starvation**
- Readers and Writers Problem
- Properties for Reader/Writers
- Java implementation
Java Implementation: Safe

class ReadWriteSafe implements ReadWrite {
    private int readers = 0;
    private boolean writing = false;

    public synchronized void acquireRead() {
        while (writing) wait();
        ++readers;
    }

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

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

Java Implementation: No Writer Starvation

class ReadWritePriority implements ReadWrite{
    private int readers = 0;
    private boolean writing = false;
    private int waitingW = 0;

    public synchronized void acquireRead() throws InterruptedException {
        while (writing || waitingW > 0) wait();
        ++readers;
    }

    public synchronized void releaseRead() {
        --readers;
        if (readers == 0) notify();
    }
}
public synchronized void acquireWrite() throws InterruptedException {
    ++waitingW;
    while (readers>0 || writing) wait();
    --waitingW; writing = true; }

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