Simple Lock Program and Service

Shankar

September 18, 2014

Outline

Program SimpleLock(N) overview

- Lock for threads 0, ⋯, N−1 // N ≥ 1
  - input functions: acq(), rel(), end()
  - non-input function: serve()
- Main
  - bool xreq[N]: xreq[i] true iff user i has ongoing request
  - bool xacq: true iff a user holds the lock
  - start thread executing serve()
- Function serve()
  - cycle through entries of xreq
  - if xreq[j] true: set xacq, unset xreq[j], wait for xacq false
- Input functions
  - acq(): set xreq[mytid], wait for it to be false; return
  - rel(): unset xacq; return
  - end(): execute endSystem(); return

```
program SimpleLock(int N) {
  ia {N ≥ 1}
  boolean[N] xreq ← false;
  boolean xacq ← false;
  int xp ← 0;
  Tid t ← startThread(serve());
  return mysid;

  function void serve() {
    while (true)
  a0: if  • (xreq[xp])
  a1:   • xacq ← true;
  a2:   • xreq[xp] ← false;
  a3:      while  • (xacq) skip;
  a4: xp ← mod(xp+1, N);
  }
```

Note the •'s

- ignore them for now
- later we refer to them as "atomicity breakpoints"

```
 input void mysid.acq()
    ia {mytid in 0..N−1}
a5: xreq[mytid] ← true;
a6: while • (xreq[mytid]) skip;
    return;
 }

 input void mysid.rel() {
    ia {mytid in 0..N−1}
a7: xacq ← false;
    return;
 }

 input void mysid.end() {
    ia {true}
    endSystem();
 }
```

```
atomicity assumption:
   reads and writes of
   xacq,
   xreq[0], ..., xreq[N−1]

progress assumption:
   weak fairness
   for threads
}
```

Some Properties of SimpleLock(N)

- Input assumptions of acq() and rel() are "weak"
  - only require caller tid to be in 0..N−1
  - allow acq() caller to hold lock
  - allow rel() caller to not hold lock
- Hence the program has some odd allowed evolutions
  - e.g., two users hold lock simultaneously

  [but it does implement SimpleLockService]
- Input assumptions are sufficient to ensure following
  - SimpleLock(N) is fault-free
    // no allowed evolution is faulty
  - the •'s are a valid set of atomicity breakpoints
    // code between two successive •'s is effectively atomic

Outline

Simple Lock Service Overview

- Lock for threads 0, ⋯, N−1
- Main
  - vars indicating: whether ending; which user (if any) has lock
- Input functions `acq()`, `rel()`, `end()`
- No output function
- Defines all acceptable io sequences
- Constrains both environment and lock, e.g.,
  - `acq.ia`: not ending, caller in `0..N−1`, does not hold lock
- Atomicity assumptions: input parts and output parts
- Progress assumptions:
  - `acq()` returns eventually if lock becomes repeatedly free
  - `rel()` and `end()` each returns eventually

```
service SimpleLockService(int N)  {
  ic {N ≥ 1}
  boolean[N] acqd ← false;   // acqd[i] true iff i has lock
  ending ← false;                    // termination initiated
  return mysid;

  input void mysid.acq() {
    ic {not ending and (mytid in 0..N−1) and not acqd[mytid]}
    oc {forall(j in 0..N−1: not acqd[j])}
    acqd[mytid] ← true;
    return;
  }
```

Program SimpleLockService – 2


```
input void mysid.rel() {
  ic {not ending and (mytid in 0..N−1) and acqd[mytid]}
  acqd[mytid] ← false;
  oc {true}
  return;
}

input void mysid.end() {
  ic {not ending}
  ending ← true;
  oc {true}
  return;
}
```

```
atomicity assumption {input parts and output parts}

progress assumption {
   // rel returns
   forall(i:(i in mysid.rel)  leads-to  (not i in mysid.rel));

   // if no one holds the lock forever then acq returns
   forall(i: acqd[i]  leads-to  not acqd[i]) ⇒
     forall(i: (i in mysid.acq)  leads-to  (not i in mysid.acq));

   // end returns
   forall(i:(i in mysid.end)  leads-to  (not i in mysid.end));
  }
}
```

- Convention: i, j range over 0..N−1

Observations on SimpleLockService(N)

- Program is fault-freee
  - otherwise it's useless as a service
- Atomicity breakpoints at (and only at) output conditions
  - natural consequence of atomicity assumptions
- Progress stated by leads-to (and not fairness) assertions
- Comparing against SimpleLock
  - input conditions stronger than SimpleLock's input assumptions
  - so precludes some ("odd") evolutions of SimpleLock
  - has io sequences not achievable by SimpleLock(N)

Outline

Overview: Lock Implements Service

- Define lock-service inverse program
  - most general environment for a lock implementation
- Define program Z:
  - concurrently executes implementation and service inverse
- Define the assertions that Z must satisfy
  - safety: Z satisfies inverse's input conditions
  - progress: Z inverse's progress assertions
- Prove that Z satisfies above assertions

Outline

```
service SimpleLockService(int N) {
program SimpleLockServiceInverse(int N, Sid lck) {
  // lck: lock system being tested
  ic {N ≥ 1}
  boolean[N] acqd ← false;
  ending ← false;
  return mysid;

  input void mysid.acq() {
  output doAcq() {
     ic oc {not ending and (mytid in 0..N−1) and not acqd[mytid]
     lck.acq();
     oc ic {forall(j in 0..N−1: not acqd[j])}
     acqd[mytid] ← true;
     return;
  }
```

SimpleLockService Inverse — 2


```
output doRel() input void mysid.rel() {
   ic oc {not ending and (mytid in 0..N−1) and acqd[mytid]}
   acqd[mytid] ← false;
   lck.rel();
   oc ic {true}
   return;
}

input void mysid.end() {
output doEnd() {
   ic oc {not ending}
   ending ← true;
   lck.end();
   oc ic {true}
   return;
}
```

```
atomicity assumption {input parts and output parts}

progress ~~assumption~~ condition {
  forall(i: (i in ~~mysid~~ lck.rel)
                  leads-to  (not i in ~~mysid~~ lck.rel));

  forall(i: acqd[i]  leads-to  not acqd[i]) ⇒
    forall(i: (i in ~~mysid~~ lck.acq)
                  leads-to  (not i in ~~mysid~~ lck.acq));

  forall(i: (i in ~~mysid~~ lck.end)
                  leads-to  (not i in ~~mysid~~ lck.end));
 }
}
```

Outline

Program of lock and inverse service

```
program Z(int N) {
  ic {N ≥ 1}
  inputs(); outputs();                // aggregate sys-
tem Z is closed

  Sid lck ← startSystem(SimpleLock(N));

  Sid lsi ← startSystem(SimpleLockServiceInverse(N, lck));

  return mysid;

  atomicity assumption {}
  progress assumption {weak fairness}
}
```

Assertions that Z(N) must satisfy

B_0 : *Inv* [(i at lsi.doAcq.ic) ⇒ forall(j: not acqd[j])]

B_1 : (i in lck.rel) *leads-to* (not i in lck.rel)

B_2 : forall(i: acqd[i] *leads-to* not acqd[i]) ⇒
forall(i: (i in lck.acq) *leads-to* (not i in lck.acq))

B_3 : (i in lck.end) *leads-to* (not i in lck.end)

- Recall conventions
  - i, j range over 0..N−1
  - free variables are universally quantified
    e.g., B_3 equivalent to forall(i: B_3)

Outline

Effective atomicity of Z(N)


```
system lck(N)
  <main>
  fn serve(){...••···...}
  input acq(){...•...}
  input rel(){...}
  input end(){...}
```

```
system lsi(N,lck)
  <main>

  output doAcq(){•oc ...}
  output doRel(){•oc ...}
  output doEnd(){•oc ...}
```

- step Z.init: Z.main, lck.init, lsi.main
- step doAcq.call: lsi.doAcq.oc• ⟶ lck.acq•
- step acq.ret: lck.acq• ⟶ lsi.doAcq.end
- step doRel: lsi.doRel.oc• ⟶ lck.rel ⟶ lsi.doRel.end
- step doEnd: lsi.doEnd.oc• ⟶ lck.end ⟶ lsi.doEnd.end
- steps in lck.serve() defined by its •'s
  - valid in Z because lck gets only allowed inputs (from lsi)

Proof of safety condition: B_0

- Recall B_0: if thread at `doAcq.ic` then every `acqd[j]` is false
- Given Z's effective atomicity, B_0 is equivalent to $Inv\ C_0$

C_0 : ((i on lck.acq•) and not lck.xreq[i]) $\Rightarrow$
    forall(j: not lsi.acqd[j])

- $Inv\ C_1$ and $Inv\ C_2$ hold // operational reasoning

C_1 : (lck.alive and (not t on a3)) $\Rightarrow$ forall(j: not acqd[j])

C_2 : (t on a3) $\Rightarrow$
    ((acqd[xp] or
      (not acqd[xp] and (xp on a6) and not xreq[xp]))
    and forall(j, j $\neq$ xp: not acqd[j]))

- $Inv\ C_0$ holds from $Inv\ C_1$ and $Inv\ C_2$ // operational reasoning

Proof of progress condition: B_1 and B_3

- Recall B_1: thread in `lck.rel` eventually leaves `lck.rel`
- B_1 holds
  - `lck.rel`.body has no loops and no blocking
  - thread has weak fairness (from `lck` progress assumption)
- Recall B_3: thread in `lck.end` eventually leaves `lck.end`
- B_3 holds just like B_1

Proof of progress condition: B_2 – 1

- Recall B_2: $D_0 \Rightarrow D_1$, where

  D_0 : `acqd[i]` *leads-to* `not acqd[i]`

  D_1 : `(k in lck.acq)` *leads-to* `(not k in lck.acq)`

- We will establish the following

  D_2 : [t at a0, xp = j, j in lck.acq] leads to
  [xp not in lck.acq]

  D_4 : [t at a0, xp = j] leads to
  [t at a0, xp = mod(j+1,N)]

- D_2 and D_4 imply D_1

- We establish

  D_2 : ((t on a0) and xp = j and xreq[j]) *leads-to*
  ((t on a3) and xp = j and acqd[j])

- Proof
  - "j in lck.acq" equivalent to "j at a6" // Z's atomicity
  - [j at a6, t at a0, xp = j] leads to // via wfair t
    [j at a6, t at a3, xreq[j] is false] leads to // via wfair j
    [j not at a6, t at a3, xreq[j] is false]

Proof of progress condition: B_2 – 3

- We establish

  D_3 : ((t on a3) and xp = j and acqd[j]) *leads-to*
  ((t on a0) and xp = mod(j+1,N))

- Proof
  - [D_2's rhs] leads to // via D_0, j.doRel

    [xacq false, t on a3] leads to // via wfair t

    [t at a0, xp is mod(j+1,N)]

- D_2 and D_3 imply

  D_4 : ((t on a0) and xp = j) *leads-to*
  ((t on a0) and xp = mod(j+1,N))

Assertional Proof of B_0–B_3

- See text

Outline

Overview

- Program ProdCons1
  - start systems: producer, consumer, lock service
  - producer and consumer use lock service
- Show that ProdCons1 is fault-free
  - show that it satisfies input conditions of lock service system
- Obtain atomicity breakpoints // effective atomicity
- Establish desired properties
  - still hold when lock service is replaced by a lock implementation

```
program ProdConsLck(...) {
   ia {...}
   <hide lck inputs>;
   lck ← startSystem(SimpleLockService());
   cons ← startSystem(Consumer(lck));
   prod ← startSystem(Producer(lck, cons));
   return [0, mysid];

   atomicity assumption {}  // none
   progress assumption {weak fairness}
}
```

Program Producer-Consumer-Lock – 2

using lock service

```
SimpleLockService(N):
 ...
 input mysid.acq():
    ic {...}
  ● oc {...}
 input mysid.rel():
   ...
 input mysid.end():
   ...
```

```
Consumer(lck):
 start-
Thd(consum());
 fn consum():
  while (...)
    lck.acq();
    ...
    lck.rel();
  lck.end();
  endSystem();
 input mysid.put():
  ...
```

```
Producer(lck,cons):
  start-
Thd(prod());
  fn produce():
    while (...)
        lck.acq();
        cons.put();
        lck.rel();
    endSystem();
```

- Single atomicity breakpoint in entire program text
  - ProdCons.init: start $\longrightarrow$ only 2 threads at lck.acq
  - cons.step: lck.acq $\longrightarrow$ lck.acq or exit
  - prod.step: lck.acq $\longrightarrow$ lck.acq or exit