Bounded Buffer

Shankar

September 18, 2014

Overview

- Bounded-buffer service
  - input functions: `void put(x)`, `Val get()`, `void end()`
  - no output functions
- Implementations using standard synchronization constructs
  - locks, condition variables
  - semaphores
- Implementations using await synchronization constructs
  - more powerful, convenient
  - mechanical transformation to standard synch constructs
- Reduce blocking (increase parallelism) in implementations
- Cancelling blocked calls to remote systems
  - allows a caller no longer interested in call to retrieve itself

Outline

Bounded-Buffer Service – 1

- Fifo bounded buffer of size N
  - input functions: put(x), get(), end()
  - no output functions
- Main
  - buff: sequence of items in buffer
  - ending: true iff end() has been called
  - putBusy: true iff put call ongoing
  - getBusy: true iff get call ongoing
- void mysid.put(x)
  - ic {not ending and no ongoing put call}
  - oc {buff has space}
    append x to buff; return

Bounded-Buffer Service – 2

- `Val mysid.get()`
  - ic {not ending and no ongoing get call}
  - output `rval`
    oc {buff has item, `rval` is `buff.head`}
    behead `buff`; return `rval`

- `void mysid.end()`
  - ic {not ending}
    set ending
  - oc {true} return

- Progress assumption
  - put call returns if `buff` has space // uses `putBusy`
  - get call returns if `buff` has item // uses `getBusy`
  - end call returns // uses "thread in `mysid.end`"

```
// main
ic {N ≥ 1}
buff ← [];
ending ← false;
putBusy ← false;
getBusy ← false;
return mysid;

input void mysid.put(Val x)
  ic {not ending and
      not putBusy}
  putBusy ← true;
  oc {buff.size < N}
  buff.append(x);
  putBusy ← false;
  return;
```

```
input Val mysid.get()
  ic {not ending and
      not getBusy}
  getBusy ← true;
  output(Val rval)
   oc {buff.size > 0
       and rval = buff[0]}
   buff.remove();
   getBusy ← false;
   return rval;

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

BoundedBuffer(int N) – 2


```
    atomicity assumption {input and output parts}

    progress assumption {
       // thread in put returns if buffer has space
       (putBusy and buff.size < N)  leads-to  not putBusy;

       // thread in get returns if buffer has an item
       (getBusy and buff.size > 0)  leads-to  not getBusy;

       // thread in end returns
       (thread u in mysid.end)  leads-to  (not u in mysid.end);
    }
}
```

Outline

- BoundedBufferInverse(N, bb)   // bb: sid of implementation
  - main: buff, ending, putBusy, getBusy
  - output functions: doPut(x), doGet(), doEnd()

- doPut(x)
  - oc {not ending and no ongoing put call}
    bb.put(x)
  - ic {buff has space}
    append x to buff

- doGet()
  - oc {not ending and no ongoing get call}
    rval ← bb.get()
  - ic {buff has item, rval is buff.head}
    behead buff

Service Inverse Overview – 2

- doEnd()
  - oc {not ending}
    set ending; bb.end()
  - ic {true}   return
- progress condition
  - put call returns if buff has space   // uses putBusy
  - get call returns if buff has item   // uses getBusy
  - end call returns   // uses "thread in bb.end"

BoundedBufferInverse(N, bb)

service inverse

```
// main
ic {N ≥ 1}
buff ← [];
putBusy ← false;
getBusy ← false;
return mysid;

output doPut(Val x) {
  oc {not end-
ing and
        not putBusy}
  putBusy ← true;
  bb.put(x);
  ic {buff.size < N}
  buff.append(x);
  putBusy ← false;
```

```
output doGet() {
  oc {not ending and not getBusy}
  getBusy ← true;
  Val x ← bb.get(x);
  ic {buff.size > 0 and x = buff[0]}
  buff.remove();
  getBusy ← false;

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

progress condition {... ~~mysid~~ bb...}

Outline

Await construct

- Await: powerful synchronization construct not provided by PLs
  - convenient for writing progams
  - implementable by standard synchronization constructs
- `await (B) S`
  - B is predicate, S is non-blocking code
  - atomically execute S only if B holds, otherwise wait
  - weak or strong fairness
  - `await (B) S`: more general than `oc {B} S`
    - S can make (non-blocking) output calls, use return values
- `atomic S`: short for `await (true) S`

Await-structured programs

- Await-structured program
  - awaits are the only synchronization construct
  - code outside awaits does not conflict with code executed by other threads
- Await-structured program
  - easier to understand than equivalent program with standard synchronization constructs
  - can be mechanically transformed to program that uses standard synchronization constructs

Outline

Implementation BBuffAwait(N)


```
program BBuffAwait(int N) {
  ia {N ≥ 1}
  Seq buff ← seq();
  return mysid;
```

```
input void mysid.put(Val x)
• await (buff.size < N)
      buff.append(x);
    return;

input Val mysid.get()
• await (buff.size > 0) {
      Val x ← buff[0];
      buff.remove();
      return x;
```

```
input void mysid.end()
    endSystem();
    return;
```

atomicity assumption
    awaits

progress assumption
    weak fairness
    for threads

BBuffAwait implements BoundedBuffer: Z

- program Z(int N) {

```
  ...
  bb ← startSystem( BBuffAwait(N) );
  si ← startSystem( BoundedBufferInverse(N,bb) );
  ...

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

BBuffAwait implements BoundedBuffer: Safety

- To not violate `si.doPut.ic`, want $Inv\ C_0$ to hold

C_0 : ((thread in bb.put) and bb.buff.size < N) $\Rightarrow$
    si.buff.size < N

- To not violate `si.doGet.ic`, want $Inv\ C_1$ to hold

C_1 : ((thread at bb.get) and bb.buff.size > 0) $\Rightarrow$
    (si.buff.size > 0 and bb.buff[0] = si.buff[0])

- Hold because $Inv\ C_2$ holds (via invariance rule)

C_2 : bb.buff = si.buff

BBuffAwait implements BoundedBuffer: Progress

- Want B_2–B_4 to hold

  B_2 : (putBusy and si.buff.size < N) *leads-to* not putBusy

  B_3 : (getBusy and si.buff.size > 0) *leads-to* not getBusy

  B_4 : (thread u in bb.end) *leads-to* (not u in bb.end)

- B_2 holds via weak fairness and *Inv* C_2
  - only a thread in bb.put can falisify B_2.lhs
  - only one such thread at any time // doPut.oc
  - so it eventually executes, establishing B_2.rhs // wfair
- B_3 holds via weak fairness and *Inv* C_2
- B_3 holds via weak fairness

Outline

Locks

- A lock is either acquired by a thread or free
- `Lock lck`: // initially free
- `lck.acq()`: // acquire
  - caller must not hold lock
  - atomically acquire `lck` only if free, o/w wait
- `lck.rel()`: // release
  - caller must hold lock
  - atomically free `lck`
- Progress: a thread at `lck.acq()` eventually gets past if
  - lock is free continuously // wfair; weak lock
  - lock is free continuously or repeatedly // sfair strong lock

Condition Variables

- `Condition(lck) cv`: // cond var cv associated with lock lck
- `cv.wait()`: // always blocks
  - caller must hold `lck`
  - atomically release `lck` and wait on cv;
    when awakened: acquire `lck`; return
- `cv.signal()`:
  - caller must hold `lck`
  - atomically awaken a thread (if any) waiting on cv; return
- Progress: a thread at `cv.wait()` eventually gets past if
  - cv is signalled, and no other process is waiting on cv // weak
  - cv is repeatedly signalled // strong

Implementing Await with Lock-Cv

- Await-structured program with distinct await guards $B_1, \cdots, B_N$
  - works even if guards are not distinct
- Introduce `lck` and associated $\mathtt{cv}_1, \cdots, \mathtt{cv}_N$
- Replace `await` (B_i) S by

```
lck.acq()
while (not Bi)
   cvi.wait()
S
for k in 1,...,N
   if (Bk)
      cvk.signal()
lck.rel()
```

- For more parallelism
  - partition awaits into “non-conflicting” groups
  - use separate lock for each group

Outline

Implementation BBuffLockCv – 1


```
program BBuffLockCv(int N) {
   ia {N ≥ 1}
   Seq buff ← [];
   Lock lck;                    // protects buffer
   Condition(lck) cvItem;       // signaled when buffer not empt
   Condition(lck) cvSpace;      // signaled when buffer not full
   return mysid;

   input void mysid.put(Val x)
      lck.acq();                // Note: no '•'
      while (buff.size = N)
       • cvSpace.wait();
      buff.append(x);
      cvItem.signal();
      lck.rel();
      return;
```

Implementation BBuffLockCv – 2


```
    input Val mysid.get()
       lck.acq();                    // Note: no '•'
       while (buff.size = 0)
        • cvItem.wait();
       Val x ← buff[0];
       buff.remove();
       cvSpace.signal();
       lck.rel();
       return x;

    input void mysid.end()
       endSystem();
       return;

   atomicity assumption {lck, cvItem, cvSpace}
   progress assumption {weak fairness for threads}
}
```

Outline

Semaphores

- Combines mutual exclusion + conditional wait
- Counting semaphores
  - `Semaphore(N) sem`: sem initialized to N $\geq$ 0
  - `sem.P()`: atomically sem−− only if sem > 0, o/w wait
  - `sem.V()`: atomically sem++
- Binary semaphores
  - `Semaphore(N) sem`: sem initialized to N in 0..1
  - `sem.P()`: atomically sem $\leftarrow$ 0 only if sem = 1, o/w wait
  - `sem.V()`: atomically sem $\leftarrow$ 1
- Progress: condition in which a thread at `P()` eventually gets past
  - sem > 0 holds continuously // wfair; weak sem
  - sem > 0 continuously or intermittently // sfair; strong sem

Implementing Lock-cv with Semaphore

- Program with locks and condition variables
- For every lock `lck`
  - introduce binary semaphore, say `lckMutex`, initialized to 1
  - `lck.acq()` $\longrightarrow$ `lckMutex.P()`
  - `lck.rel()` $\longrightarrow$ `lckMutex.V()`
- For every condition variable `cv` associated with lock `lck`
  - introduce binary semaphore, say `cvGate`, initialized to 0
  - `cv.wait()` $\longrightarrow$ `lckMutex.V(); cvGate.P(); lckMutex.P()`
  - `cv.signal()` $\longrightarrow$ `cvGate.V()`
- To have waiting thread come before entering thread
  - skip `lckMutex.P()` after `cvGate.P()`
  - skip `lckMutex.V()` after `cvGate.V()`
  - $\cdots$

Outline

Outline

Increasing Parallelism

- Consider an await-structured program
- Define two awaits to be strongly nonconflicting if they do not conflict even without atomicity of awaits
- Technique 1 to increasing parallelism
  - partition awaits into non-conflicting groups of awaits
  - use separate locks for the groups
  - to avoid deadlock, obtain locks in increasing order
- Technique 2 to increasing parallelism
  - modify code to increase extent of strongly-nonconflicting awaits
  - duplicate hot spots into separate memory areas
  - loosen coupling between duplicates

BBuffPar: BBuff with concurrent put and get

- Implement buff as a circular array
  - buffA[N]
  - in ← 0 // next put call accesses buffA[in]
  - out ← 0 // next get call accesses buffA[out]
  - cnt ← 0 // # items in buffA

- input void mysid.put(x)

```
p1: •await (cnt < N);
p2: •buffA[in] ← x;
     await (true)
         cnt ← cnt+1;
     in ← mod(in+1, N);
     return
```

- input Val mysid.get(x)

```
g1: •await (cnt > 0);
g2: •x ← buffA[out];
     await (true)
         cnt ← cnt − 1;
     out ← mod(out+1, N);
     return
```

- If statements p2 and g2 do not conflict, we can remove their •'s

Proving that p2 and g2 do not conflict

- Let X be BBuffPar with statements p2 and g2 replaced by skip
- Can remove •'s at p2 and g2 if X satisfies $Inv\ D_0$

  D_0 : (thread at p2) and (thread at g2) $\Rightarrow$ in $\neq$ out

- D_1–D_5 satisfies invariance rule and implies D_0, // $Inv\ D_0$ holds

  D_1 : (at most one thread in put)

  D_2 : (at most one thread in get)

  D_3 : (thread on p2) $\Rightarrow$ cnt < N

  D_4 : (thread at g2) $\Rightarrow$ cnt > 0

  D_5 : cnt = mod(in − out, N)

- So •'s at p2 and g2 can be removed from BBuffPar
- Now easy to show BBuffPar implements BoundedBuffer

Outline