Program Structuring

William Cook and Jayadev Misra

Department of Computer Science
University of Texas at Austin

Email: {cook,misra}@cs.utexas.edu
web: http://www.cs.utexas.edu/users/psp

Program Structuring: Running an Auction

- Advertize the item and a minimum bid price v: call $Adv(v)$
- Get bids: $Bids(v)$ returns a stream of increasing bids, all above v.
- Post successive bids at a web site by calling $PostNext$

$$
\begin{array}{l}
Auction_1(v) \;\underline{\Delta} \\
\quad Adv(v) \gg Bids(v) > u > PostNext(u) \gg 0
\end{array}
$$

Program $Bids$

Get the next bid exceeding v.
Assume that bidders put their bids on channel c.

$$
\begin{array}{l}
nextBid(v) \;\underline{\Delta} \\
\qquad c.get \\
\quad >x> \\
\qquad (\quad if\,(x>v) \;\gg\; let(x) \\
\qquad \;\mid\; if\,(x\leq v) \;\gg\; nextBid(v) \\
\qquad)
\end{array}
$$

Output successively increasing bids, all above v.

$$Bids(v) \;\underline{\Delta}\; nextBid(v) \;>u>\; (let(u) \mid Bids(u))$$

A Terminating Auction

- Terminate if no higher bid arrives for an hour (h time units).
- Post the winning bid by calling $PostFinal$.
- Return the value of the winning bid.

$Tbids(v)$ returns pairs (x, b): $b \Rightarrow x > v$, $\neg b \Rightarrow x = v$

$$
\begin{array}{lll}
Auction_2(v) \;\underline{\Delta} & & \\
 & & Adv(v) \\
\gg & & Tbids(v) \\
>(x,b)> & & \\
 & & (\quad if\,(b) \quad \gg \; PostNext(x) \;\gg\; 0 \\
 & & \;\mid\;\; if\,(\neg b) \quad \gg \; PostFinal(x) \;\gg\; let(x) \\
 & &)
\end{array}
$$

$Tbids$

$Tbids(v)$ returns a stream of pairs (x, b):
x is a bid, $x \geq v$, and b is boolean.

$b \quad \Rightarrow x$ exceeds the previous bid
$\neg b \quad \Rightarrow x$ equals the previous bid,
i.e., no higher bid has been received in an hour.

$$
\begin{array}{l}
Tbids(v) \;\underline{\Delta} \\
\quad let(x, b) \mid if(b) \gg Tbids(x) \\
\quad\quad \text{where} \\
\quad\quad\quad (x, b) :\in \; nextBid(v) \; >u> \; let(u, \textit{true}) \\
\quad\quad\quad\quad\quad\quad \mid \; Rtimer(h) \; \gg \; let(v, \textit{false})
\end{array}
$$

Batch Processing the Bids

- Post higher bids only once each hour.
- As before, terminate if no higher bid arrives for an hour.
- As before, post the winning bid by calling $PostFinal$.
- As before, return the value of the winning bid.

$$
\begin{array}{lll}
Auction_3(v) \ \underline{\Delta} & & \\
 & Adv(v) & \\
\gg & Hbids(v) & \\
>(x,b)> & & \\
 & (\quad if\,(b) & \gg PostNext(x) \gg 0 \\
 & \mid \quad if\,(\neg b) & \gg PostFinal(x) \gg let(x) \\
 &) &
\end{array}
$$

$Hbids$

$Hbids(v)$ returns a stream of pairs (x, b), one per hour:
x is a bid, $x \geq v$, and b is boolean.

$b \quad \Rightarrow x$ is the best bid in the last hour and exceeds the last bid
$\neg b \quad \Rightarrow x$ equals the previous bid,
i.e., no higher bid has been received in an hour.

$$
\begin{array}{ll}
Hbids(v) \;\underline{\Delta} & \\
 & clock \\
>t> & bestBid(t+h, v) \\
>x> & \\
 & (\quad let(x, x \neq v) \\
 & \;\mid\quad if(x \neq v) \gg Hbids(x) \\
 &)
\end{array}
$$

$bestBid$

- $bestBid(t, v)$ where t is an absolute time and v is a bid,
- Returns x, $x \geq v$, where x is the best bid received up to t.
- If $x = v$ then no better bid than v has been received up to t.

$$
\begin{array}{l}
bestBid(t, v) \ \underline{\Delta} \\
\quad (if(b) \gg bestBid(t, y) \mid if(\neg b) \gg let(v) \\
\qquad \text{where} \\
\qquad\quad (y, b) :\in \quad nextBid(v) \ >x> \ let(x, \textit{true}) \\
\qquad\qquad\qquad \mid \ Atimer(t) \ >x> \ let(x, \textit{false}) \\
\quad)
\end{array}
$$

Custom Site

Sites may be specific to an application.

- Call sites $M_1, \cdots, M_k$ and respond after a majority of them do.
- Use site Maj to maintain counter c; initially $c = 0$.
- Calling Maj increments c, and returns a signal iff $2 \times c > k$.

$let(u)$

where

$u :\in M_1 \gg Maj \mid \cdots \mid M_k \gg Maj$

Custom Site

Expressions f and g publish increasing sequences of integers.

Publish their merge, casting out duplicates.

Employ site c with special put method.

$$
\begin{array}{l}
(\quad f \; >x> \; c.put(\langle x, \textit{true} \rangle)) \\
\;\; \mid g \; >x> \; c.put(\langle x, \textit{false} \rangle)) \\
) \\
\gg \quad c.get
\end{array}
$$

Available sites

- register holds a data value. Two non-blocking methods:
  - $read$ returns the value.
  - $write(x)$ writes x into the register.
- lock is a monitor. Has a state which is full or empty.
  We have to specify its initial state.

  Two blocking methods:
  - put: if empty, becomes full and returns a signal; else blocks.
  - get: if full, becomes empty and returns a signal; else blocks.

We show how to build more complex sites.

Execution of monitor methods

- A monitor method is executed when it is called.
- Returns just one value. (write just f for $let(z)$ where $z{:\in}\ f$)
- Several methods may be executed simultaneously;
  there may be contention for data.

Typically, only one monitor method is executed at a time.

When one blocks, another is started. Consider

A :: $P \gg Q$ — Q may block.
B :: R

- Executions are serializable. (programmer's obligation)

lock can be used as a binary semaphore

Replace $P \cdots V$ by

$u.get \cdots u.put$,

where lock u is initially full.

Implementing Binary Semaphore, General strategy

Semaphore s is implemented by two complementary locks, u and v.

$$s = 1 \equiv u.full \wedge \; v.empty$$
$$s = 0 \equiv u.empty \wedge \; v.full$$

u: $lock(full)$, v: $lock(empty)$
P:: $u.get \gg v.put$
V:: $(u.get \mid v.get) \gg u.put$

Serializability: Method executions can not be interleaved.

Exactly one of $u.get$ and $v.get$ succeeds.

After execution of either $u.get$ or $v.get$ both u and v are empty.

This prevents any other method from starting.

Binary Semaphore can implement lock

Use complementary semaphores, s and t, to simulate lock u.

$$\begin{array}{lcl} u.empty & \equiv & s = 1 \wedge t = 0 \\ u.full & \equiv & s = 0 \wedge t = 1 \end{array}$$

site $lock$

s, t: $BinSemaphore$

get:: $t.P \gg s.V$

put:: $s.P \gg t.V$

Method executions can not be interleaved.

After execution of $t.P$ or $s.P$ both s and t are zero.

This prevents any other method from starting.

word

A $word$ is a 1-place buffer. It has two blocking methods.

1. $put(x)$: blocks if the word is full;

   otherwise, it writes x to the word and returns a signal.

2. get: blocks if the word is empty;

   otherwise, returns the value of the word and makes it empty.

Implementation of word

Implement word w using lock u and register c.

Invariants

$u.full \equiv w.full$

$w.full \Rightarrow w = c$

site $word$

u: $lock(empty)$, c: $register$

$put(x) :: u.put \gg c.write(x)$

$get :: u.get \gg c.read$

Implementation is not serializable

site $word$

u: $lock(empty)$, c: $register$

$put(x) :: u.put \gg c.write(x)$
$get :: u.get \gg c.read$

Consider
u is empty;
A attempts $u.put$ and succeeds;
B executes $u.get$ and $c.read$, thus reading the previous value.

Conversely,
u is full;
P attempts $u.get$ and succeeds;
Q executes $u.put$ and $c.write$, thus overwriting the previous value.

Correct Implementation

Use complementary locks u and v.

$$u.full \equiv w.full\, ;\;\; v.full \equiv w.empty$$
$$w.full \Rightarrow w = c$$

site $word$

$u\colon lock(empty)$, $v\colon lock(full)$, $c\colon register$

$$put(x) :: v.get \gg c.write(x) \gg u.put$$
$$get :: u.get \gg c.read > x > v.put \gg let(x)$$

- Either put or get succeeds. (semaphore instead of v?)
- Once a method starts executing $u.empty \wedge v.empty$.
  No other method can then start.

word’

$word'$ is same as $word$ with two more non-blocking methods.

1. $put(x)$: blocks if the word is full;

   otherwise, it writes x to the word and returns a signal.

2. get: blocks if the word is empty;

   otherwise, returns the value of the word and makes it empty.

3. $put'(x)$: returns *true* if it succeeds, *false* otherwise.

4. get': returns $(w, \textit{true})$ if w is full, $(-, \textit{false})$ otherwise.

Implementing word', complementary locks u and v

$u.full \equiv w.full$; $v.full \equiv w.empty$
$w.full \Rightarrow w = c$

site $word'$
u: $lock(empty)$, v: $lock(full)$, c: $register$

$put'(x)$ :: $let(z) \gg u.put \gg let(z)$ $\{ u.put$ does not block$\}$
where $z :\in v.get \gg c.write(x) \gg let($*true*$)$
$\mid u.get \gg let($*false*$)$

get' :: $let(z) \gg v.put \gg let(z)$ $\{ v.put$ does not block$\}$
where $z :\in u.get > x > let(x,$ *true*$)$
$\mid v.get \gg let(-,$ *false*$)$

Prove serializability with all four methods.

Unbounded Channel

- $put(x)$: non-blocking. Adds x to the end of the channel.
- get: blocks if channel is empty, else returns the head of the channel.
- get': returns $(w, \textit{true})$ if channel non-empty, else returns $(-, \textit{false})$.

There is no put' because the channel is unbounded.

Implementation of Unbounded Channel

Site UnboundedChannel

p: UnboundedChannel; w: word' $\{$ w is the first word$\}$

$$
\begin{aligned}
&put(x):: && w.get' \\
& && >y,b> \ (if(b) \gg p.put(y) \mid if(\neg b)) \\
& && \gg w.put(x)
\end{aligned}
$$

$$
\begin{aligned}
&get:: && w.get \ >x> \ p.get' \\
& && >y,b> \ (if(b) \gg w.put(y) \mid if(\neg b)) \gg let(x)
\end{aligned}
$$

$$
\begin{aligned}
&get':: && w.get' \\
& && >x,b> \ (if(b) \gg p.get' \ >y,c> \ (if(c) \gg w.put(y) \mid if(\neg c)) \\
& && \quad \mid \ if(\neg b)) \\
& && \gg let(x,b)
\end{aligned}
$$

complementary locks, $u.full \equiv w.full$; $v.full \equiv w.empty$

Site UnboundedChannel

p: UnboundedChannel; w: word'; u: $lock(empty)$, v: $lock(full)$

$$
\begin{aligned}
put(x) ::\quad & (u.get \mid v.get) \gg w.get' \\
& >y,b>\ (if(b) \gg p.put(y) \mid if(\neg b)) \\
& \gg w.put(x) \gg u.put
\end{aligned}
$$

$$
\begin{aligned}
get ::\quad & u.get \gg w.get \ >x>\ p.get' \\
& >y,b>\ (if(b) \gg w.put(y) \gg u.put \mid if(\neg b) \gg v.put) \gg let(x)
\end{aligned}
$$

$$
\begin{aligned}
get' ::\quad & (u.get \mid v.get) \gg w.get' \\
& >x,b>\ (if(b) \gg p.get' \\
& >y,c>\ \ (if(c) \gg w.put(y) \gg u.put \mid if(\neg c) \gg v.put) \\
& \qquad \mid\ if(\neg b) \gg v.put)) \\
& \gg let(x,b)
\end{aligned}
$$

Bounded Channel

Site Boundedchannel(1)
w: word

Site Boundedchannel (n)
b: Boundedchannel $(n-1)$, w: word, u: $lock(empty)$, v: $lock(full)$
$\vdots$

Rendezvous

- sender executes put, receiver executes get.
- Both methods complete together.
- Other senders, receivers are blocked until then.

For the moment, assume no data is transferred. Only signals returned.

Implementation

site $SignalRendezvous$

u: $lock(empty)$, v: $lock(empty)$

$put \quad :: u.get \parallel v.put$

$get \quad :: v.get \parallel u.put$

- sender does $v.put$.
- receiver completes its operation (both $v.get$ and $u.put$)
- second sender completes its operation.

Two senders or two receivers should not be simultaneously active.

Mutual exclusion: among senders and among receivers

Use lock r for receivers and s for senders.

site $SignalRendezvous$

$u\colon lock(empty)$, $v\colon lock(empty)$,

$r\colon lock(full)$, $s\colon lock(full)$

$$put \quad :: s.get \gg (u.get \parallel v.put) \gg s.put$$

$$get \quad :: r.get \gg (v.get \parallel u.put) \gg r.put$$

Rendezvous with Data Transfer

Identical program, except v is a word.

site $Rendezvous$

u: $lock(empty)$, v: $word$,

r: $lock(full)$, s: $lock(full)$

$$put(x) :: s.get \gg (u.get \parallel v.put(x)) \gg s.put$$

$$get :: r.get \gg f > y > r.put \gg let(y)$$

$$f \triangleq let(x, y) \gg let(y)$$

where $x :\in u.put$

$y :\in v.get$

Exercise

In $(f \text{ where } x :\in g)$, executions of f and g start simultaneously.

Modify the expression so that g is evaluated when needed.

In $(M \gg N(x) \text{ where } x :\in g)$, g may not be evaluated at all.

Hint: Use a boolean register site.