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 \( \text{Adv}(v) \)
- Get bids: \( \text{Bids}(v) \) returns a stream of increasing bids, all above \( v \).
- Post successive bids at a web site by calling \( \text{PostNext} \)

\[
\text{Auction}_1(v) \quad \Delta \\
\text{Adv}(v) \quad \rightarrow \quad \text{Bids}(v) \quad \rightarrow u \quad \rightarrow \quad \text{PostNext}(u) \quad \rightarrow 0
\]
Program \textit{Bids}

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

\[
\text{nextBid}(v) \triangleq c.\text{get} \\
> x > \\
( \text{if } (x > v) \Rightarrow \text{let}(x) \\
| \text{if } (x \leq v) \Rightarrow \text{nextBid}(v) \\
) 
\]

Output successively increasing bids, all above \( v \).

\[
\textit{Bids}(v) \triangleq \text{nextBid}(v) > u > (\text{let}(u) \mid \textit{Bids}(u))
\]
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.

\[ T\text{bids}(v) \text{ returns pairs } (x, b) : b \Rightarrow x > v, \: \neg b \Rightarrow x = v \]

\[ Auction_2(v) \triangleq \]

\[ \begin{align*}
\text{Adv}(v) \\
\triangleright \\
(x, b) > \\
\text{PostNext}(x) \triangleright 0 \\
\text{PostFinal}(x) \triangleright \text{let}(x)
\end{align*} \]
$Tbids(v)$ returns a stream of pairs $(x, b)$:

- $x$ is a bid, $x \geq v$, and $b$ is boolean.

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

\[
Tbids(v) \triangleq \text{let}(x, b) \mid \text{if}(b) \Rightarrow Tbids(x)
\]

where

\[
(x, b) \in \text{nextBid}(v) > u > \text{let}(u, \text{true}) \mid \text{Rtimer}(h) \Rightarrow \text{let}(v, \text{false})
\]
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 $\text{PostFinal}$.
- As before, return the value of the winning bid.

\[
\text{Auction}_3(v) \triangleq \\
\begin{align*}
&\text{Adv}(v) \\
\Rightarrow & \text{Hbids}(v) \\
\geq (x, b) > & \\
( & \text{if } (b) \Rightarrow \text{PostNext}(x) \Rightarrow 0 \\
& \text{if } (-b) \Rightarrow \text{PostFinal}(x) \Rightarrow \text{let}(x) \\
) 
\end{align*}
\]
\textit{Hbids} returns a stream of pairs \((x, b)\), one per hour:

- \(x\) is a bid, \(x \geq v\), and \(b\) is boolean.

\(b \Rightarrow x\) is the best bid in the last hour and exceeds the last bid
\(\neg b \Rightarrow x\) equals the previous bid,

i.e., no higher bid has been received in an hour.

\[
\text{Hbids}(v) \triangleq \begin{align*}
&\text{clock} \\
&> t > \quad \text{bestBid}(t + h, v) \\
&> x > \\
&\quad \left( \begin{array}{c}
\text{let}(x, x \neq v) \\
\quad \text{if}(x \neq v) \Rightarrow \text{Hbids}(x)
\end{array} \right)
\end{align*}
\]
• $bestBid(t, v)$ 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 $t$.

\[
bestBid(t, v) \triangleq \begin{cases} 
    \text{if}(b) \implies bestBid(t, y) & \text{if}(\neg b) \implies \text{let}(v) \\
\end{cases}
\]

where

\[
(y, b) \in nextBid(v) > x > \begin{cases} 
    \text{let}(x, \text{true}) & \text{Atimer}(t) > x > \text{let}(x, \text{false}) 
\end{cases}
\]
Custom Site

Sites may be specific to an application.

- Call sites $M_1, \ldots, M_k$ 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 $2 \times c > k$.

\[
\text{let}(u) \\
\text{where} \\
\ u : \in M_1 \Rightarrow Maj \ | \ \cdots \ | \ M_k \Rightarrow Maj
\]
Expressions $f$ and $g$ publish increasing sequences of integers.

Publish their merge, casting out duplicates.

Employ site $c$ with special put method.

\[
\begin{align*}
( & f \ x > c.\text{put}(\langle x, \text{true}\rangle) \\
| & g \ x > c.\text{put}(\langle x, \text{false}\rangle) \\
) \\
\Rightarrow & \ c.\text{get}
\end{align*}
\]
Available sites

- **register** holds a data value. Two non-blocking methods:
  - \textit{read} returns the value.
  - \textit{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:
  - \textit{put}: if empty, becomes full and returns a signal; else blocks.
  - \textit{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 $\text{let}(z) \text{ where } z \in f$)
- 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

$$\begin{align*}
A &::\quad P \Rightarrow Q \quad \text{— } Q \text{ may block.} \\
B &::\quad R
\end{align*}$$

- 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 \( v \).

\[
\begin{align*}
  s = 1 & \equiv u.\text{full} \land v.\text{empty} \\
  s = 0 & \equiv u.\text{empty} \land v.\text{full}
\end{align*}
\]

\( u: \text{lock(full)} \), \( v: \text{lock(EMPTY)} \)

\[
\begin{align*}
P:: & u.\text{get} \rightarrow v.\text{put} \\
V:: & (u.\text{get} | v.\text{get}) \rightarrow u.\text{put}
\end{align*}
\]

**Serializability**: Method executions can not be interleaved.

Exactly one of \( u.\text{get} \) and \( v.\text{get} \) succeeds.

After execution of either \( u.\text{get} \) or \( v.\text{get} \) 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 \( u \).

\[
\begin{align*}
\text{u.empty} & \equiv s = 1 \land t = 0 \\
\text{u.full} & \equiv s = 0 \land t = 1
\end{align*}
\]

site \lock
\[
\begin{align*}
s, t: & \text{BinSemaphore} \\
\text{get::} & t.P \Rightarrow s.V \\
\text{put::} & s.P \Rightarrow t.V
\end{align*}
\]

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.
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

\[ u.\text{full} \equiv w.\text{full} \]
\[ w.\text{full} \Rightarrow w = c \]

site \textit{word}

$u$: lock(\text{empty}), \ c$: register

\[ \text{put}(x): u.\text{put} \Rightarrow c.\text{write}(x) \]
\[ \text{get} \quad \Rightarrow u.\text{get} \Rightarrow c.\text{read} \]
Implementation is not serializable

site \textit{word}
\begin{itemize}
  \item \textit{u: lock} (empty), \textit{c: register}
\end{itemize}

\begin{align*}
\text{put}(x) &:: u.\text{put} \Rightarrow c.\text{write}(x) \\
\text{get} &:: u.\text{get} \Rightarrow c.\text{read}
\end{align*}

Consider
\begin{itemize}
  \item \textit{u} is empty;
  \item \textit{A} attempts \textit{u.put} and succeeds;
  \item \textit{B} executes \textit{u.get} and \textit{c.read}, thus reading the previous value.
\end{itemize}

Conversely,
\begin{itemize}
  \item \textit{u} is full;
  \item \textit{P} attempts \textit{u.get} and succeeds;
  \item \textit{Q} executes \textit{u.put} and \textit{c.write}, thus overwriting the previous value.
\end{itemize}
Correct Implementation

Use complementary locks \( u \) and \( v \).

\[
\begin{align*}
    u.\text{full} &\equiv w.\text{full} ; \quad v.\text{full} \equiv w.\text{empty} \\
    w.\text{full} &\Rightarrow w = c
\end{align*}
\]

site \textit{word}

\[
\begin{align*}
    u: & \text{lock(}\text{empty}\text{)} , \quad v: \text{lock(}\text{full}\text{)} , \quad c: \text{register} \\
    \text{put}(x) &:: v.\text{get} \Rightarrow c.\text{write}(x) \Rightarrow u.\text{put} \\
    \text{get} &:: u.\text{get} \Rightarrow c.\text{read} > x > v.\text{put} \Rightarrow \text{let}(x)
\end{align*}
\]

- Either \textit{put} or \textit{get} succeeds. (semaphore instead of \( v \)?)

- Once a method starts executing \( u.\text{empty} \land v.\text{empty} \).
  No other method can then start.
word' is same as word with two more non-blocking methods.

1. \( \text{put}(x) \): blocks if the word is full;
   otherwise, it writes \( x \) to the word and returns a signal.

2. \( \text{get} \): blocks if the word is empty;
   otherwise, returns the value of the word and makes it empty.

3. \( \text{put'}(x) \): returns \( \text{true} \) if it succeeds, \( \text{false} \) otherwise.

4. \( \text{get'} \): returns \( (w, \text{true}) \) if \( w \) is full, \( (−, \text{false}) \) otherwise.
Implementing word’, complementary locks $u$ and $v$

$u.full \equiv w.full; \ v.full \equiv w.empty$

$w.full \Rightarrow w = c$

site $word'$

$u: lock(\emptyset), \ v: lock(\text{full}), \ c: \text{register}$

$put'(x) \ :: \ let(z) \ \triangleright u.put \ \triangleright let(z) \ \{ \ u.put \ \text{does not block}\}$

$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{where } z: \in \ v.get \ \triangleright c.write(x) \ \triangleright let(\text{true})$

$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ | \ u.get \ \triangleright let(\text{false})$

$get' \ :: \ let(z) \ \triangleright v.put \ \triangleright let(z) \ \{ \ v.put \ \text{does not block}\}$

$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{where } z: \in \ u.get > x > let(x, \text{true})$

$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ | \ v.get \ \triangleright let(\neg, \text{false})$

Prove serializability with all four methods.
Unbounded Channel

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

There is no \textit{put}' because the channel is unbounded.
Implementation of Unbounded Channel

Site UnboundedChannel

\[ p: \text{UnboundedChannel}; \quad w: \text{word}' \quad \{ \ w \text{ is the first word} \} \]

\[
\text{put}(x):: \quad w.\text{get}' \\
> y, b > \quad (\text{if}(b) \quad \Rightarrow p.\text{put}(y) \quad \mid \quad \text{if}(\neg b)) \\
\Rightarrow w.\text{put}(x)
\]

\[
\text{get}:: \quad w.\text{get} > x > p.\text{get}' \\
> y, b > \quad (\text{if}(b) \quad \Rightarrow w.\text{put}(y) \quad \mid \quad \text{if}(\neg b)) \quad \Rightarrow \text{let}(x)
\]

\[
\text{get}':: \quad w.\text{get}' \\
> x, b > \quad (\text{if}(b) \quad \Rightarrow p.\text{get}' > y, c > \quad (\text{if}(c) \quad \Rightarrow w.\text{put}(y) \quad \mid \quad \text{if}(\neg c)) \\
\mid \quad \text{if}(\neg b)) \\
\Rightarrow \text{let}(x, b)
\]
complementary locks, \( u.\text{full} \equiv w.\text{full} \); \( v.\text{full} \equiv w.\text{empty} \)

Site UnboundedChannel

\( p: \text{UnboundedChannel}; \; w: \text{word'}; \; u: \text{lock}(\text{empty}), \; v: \text{lock}(\text{full}) \)

\[
\begin{align*}
\text{put}(x): & \quad (u.\text{get} \mid v.\text{get}) \to w.\text{get}' \\
& \quad > y, b > \quad (\text{if}(b) \to p.\text{put}(y) \mid \text{if}(\neg b)) \\
& \quad \to w.\text{put}(x) \to u.\text{put} \\
\text{get}: & \quad u.\text{get} \to w.\text{get} > x > p.\text{get}' \\
& \quad > y, b > \quad (\text{if}(b) \to w.\text{put}(y) \to u.\text{put} \mid \text{if}(\neg b) \to v.\text{put}) \to \text{let}(x) \\
\text{get'}: & \quad (u.\text{get} \mid v.\text{get}) \to w.\text{get}' \\
& \quad > x, b > \quad (\text{if}(b) \to p.\text{get}') \\
& \quad > y, c > \quad (\text{if}(c) \to w.\text{put}(y) \to u.\text{put} \mid \text{if}(\neg c) \to v.\text{put}) \\
& \quad \mid \text{if}(\neg b) \to v.\text{put}) \to \text{let}(x, b)
\end{align*}
\]
Bounded Channel

Site Boundedchannel(1)
\[ w: \text{word} \]

Site Boundedchannel \( (n) \)
\[ b: \text{Boundedchannel } (n - 1), \ w: \text{word}, \ u: \text{lock(}\text{empty}\text{)}, \ v: \text{lock(}\text{full}\text{)} \]

\[ : \]
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 $\textit{SignalRendezvous}$

\[
\begin{align*}
    u: \text{lock}(\text{empty}), & \quad v: \text{lock}(\text{empty}) \\
    \text{put} & \quad :: u.\text{get} || v.\text{put} \\
    \text{get} & \quad :: v.\text{get} || u.\text{put}
\end{align*}
\]

- sender does $v.\text{put}$.

- receiver completes its operation (both $v.\text{get}$ and $u.\text{put}$)

- 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 $\text{SignalRendezvous}$  
$u: \text{lock}(\text{empty})$,  $v: \text{lock}(\text{empty})$,  
$r: \text{lock}(\text{full})$,  $s: \text{lock}(\text{full})$

$\begin{align*}
\text{put} &:: s.\text{get} \parallel (u.\text{get} \parallel v.\text{put}) \parallel s.\text{put} \\
\text{get} &:: r.\text{get} \parallel (v.\text{get} \parallel u.\text{put}) \parallel r.\text{put}
\end{align*}$
Rendezvous with Data Transfer

Identical program, except $v$ is a word.

site \textit{Rendezvous}  
\begin{align*}  
u &::= \text{lock}(\text{empty}), \quad v::= \text{word}, \\
r &::= \text{lock}(\text{full}), \quad s::= \text{lock}(\text{full})  \\
\end{align*}

\begin{align*}  
\text{put}(x) &:: s.\text{get} \gg (u.\text{get} \parallel v.\text{put}(x)) \gg s.\text{put} \\
\text{get} &:: r.\text{get} \gg f > y \gg r.\text{put} \gg \text{let}(y)  \\
\end{align*}

\begin{align*}  
f \triangleq \text{let}(x, y) \gg \text{let}(y)  \\
\text{where} \quad x &::= u.\text{put} \\
\quad y &::= v.\text{get}  \\
\end{align*}
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 \((M \implies N(x) \text{ where } x:\in g)\), \(g\) may not be evaluated at all.

Hint: Use a boolean register site.