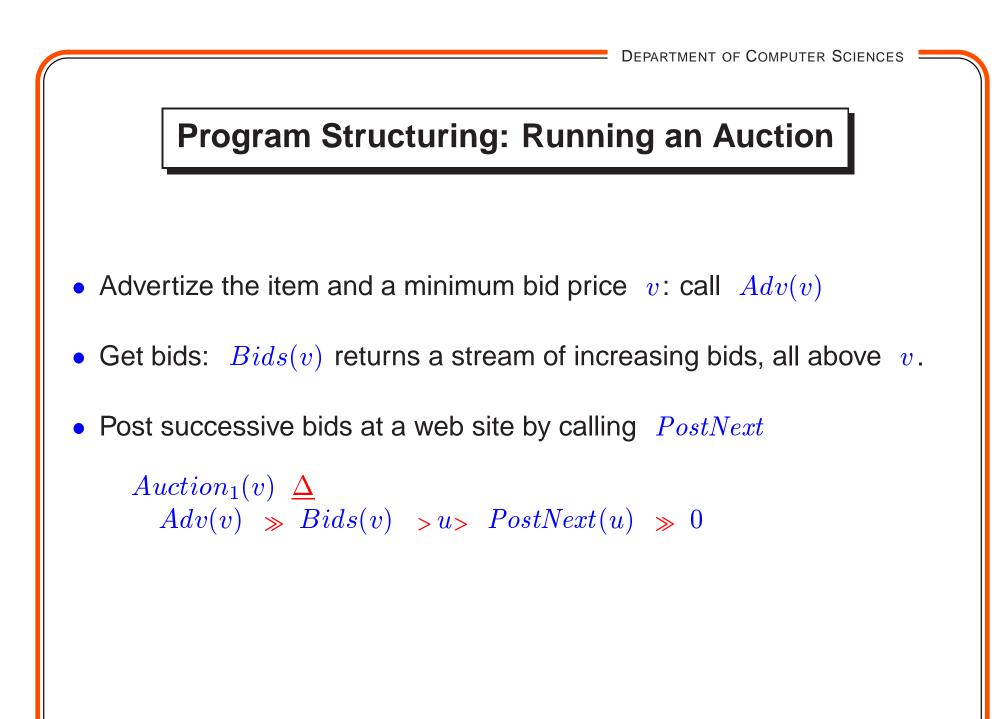
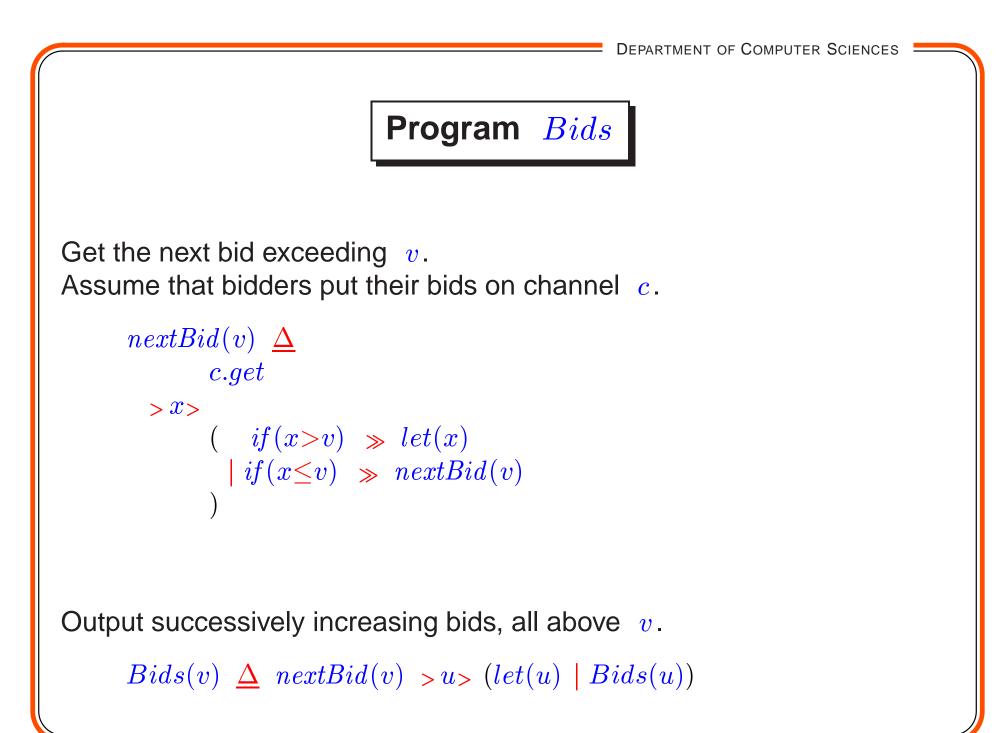
Program Structuring

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

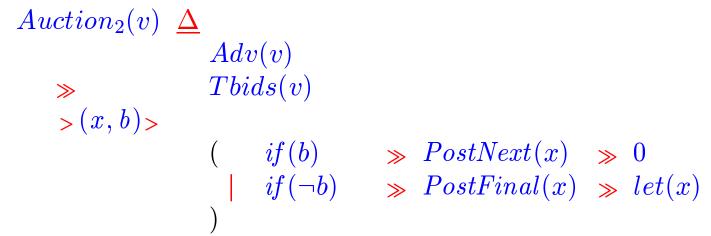




A Terminating Auction

- 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): $b \Rightarrow x > v$, $\neg b \Rightarrow x = v$





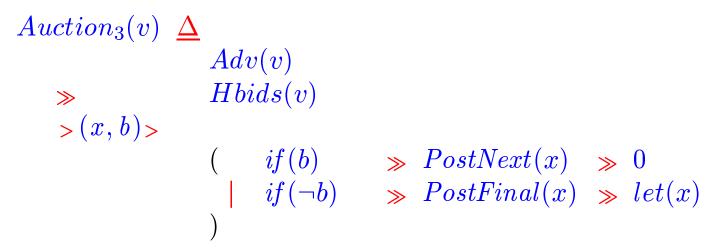
Tbids(v) returns a stream of pairs (x, b): x is a bid, $x \ge v$, and b is boolean.

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

```
\begin{array}{rll} Tbids(v) & \underline{\Delta} \\ let(x,b) & \mid if(b) \gg Tbids(x) \\ & \text{where} \\ & (x,b) :\in \ nextBid(v) \ >u > \ let(u, \textit{true}) \\ & \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.





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

Hbids(v) Δ

> <i>t</i> >	bestBid(t+h,v)
> x >	$(\begin{array}{cc} let(x, x \neq v) \\ \mid & if(x \neq v) \gg Hbids(x) \\) \end{array}$

clock

bestBid

- bestBid(t, v) where t is an absolute time and v is a bid,
- Returns x, $x \ge v$, 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.

Custom Site

Sites may be specific to an application.

- Call sites M_1, \dots, 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.

```
( f > x > c.put(\langle x, true \rangle) \\ | g > x > c.put(\langle x, false \rangle) \\) \\ \gg c.get
```

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)

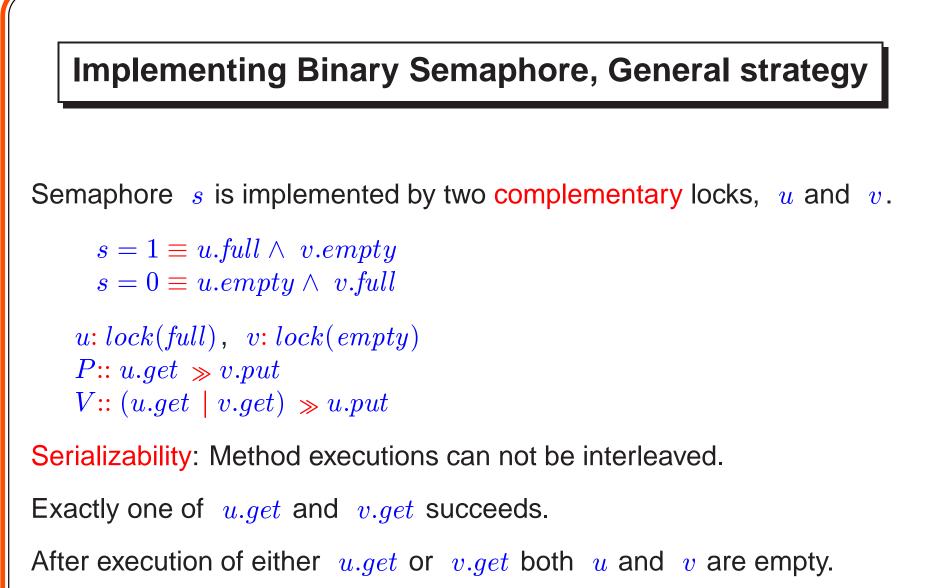


lock can be used as a binary semaphore

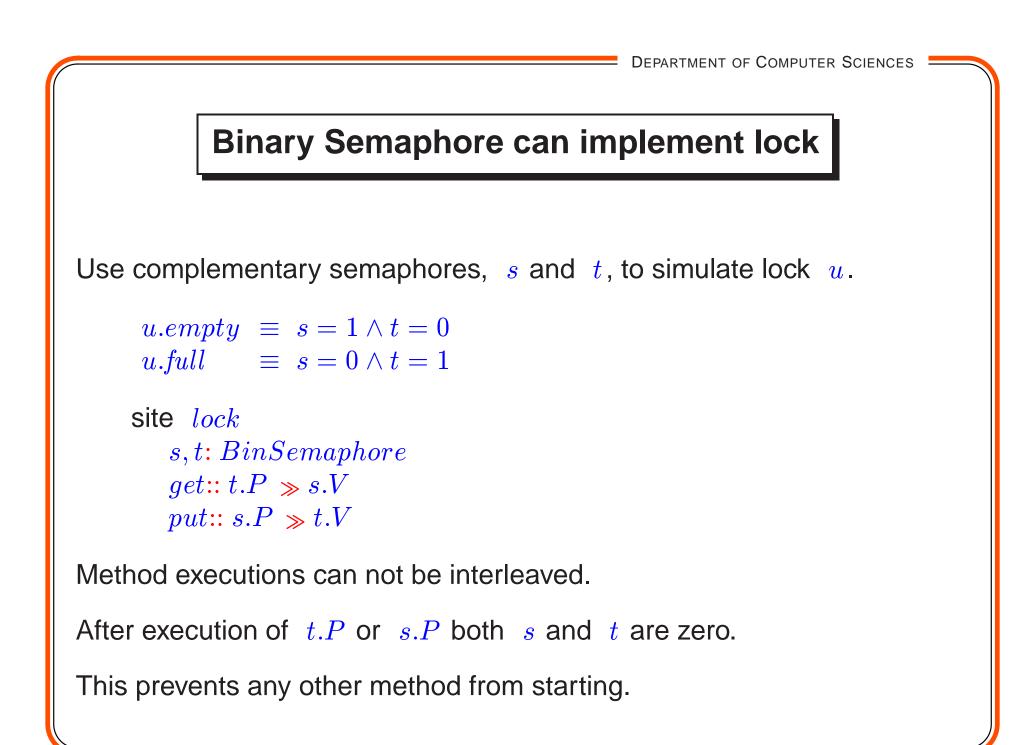
Replace $P \cdots V$ by

 $u.get \cdots u.put$,

where lock u is initially full.



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
```

```
\begin{array}{ll} put(x) :: u.put \ \gg c.write(x) \\ get & :: u.get \ \gg c.read \end{array}
```

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: lock(empty), v: lock(full), c: register
```

```
put(x) :: v.get \gg c.write(x) \gg u.putget :: 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 $u.empty \land v.empty$. 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, true) if w is full, (-, false) otherwise.

Implementing word', complementary locks u and v

```
\begin{array}{l} u.full \equiv w.full; \ v.full \equiv w.empty \\ w.full \Rightarrow w = c \end{array}

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) \\ & \ \| u.get \gg let(false) \\ \end{array}

get' \ :: let(z) \gg v.put \gg let(z) \ \{ v.put \ does \ not \ block \} \\ \end{array}
```

where $z :\in u.get > x > let(x, true)$ $v.get \gg let(-, false)$

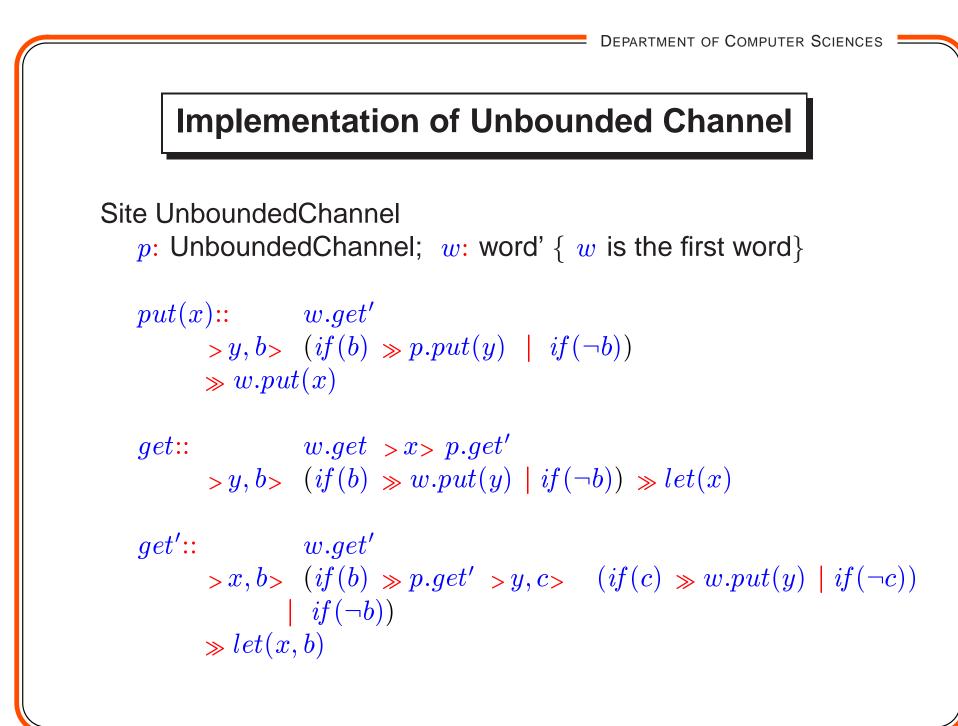
Prove serializability with all four methods.



Unbounded Channel

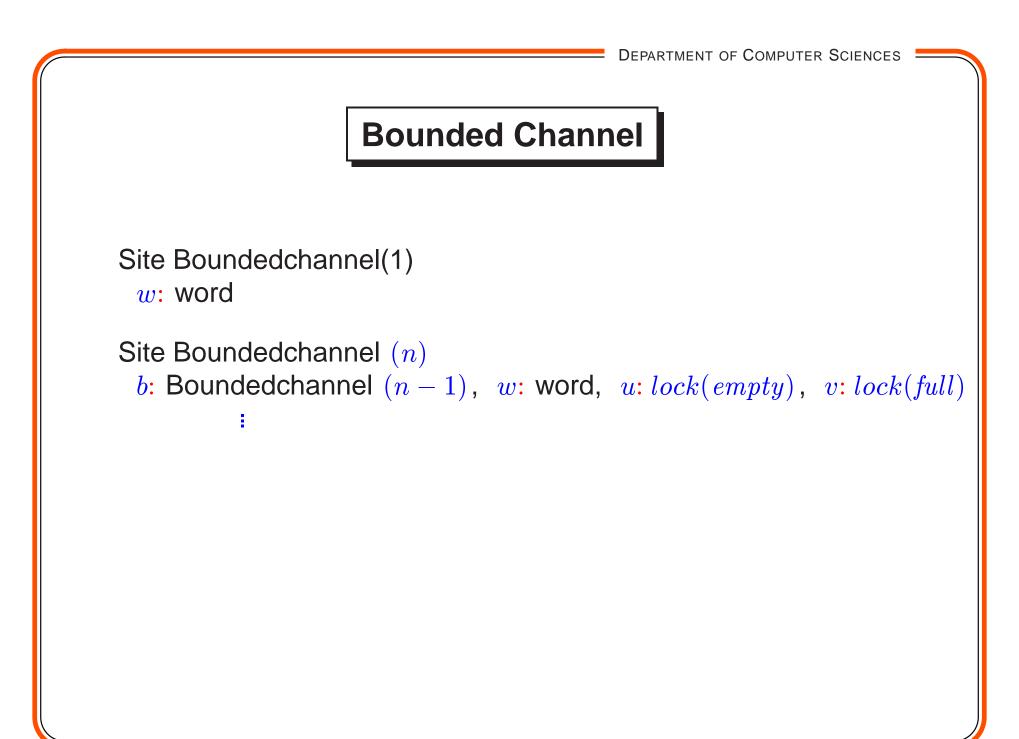
- put(x): non-blocking. Adds x to the end of the channel.
- *get*: blocks if channel is empty, else returns the head of the channel.
- get': returns (w, true) if channel non-empty, else returns (-, false).

There is no put' because the channel is unbounded.



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

Site UnboundedChannel *p*: UnboundedChannel; *w*: word'; *u*: lock(empty), *v*: lock(full)put(x):: $(u.get \mid v.get) \gg w.get'$ >y, b> (if (b) $\gg p.put(y)$ | if $(\neg b)$) $\gg w.put(x) \gg u.put$ get:: $u.get \gg w.get > x > p.get'$ >y, b> (if (b) $\gg w.put(y) \gg u.put | if(\neg b) \gg v.put) \gg let(x)$ get':: $(u.get \mid v.get) \gg w.get'$ >x,b> (if (b) $\gg p.get'$ >y,c> (if (c) $\gg w.put(y) \gg u.put \mid if(\neg c) \gg v.put$) $| if(\neg b) \gg v.put))$ $\gg let(x,b)$



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 :: u.get || v.put
    get :: v.get || 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.





Use lock r for receivers and s for senders.

```
site SignalRendezvous
    u: lock(empty), v: lock(empty),
    r: lock(full), s: lock(full)
```



Rendezvous with Data Transfer

```
Identical program, except v is a word.
```

```
site Rendezvous
    u: lock(empty), v: word,
    r: lock(full), s: lock(full)
```

```
\begin{array}{rl} f & \underline{\Delta} & let(x,y) \ \gg let(y) \\ & \text{where} & x :\in u.put \\ & y :\in v.get \end{array}
```

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.