Announcements 10/1

- Tutorial on this week's lectures (10/1, 10/3)
  Sunday 10/6  3pm-4:30pm
  Hollister B14

- Looking ahead: Prelim 1 is Thurs 10/8
  +1 week likely review session 10/8 in class
Beyond Semaphores (cont.)

\[ M : \text{Monitor} \]
\[ \text{var} \quad \text{... monitor vars...} \]
\[ \text{C : Condition} \quad \in B_c \subseteq \]

\[ \text{op : operation}(p_1, p_2, ...) \]
\[ \text{var} \quad \text{loads} \]
\[ \text{end} \]
\[ \vdots \]

\text{C.wait} \quad \text{thread suspended}

\text{C.continue:} \quad \text{thread exits Monitor & thread blocked on C runs}

\text{VS}

\text{C.signal:} \quad \text{thread suspends on "urgent" queue & thread blocked on C ring}
var ant: integer

inc. cond: \text{condition} \quad \text{if} \quad \text{w} \geq \text{ant}?

\text{deposit}: \quad \text{operate} (N: \text{integer})

\text{ant} := \text{ant} + N

\text{inc. signal}

\text{end}

\text{withdraw}: \quad \text{operate} (W: \text{integer})

\text{while} \quad \text{ant} < \text{w} \quad \text{do} \quad \text{inc. wait and end}

\text{ant} := \text{ant} - \text{w}

\text{end}

\text{end Aent}
Suppose signal operation does not cause yield.

If (B) runs next...

Note: Actually, I need not hold task after notify because no process gets control at that point!
Implementing of signal/agent semantics

entry queue for monitor
condition queue for each condition variable
urgent queue for monitor

monitor call: If monitor in use
then add thread to entry queue
else grant access

e. wait: Put thread on queue for condition c
Invoke scheduler

e. notify: 1 process on condition queue for c
made runnable output
Continue executing in monitor

c. notifyAll: all processes on condition queue for c
made runnable
Continue executing in monitor

Scheduler: Pick some runnable thread
& run it - at most one process executes in monitor
Summary of signal regimes

C. Continue: thread exits monitor

C. Signal: thread suspends on "urgent" queue

C. Notify: thread continues to exec in monitor

All cause thread suspended on C. wait to obtain monitor lock eventually

C. Continue \rightarrow immediately

C. Signal \rightarrow eventually

C. Notify \rightarrow eventually

Compare with P & V!
Use of notify is tricky.
Two processes can Acquire when if instead of while.

monitor:
  var locked: boolean init false
  Q: condition

Acquire: operation
  if locked then Q.wait
  while locked do Q.wait end
  locked := true
end

Release: operation
  locked := false
  Q.notify
end

end monitor
Cautions when using monitors

I: Nested locking

1. Call m.op
2. Call m.op
3. C.wait

I would have to hold prior to call if we wanted to relax mutex for m (Bad idea!)
Priority Inversion

Thread

A: high pro
B: med pro
C: low pro

Time

C: locks mutex M
C: yields processor to B
B: runs long computation

B: yields processor to A
A: attempts to lock mutex M
   (blocked due to C)
   i: A yields processor to B
B: continues long computation
Deconstructing Monitors: (Critical) Regions

Yet another synchronization pinata.

(Good ref: Birrell, "An Introduction to Programming with Threads")

Monitor: 

\[ \begin{align*} 
\text{mutual exclusion (for ops)} \\
\text{(3 ideas)} \quad \text{and synchron (within ops)} \\
\text{selective mutex (due to visibility)} \\
\text{rules for vars} \end{align*} \]

- locks define regions
- syntax of a region:

```plaintext
var m: lock
region m do
    ...
end
```
Condition variables allow release of locks (wait, notify, notifyAll)

```java
var m: lock

space: condition with m
stuff: condition with m

//

region m do
while slots = 0 do space.wait end
buff [shl + len] := 0
len := len + 1: slots := slots - 1
stuff.notify
end

//

region m do
while len = 0 do space.wait end
val := buff[shl]
len := len - 1; slots := slots + 1
shl := shl + 1 mod n + 1
space.notify
end
```
- Associate sets of variables with each lock.
  - Allows arbitrary fine-grain grouping of variables.

- Problems if each var is associated with multiple locks.
Message Passing

\[
\text{send } m \text{ to dest} \\
\text{receive } m \text{ from source}
\]

Various design decisions
How to specify destination & source of msg

↑ for send  ↑ for receive

1. **direct naming:**
   - sender names receiver names sender names receiver names
   - $O(N^2)$ channels
   - **Problem:** Processes need to know each other's names

2. **asymmetric direct naming:**
   - sender names receiver names nobody
Synchrony

- blocking ("synchronous"): causes primitive to delay until some event
- non-blocking ("asynchronous"): primitive continues

Blocking send: sender delayed until msg received

Client

Send x, y, z to C1

Receive val

Server

Receive a, b, c

Send res to
Buffer capacity

How many sent but not received msgs allowed?

0 - capacity
N - capacity

send as V?
receive as P.