- Tutorial on Semaphores (Lects 9/17, 9/19) for students who attended Tapia Conference.
  But any CS4410 student welcome to attend Sunday 9/29 3pm-4:30pm Hollister B14 (?)

- Assignment 2 hints
  - for problem 1:
    - Critical sections always terminate.
    - Non-critical sections (NCS) is not required to terminate and might not!

  - for problem 2:
    - mutual exclusion protocols are expected to work for $N \geq 2$ processes.

- Looking ahead: Prelim 1 is Thurs 10/10 + 2 weeks 😞
  Likely review session 10/8 in class
Synchronization primitives
- shared memory
- semaphores
- monitors vs regions

Tension:
- generality / expressiveness
- usability / clarity

Exploit:
- syntax
- compilers (syntax checking)
- run-time
foo: object

```plaintext
var monitor vars

op1: operation (p1, p2, ...)
    var locals
    \{
    code
    \}

op2: operation (p1, p2, ...)
    var locals
    \{
    code
    \}

... 

initialization
```

Schematic representation of monitor
Support for synchronization =

\text{mutex} + \text{Cond\ synch}

\text{per\ operation}

\text{op.}\ \text{operation}\ (p_1, p_2, \ldots)
\text{var}\ \text{locals}

\text{Require that } B\ \text{holds}\ (*)\ \text{and}\ \text{shuts IN} \ldots

\text{end}
BB:  Monitor

var  buff [0..N] of portion
    stt:  0..N
    len:  0..N+1
    slots:  0..N+1

insert:   operation (v: portion)
  need a slot!
  buff [stt+len] := v
  len := len + 1;  slots := slots - 1

RMV: operation (v: val: portion)
  need a portion!
  val := buff [stt]
  len := len - 1;  slots := slots + 1
  stt := stt + 1 mod N+1

begin
  stt := 0
  len := 0
  slots := 0
end
Condition variables

\[
\texttt{var} \quad C : \text{condition} \quad \rightarrow \quad \text{associate } B_c
\]

- \texttt{a.wait}
  
  \[\text{1. releases mutex block for Monta}\]
  
  \[\text{2. blocks process}\]

- \texttt{c.continue}
  
  \[\text{exit monitor \& unblock 1 process}\]
BB: Monitor

var buff [0..N] of partition

start: 0..N
len: 0..N+1
slots: 0..N+1

space: cond
stuff: cond

\[ \text{if } \text{slots} > 0\text{ then } \text{space. out}\]

\[ \text{buff}[\text{start}+\text{len}]:=V \]
\[ \text{len}:=\text{len}+1; \text{ slots}:=\text{slots}-1 \]
\[ \text{stuff. continue} \]

RMV: operation (var val: partition)

\[ \text{need a partition!} \text{ if } \text{len}=0 \text{ then stuff. wait} \]

\[ \text{val}:=\text{buff}[\text{start}] \]
\[ \text{len}:=\text{len}-1; \text{ slots}:=\text{slots}+1 \]
\[ \text{start}:=\text{start}+1 \text{ mod } N+1 \]
\[ \text{space. continue} \]

begin

\[ \text{start}:=0 \]
\[ \text{len}:=0 \]
\[ \text{slots}:=0 \]

end BB
Suppose signal operation does not cause output?

If A runs after C.signal, is C.wait C.out A.B.C.
   C.out A.B.C. C.signal C.out
Then B runs signal/urgent semantics is: before new entry to monitor.
Implementation of signal/urgent 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 c
    - Invoke scheduler

- e.signal:
  - Put thread on urgent queue
    - If condition queue for c not empty:
      - then run head next
    - else invoke scheduler

Scheduler:

- Any thread on urgent queue?
  - Run it!
  - Else any thread on entry queue?
  - Run it!
Implementation of signal/urgent semaphores with semaphores:

\begin{verbatim}
M: monitor

op: operation(---)

C: count
if C.count > 0
then C.count := C.count - 1
else V(muxM)
else V(semC)
end

if urgentM > 0
then V(urgentM)
else V(muxM)
\end{verbatim}
CS4410 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 10/10
  + 1 week
  Likely review session 10/8 in class
Beyond Semaphores (Cont.)

M: Monitor

\[ \text{var} \quad \text{monitor vars} \ldots \]

C: Condition \( \in B_c \in \mathbb{E} \)

op: operation \((p_1, p_2, \ldots)\)

\[ \text{var loads} \]

\{ \]

\}

end

\vdots

C.wait

\text{thread suspended}

C.continue:

\text{thread exits Monitor & thread blocked on C runs}

vs

C.signal:

\text{thread suspends on "wqent" queue & thread blocked on C runs}
Acut: monitor

\[ \text{var } \text{amt}: \text{integer} \]
\[ \text{inc: condition} \]
\[ w \geq \text{amt}\] 

\[ \text{deposit: operaton (} w: \text{integer}) \]
\[ \text{amt} := \text{amt} + w \]
\[ \text{inc. signal} \]
\[ \text{end} \]

\[ \text{withdraw: operaton (} w: \text{integer}) \]
\[ \text{while } \text{amt} < w \text{ do} \]
\[ \text{inc. wait end} \]
\[ \text{end} \]
\[ \text{ant} := \text{amt} - w \]
\[ \text{end} \]
\[ \text{end Acut} \]
 Suppose signal operation does not cause yield.

If $B$ runs next...

- $c$.wait
- $c$.notify

Note: Actually I need not hold $B$ after notify because no process gets control at that point!
Implementing of sign/sync 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

c.notify: 1 process on condition queue for c
made runnable
continue executing in monitor

c.notifyAll: all process 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:

\[
\begin{align*}
\text{var} & \quad \text{locked: boolean init false} \\
\text{Q: condition} & \\
\text{Acquire: operation} & \\
\text{if locked then Q.wait} & \\
\text{while locked do Q.wait end} & \\
\text{locked := true} & \\
\end{align*}
\]

Release: operation

\[
\begin{align*}
\text{locked := false} & \\
\text{Q.notify} & \\
\end{align*}
\]

end monitor

--------------
Cautions when using mutexes

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 prio
B: med prio
C: low prio

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)
*: 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: mutual exclusion (for ops)
(3 ideas)
   - Cond sync (within ops)
   - Selective mutex (due to visibility rules for vars)

- Locks define regions

Syntax of a region:

```java
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 [sh+len] := 0
    len := len + 1; slots := slots - 1
    stuff.notify
end

region m do
    while len = 0 do stuff.wait end
    val := buff[sh]
    len := len - 1; slots := slots + 1
    sh := sh + 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

send \( m \) to dest
receive \( m \) from source

Various design decisions
How to specify destination and source of msg for send and for receive

1. Direct naming: sender names receiver receiver names sender

   \(O(N^2)\) channels

   Problem — Processes need to know each others names

2. Asymmetric direct naming: sender names receiver 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 Cl
- Receive val

**Service**

- Receive x,y,z
- Send res to
Buffer capacity

How many sent but not received msgs allowed?

O - capacity
N - capacity

send as Y?
receive as P?