CS4410 Announcements 10/1

- Tutorial on this week's lectures (10/1, 10/3)
  Sunday 10/6 3pm-430pm
  Hollister B14

- Looking ahead: Prelim 1 is Thurs 10/10
  +1 week (ish)
  Likely review session 10/8 in class

Beyond Semaphores (Con't)

```
M: Monitor
   var    ... Monitor vars...
          C: Condition        {Bc}

   op: operation (P1, P2 ... )
       var locals

           {
           }

       end
   :
```

C.wait — thread suspended

C.Continue: — thread exits Monitor & thread blocked on c runs

VS

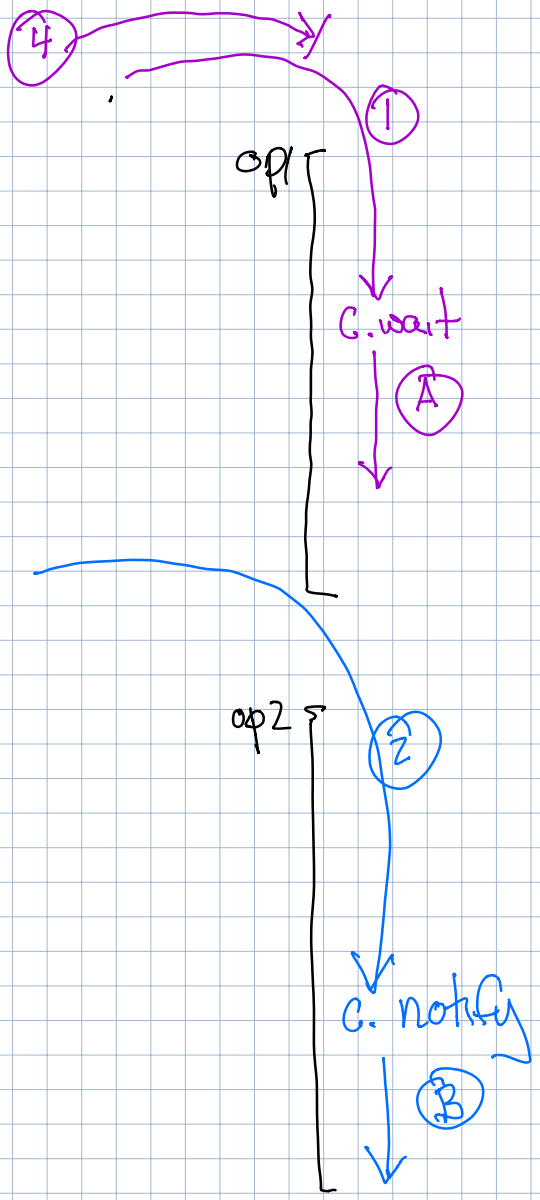
C.Signal: — thread suspends on "urgent" queue & thred Blocked on c runs

```
Acnt: monitor
      var amt: integer
          incr: condition   {w ≥ acnt}

deposit: operation (v: integer)
         amt := amt + v
                incr.signal
         end

withdraw: operation (w: integer)
          while amt < w do incr.wait end
{amt ≥ w}  amt := amt - w
           end

end Acnt
```



"notify"

Suppose signal operation does not cause yield.

If (B) runs next....

{I} C.wait {I}

{I} C.notify {I}

Note: Actually, I need not hold before/after notify because no process gets control at that point!

Implementation of ~~signal/wait~~ notify 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 que
else grant access

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

c.notify: 1 process on condition que for c
made runnable but
Continue executing in monitor

c.notifyAll: all process on condition que 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, C. signal } immediately

C. notify } eventually

Compare with P & V!

Use of notify is tricky

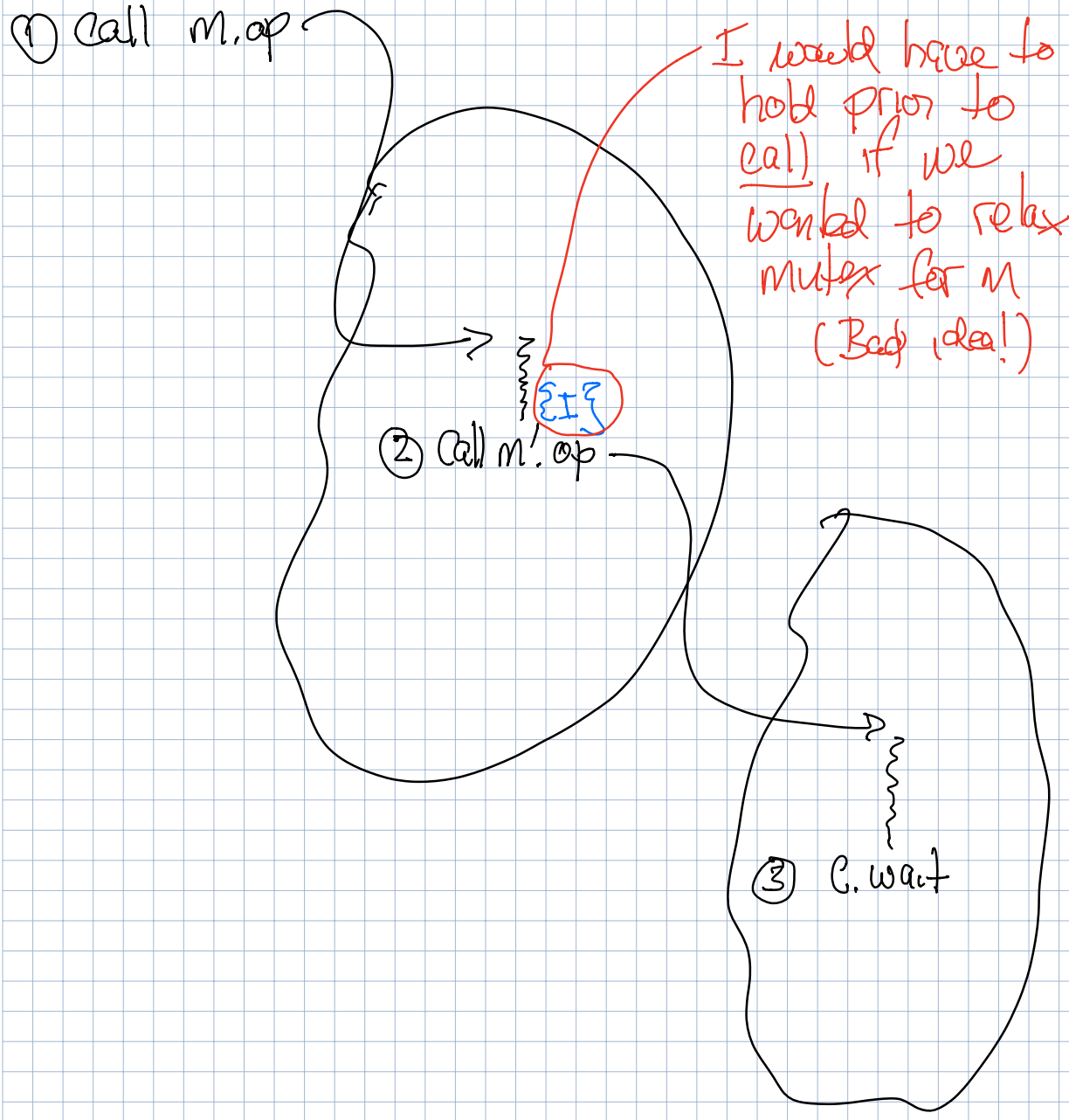
Two processes can Acquire when if instead of while.

```
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

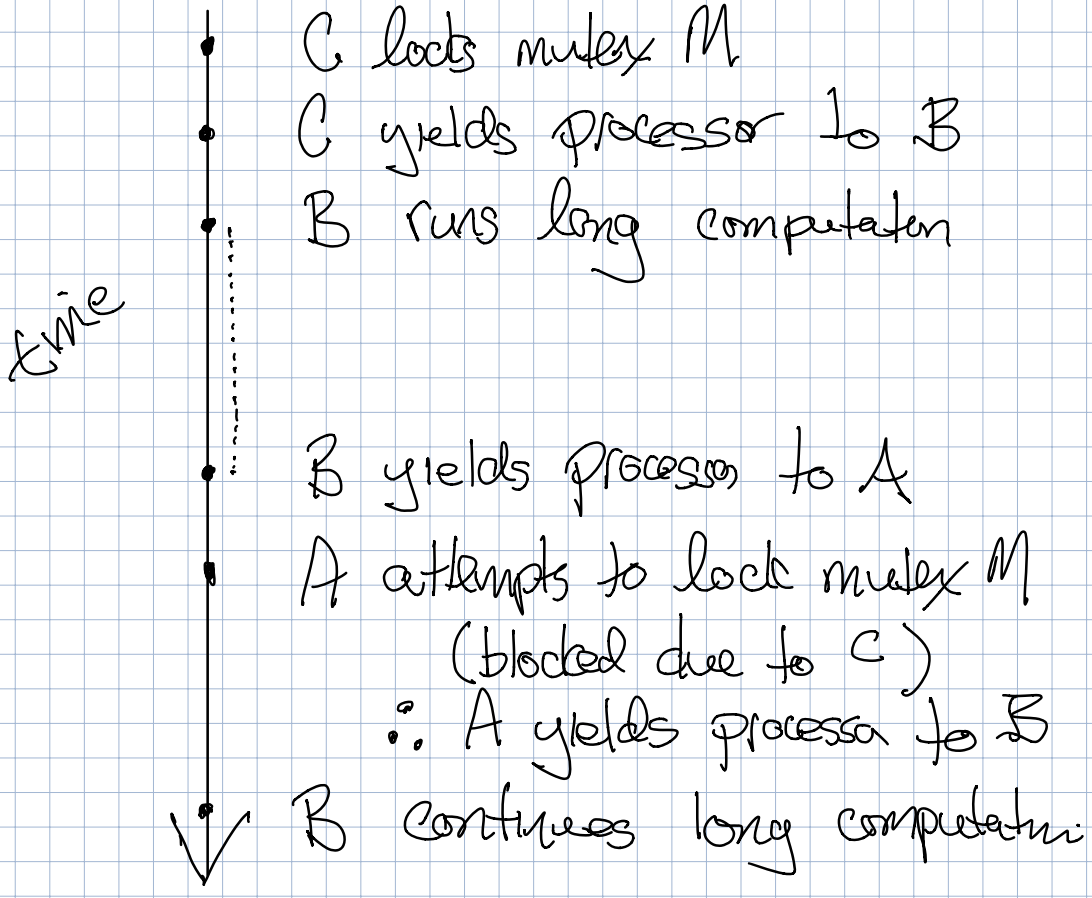
I Nested locking



11 Priority Inversion

Thread

A: high prio
B: med prio
C: low prio



Deconstructing Monitors: (Critical) Regions

Yet another synchronization structure.

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

Monitor (3 ideas):
- mutual exclusion (for ops)
- cond synch (within ops.)
- selective rules (due to visibility rules for vars)

- locks define regions

syntax of a region:

```
var m: lock

region m do
    .
    .
    .
end
```

- Condition variables allow release of locks (wait, notify, notifyAll)

```
var m: lock
    space: condition with m
    stuff: condition with m

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

region m do
  while len=0 do space.wait end
  val := buff[sht]
  len := len-1;  slots := slots+1
  sht := sht+1 mod N+1
  space.notify
  end
```

- associate sets of variables with each lock.
  - allows arbitrary fine-gran 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 destnatn & soure of msg

↑ for send ↑ for recve

① direct namy: sender names receiver
receiver names sender

$O(N^2)$ chanels

Problem — Processes need to know each other's names

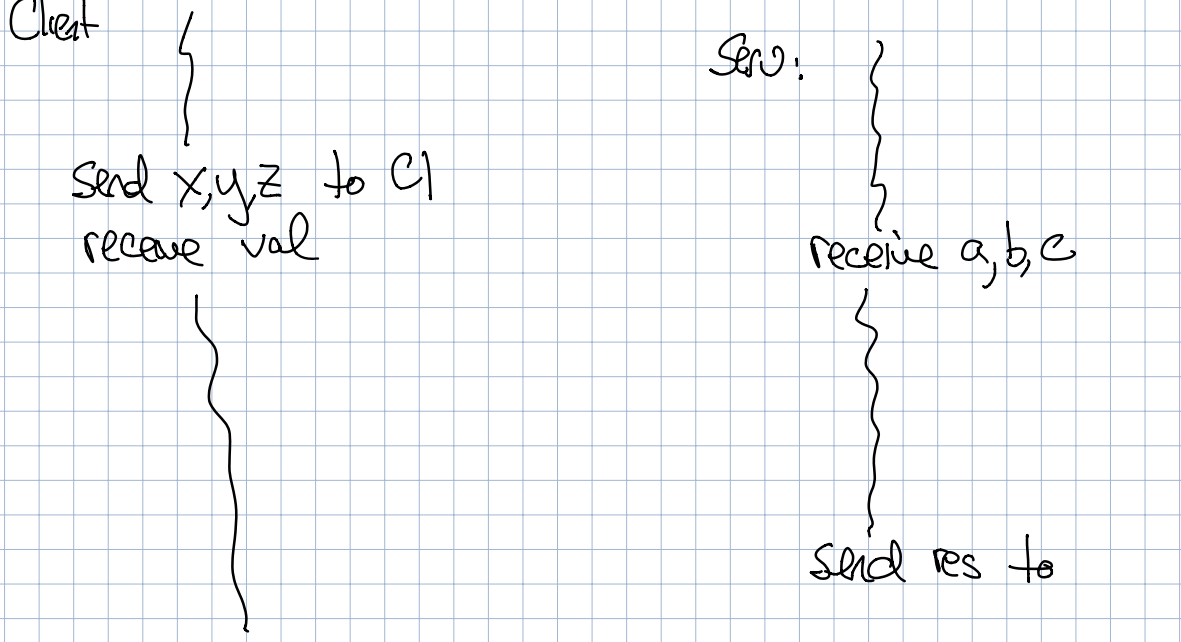
② asymetric direct namy: sender names receiver
receiver names nobody

Synchrony

blocking ("synchronous"): causes primitive to delay until some event

nonblocking ("asynchronous"): primitive continues

blocking send: sender delayed until msg received



Buffering capacity

How many sent but not received msgs allowed

0 – capacity

N – capacity

send as V ?

receive as P .