CS4410 Announcements

- 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

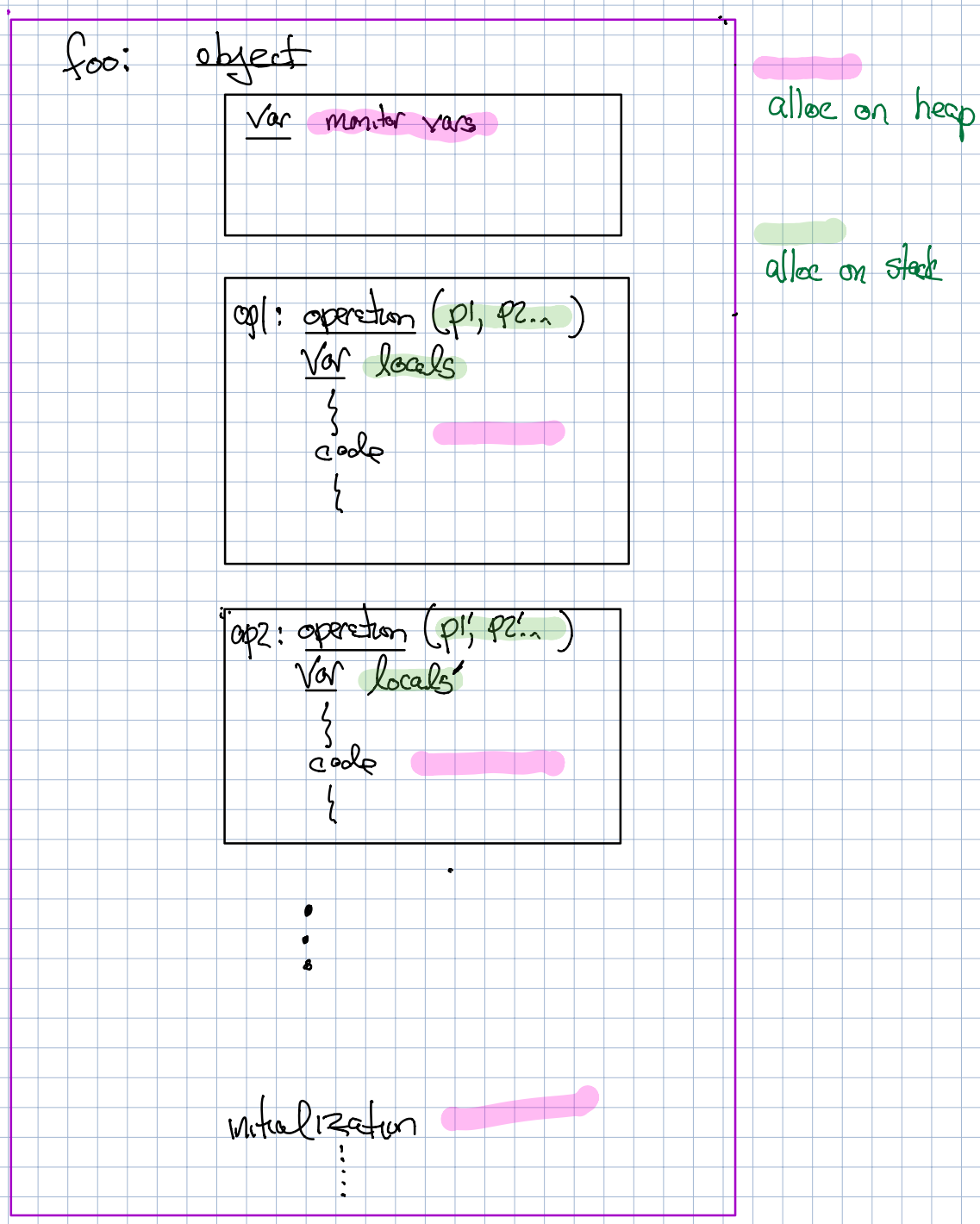
Beyond Semaphores (9/26/2019 CS4410 FBS)

Synchronization primitives
- shared memory
- semaphores
- monitors & regions

Tension:
- generality / expressiveness
- vs
- usability / clarity

↳ avoiding errors!

Exploit:
- syntax
- compilers (= syntax checking)
- run-time



alloc on heap

alloc on stack

schematic representation of monitor

Support for synchronization ≃

mutex + cond synch

per operation

op. operation (p1, p2 ...)

var locals

Require that B holds (*) ~~await slots ≥ N~~ ...

end

```
BB:  monitor
     var  buff[0..N] of parton
          strt: 0..N
          len:  0..N+1
          slots: 0..N+1

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

rmv: operation (var val: parton)
          need a parton!
          val := buff[strt]
          len := len - 1;  slots := slots + 1
          strt := strt + 1 mod N+1

begin
          strt := 0
          len := 0
          slots := 0
end
```

Condition variables

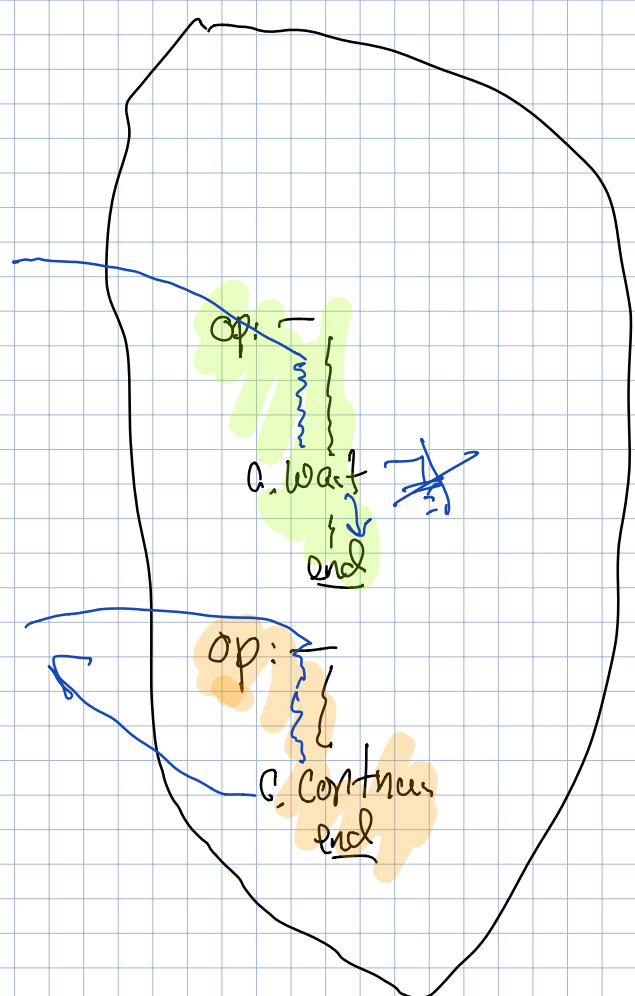
a Boolean

var C : condition ⟶ associate B_C

C.wait

① releases mutex lock for Monitor
② blocks process

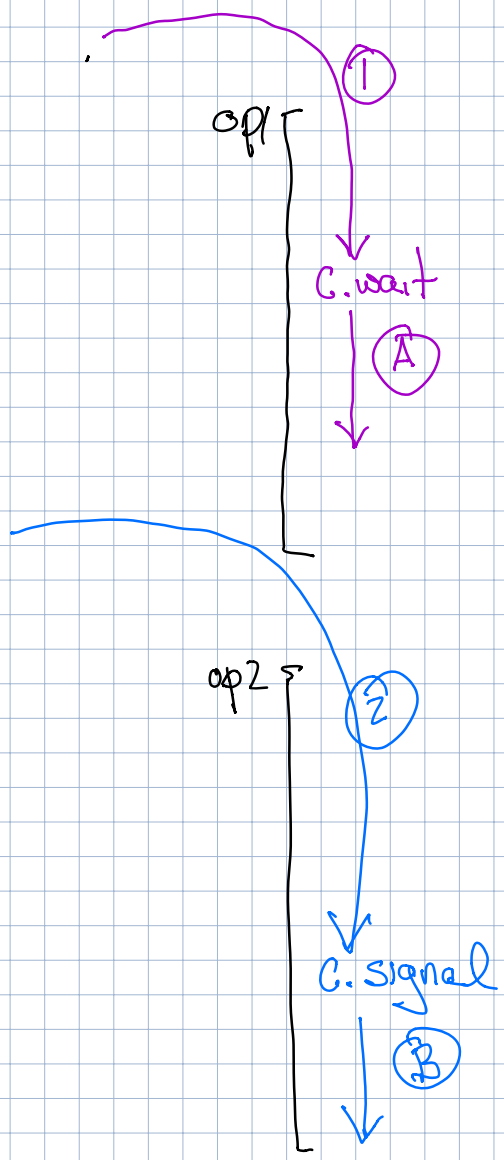
C.continue : exit monitor & unblock 1 process



$\{I\}$ C.wait $\{I \wedge B_C\}$

$\{I \wedge B_C\}$ C.continue $\{I \wedge B_C\}$

```
BB:  monitor
     var  buff[0..N] of parton
          strt:  0..N
          len:   0..N+1
          slots: 0..N+1
          space: condition      { slots > 0 }   <- B_space
          stuff: condition      { len = 0 }     <- B_stuff

insert:  operation (v: parton)
         ~~need a slot!~~   if slots=0 then space.wait
         buff[strt + len] := v
         len := len + 1;  slots := slots - 1
         stuff.continue

rmv: operation (var val: parton)
         ~~need a parton!~~   if len=0 then stuff.wait
         val := buff[strt]
         len := len - 1;  slots := slots + 1
         strt := strt + 1 mod N+1
         space.continue

begin
         strt := 0
         len := 0
         slots := 0
end BB
```



Suppose signal operation does not cause blt?

If (A) runs after c.signal

$\{I\}$ c.wait $\{I \wedge B_c\}$

$\{I \wedge B_c\}$ c.signal $\{I\}$

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

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

c.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 semantics with semaphs

var $mutex_M$: semaphore init (1)
sem_c : semaphore init (0)
urg_M : semaphore init (0)

M: monitor

{

op: operation(...) ← P($mutex_M$)

{

c.wait

{

c.signal

{

end

c.count := c.count + 1
if $urgcount_M > 0$
then $urgcount_M := urgcount_M - 1$
V(urg_M)
else V($mutex_M$)
P(sem_c)

$urgcount_M := urgcount_M + 1$
if c.count > 0
then c.count := c.count − 1
V(sem_c)
P(urg_M)

if $urgcount_M > 0$ then V(urg_M)
else V($mutex_M$)

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 ☹
  Likely review session 10/8 in class

Beyond Semaphores (Con't)

```
M: Monitor
   var      ... Monitor vars...
            c : Condition      {B_c}

   op:  operation (p1, p2 ... )
        var locals
                {
                }
        end
   :
```

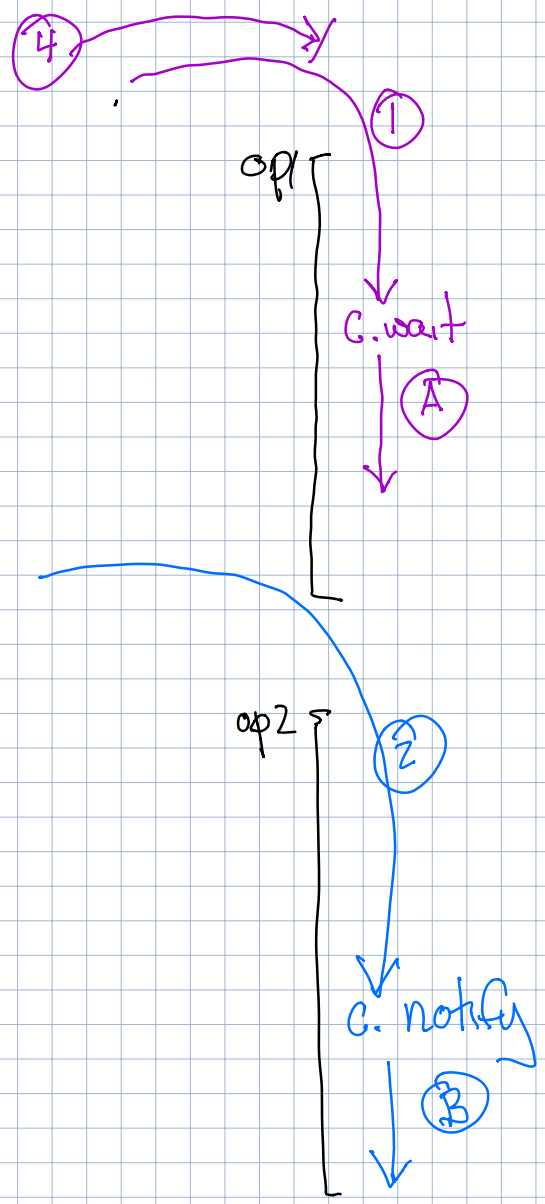
c.wait — thread suspended

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

VS

c.signal: — thread suspends on "urgent" queue & thread 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/urgent~~ 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 queue
else grant access

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

c.notify: 1 process on condition queue for c
made runnable but
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, C.signal } immediately

C.notify } eventually

Compare with P & V!

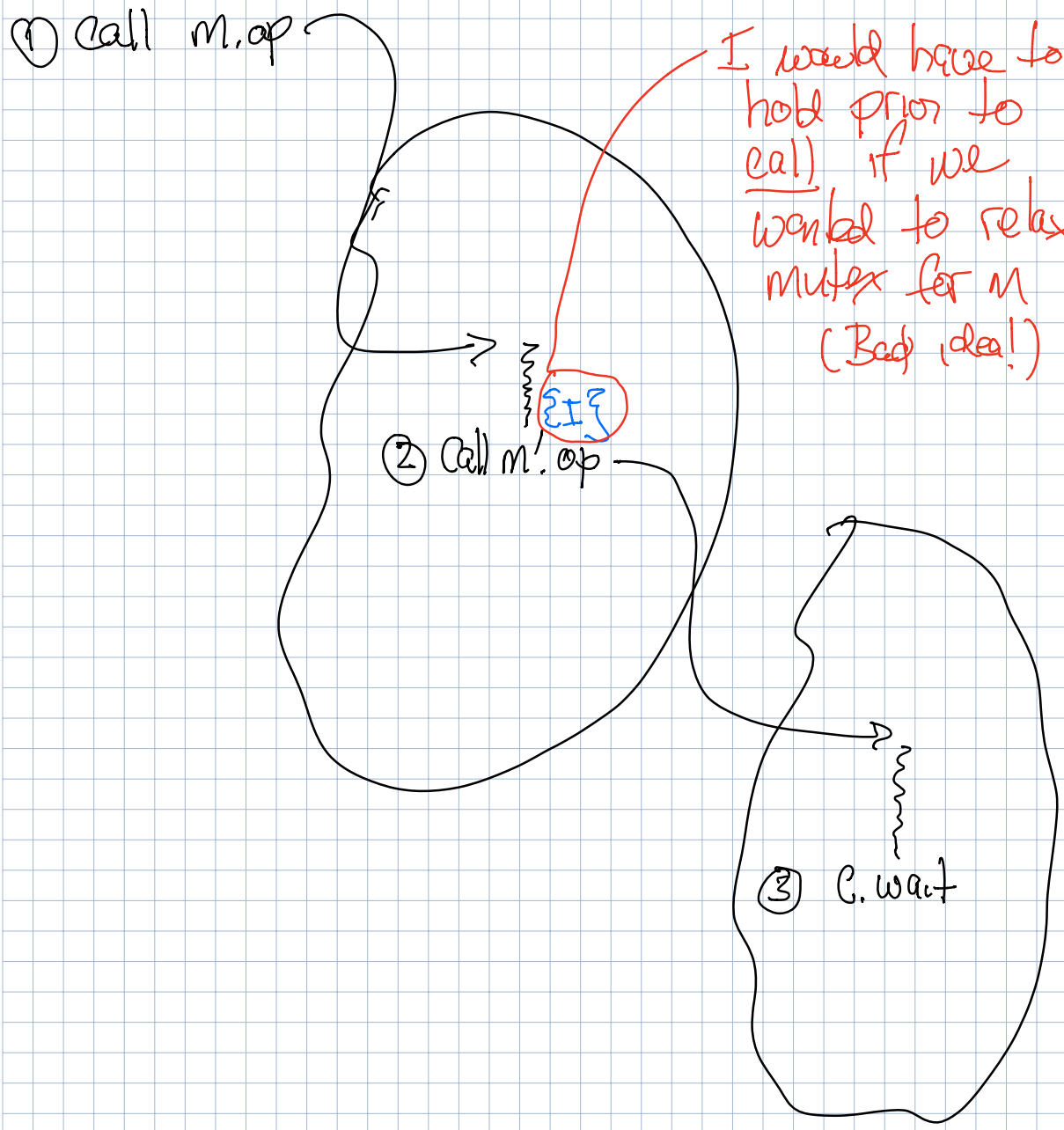
Use of <u>notify</u> is tricky

Two processes can Acquire when <u>if</u> instead of <u>while</u>.

```
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



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

II 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 primitive.

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

Monitor: (3 ideas)
- mutual exclusion (for ops)
- cond synch (within ops.)
- selective mutex (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 N-len = 0 do space.wait end
        buff[(sht+len) mod N] := val
        len := len +1
        stuff.signal
        end

    }

    region m do
        while len = 0 do stuff.wait end
        val := buff[sht]
        sht := (sht+1) mod N
        len := len -1
        space.signal
        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 destination & source of msg

↑ for send ↑ for receive

① direct naming: sender names receiver
receiver names sender

$O(N^2)$ channels

Problem — Processes need to know each other's names

② asymetric direct naming: 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

Client

send x,y,z to C1
receive val

Serv:

receive a,b,c

send res to

Buffering capacity

How many sent but not received msgs allowed

0 – capacity
N – capacity

send as V ?

receive as P ?