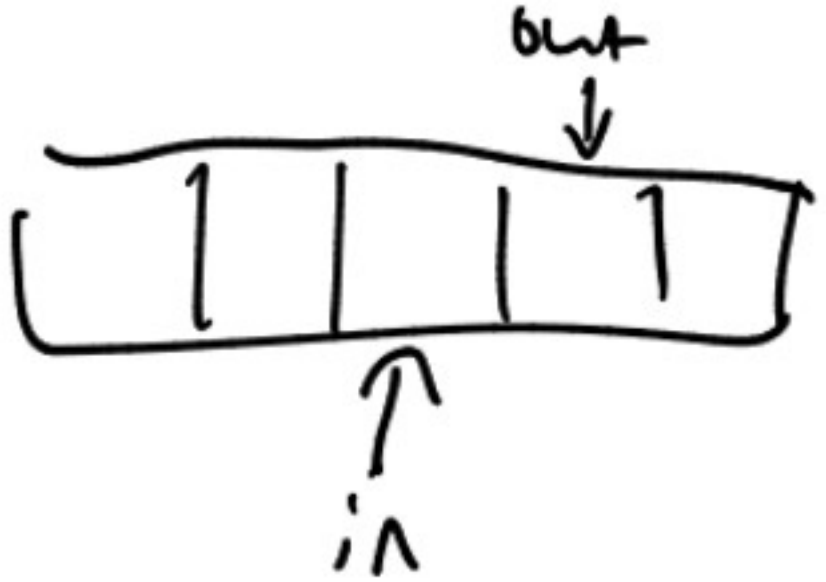
4/4/10 Lecture 9: Mesa-style monitors

- Reader/Writer problem
- MESA-style monitors
- optimization

- (time permitting) conditions for deadlock

Bounded buffer w/ Hoare-style monitor



mutual exclusion

```
class Buffer (Monitor):
    def __init__(self, N):
        self.N = N
        self.in = 0
        self.out = 0
        self.buf = Object[N]

    def hasSpace()
        in ≠ out - 1
```

```
def put(object o):
    • wait until hasSpace()
    buf[in++] = o
```

Reader-writer lock

or acquire/release

- have methods to enter/leave as a reader
  enter/leave as a writer

- Either have (- any # of readers / or - one writer) at a time

```
class RWLock (~~Monitor~~):
    def __init__(self):
        self.lock = Lock()
        self.num_readers = 0
        self.writer = False

        # invariant:
        #   either writer is False
        #     or num_readers = 0
        self.no_writers = Condition(lock)
            # predicate writers = False
        self.no_readers_writers = Condition(lock)
            # predicate: readers = 0 and writers = False

    # note: uses notify ~~All~~

    def enter_reader(self):
        with self.lock:
            # wait until no writers
            while not (writers = False):
                no_writers.wait()
            num_readers ++

    def leave_reader(self):
        with lock:
            num_readers --
            self.no_readers_writers.notifyAll()

    def enter_writer(self):
        with self.lock:
            while not (readers = 0 and writers = False):
                self.no_readers_writers.wait()
            self.writers = True

    def leave_writer(self):
        with self.lock:
            self.writers = False
            self.no_writers.notifyAll()
            self.no_readers_writers.notifyAll()
```

only thing connecting condition variable to predicate are ① comment ② while loop where we wait

Condition needs mon. lock so it can acquire & release it

first writer to wake immediately make predicate false, so no reason to notify all threads, one is enough.

the predicate.

```
Class Example:
    _init_
      lock
      X = True
      X_false = Cond(lock)
        # predicate: x is false
      # notify ~~All~~
```

```
def f():
    while not (x = False):
        x_false.wait()
    x = True
```

```
def g():
    while not (x = False):
        x_false.wait()
    ~~return~~
    x_false.notify()
```

```
def h():
    X = False
    X_false.notify ~~All~~()
```

- might only notify thread calling g.

- f can make progress after g runs (because x = True) but is still sleeping because g notified.

Writing monitor

1. write in Hoare style:
   - think about state to maintain
   - think about preconditions, postconditions, invariants (write down)

2. add condition variables with comments indicating predicates

```
while not (predicate):
    cv.wait()
```
to wait;

3. compare state changes to predicates, add notifyAll when I make a predicate true

4. optimize by changing notifyAll's to notify where appropriate, make sure that all fn's that wait for that predicate make it False before returning.

WARNING!

- use monitor design pattern:
  - all CVs always have associated predicates (boolean exprs involving member vars)
  - while not(pred):
    cv.wait()

Lock

with lock:
- acquiring mutual exclusion lock upon entering, releasing it when you leave

(like a waiting room)

Condition class

- wait: ① release monitor lock
  ② sleeping until notifyAll called
  ③ reacquire monitor lock.
- notifyAll: wakes up all sleeping threads

shouting into waiting room.

- notify: wakes up one sleeping thread.

- instead of notifying when we <u>might</u> have made predicate true, check & only notify when we <u>did</u> make it true

```
def leave_reader():
    with lock:
        readers--
        if readers = 0:          <- optimize
            no_readers.notifyAll()
```

- use notify if only one thread will be able to make progress.

General synch. advice:

1. use high-level primitives wherever possible:
   - bounded buffer or RW lock provided by OS / language / standard library
   - monitors
   - semaphores

   blocking primitives

   - spin locks, TAS, CAS, ... ⟸ occasionally need lock-free data structures

2. don't mix primitives
   - e.g. monitor code (outside of wait) shouldn't block, since it holds the lock.

3. document your code, use good var names, write down predicates & invariants & specifications, ...

Monitor implementation (pseudocode on website)

- very similar to semaphore impl:
  - spin lock to protect internal state
  - queue of processes trying to acquire monitor lock
    - add to queue when "with lock"
    - dequeue & run when "with lock" returns
  - queue of processes waiting on each cond. var.
    - add to queue when wait()
    - remove & ~~run~~ on notify
      add to monitor lock queue.

Java "syncronized" blocks are like monitors.

- Every object can be made a monitor by using "synchronized"

- Every object is a CV.