Lecture 9: Mesa-style monitors

- Reader/Writer problem
- Mesa-style monitors
- Optimization

(time permitting) Conditions for deadlock
Bounded buffer w/ Hoare-style monitor

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

    def hasSpace(self):
        return n != out - 1

    def put(object o):
        if not hasSpace():
            wait until hasSpace()
        buf[in++] = o
```


**Condition**

Release it if it can acquire:

- First write immediately
- Reason to make prediction false, so no threads
- Write immediately

```
class RWLock:  
    def __init__(self):  
        self.no_reader_writer = CthreadingLocal()  
        self.no_reader_writer.value = False

    def enter_writer(self):  
        self.writer.acquire()  
        if not self.no_reader_writer.value:  
            self.writer.release()  
            self.reader.release()

    def exit_writer(self):  
        self.writer.release()

def leave_reader(self):  
    self.reader.release()

def enter_reader(self):  
    self.reader.acquire()  
    if not self.no_reader_writer.value:  
        self.reader.release()  
        self.writer.release()

def main():  
    rwlock = RWLock()  
    writer = RWLock.writer()  
    reader = RWLock.reader()

```

Either have (any # of readers) at a time or (one writer) at a time.
def f():
    while not (x = False):
        x = False, wait()
    x = True
    return
    x = False, notify()
Writing monitor

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

2. add condition variables
   
   while not (predicate):
   
   with comments indicating
   
   predicates
   
   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 fs that wait for that predicate make it false before returning

WARNING:

- use monitor design pattern:

  • all CVs always have associated predicates
    (broken expr involving member vars)

  • while not (pred):
    
    cv.wait()
Lock

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

Condition class
- release monitor lock
- wait & sleeping until notifyAll called
- reacquire monitor lock
- notifyAll: wakes up all sleeping threads
- notify: wakes up one sleeping thread

(like a waiting room)
- Instead of notifying when we might have made predicate true, check if only notify when we did make it true.

```python
def leave_reader():
    with lock:
        readers -=
        optimize
        if readers == 0:
            no_readers. notify All()

- Use notify if only one thread will be able to make progress.
General synchronisation advice:

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

1. very similar to semaphore mpi:
   1. spin lock to protect internal state
   2. queue of processes trying to acquire monitor lock
      - add to queue when "with lock"
      - dequeue and run when "with lock" returns
   3. queue of processes waiting on each cond. var.
      - add to queue when wait()
      - remove and take on notify
      - add to monitor lock queue
Java "synchronized" blocks are like monitors.
- Every object can be made a monitor by using "synchronized"
- Every object is a CV.