Previously, on CS4410...
Back to Split Binary Semaphores

Nurse’s office: critical section protecting variables that determine when to wait
Rooms: waiting conditions

At any time, exactly one semaphore or thread is green (and thus, at most one semaphore is green (Invariant))
Split Binary Semaphores

Nurse’s office: critical section protecting variables that determine when to wait

Rooms: waiting conditions

At any time, exactly one semaphore or thread is green (and thus, at most one semaphore is green (Invariant))
If $n$ readers in the critical section, then $n_{\text{readers}} \geq n$

$n_{\text{readers}}$ incremented inside R/W lock before entering the CS (i.e., the database)
Two Types of Monitors

**Hoare Monitors**

Tony Hoare

**Mesa Monitors**

Butler Lampson

Different semantics as to what happens when a thread waiting on a condition is alerted that the condition holds.
Hoare Monitors
Tony Hoare, 1974

Syntactic sugar above split binary semaphores

- **monitor**: one thread can execute at a time
- **wait**(cond. var.): thread waits for given condition
- **signal**(cond. var.): transfer control to a thread waiting for the given condition, if any

Similar construct proposed by Per Brinch Hansen in 1973
import synch

def Monitor() returns monitor:
    monitor = synch.Lock()

def enter(mon):
    synch.acquire(mon)

def exit(mon):
    synch.release(mon)

def Condition() returns condition:
    condition = { .sema: synch.BinSema(True), .count: 0 }

def wait(cond, mon):
    cond->count += 1
    exit(mon)
    synch.acquire(?cond->sema)
    cond->count -= 1

def signal(cond, mon):
    if cond->count > 0:
        synch.release(?cond->sema)
        enter(mon)

a no-op if no one is waiting!
What happens when a thread signals?

**Hoare semantics:**

- Signaling thread is suspended and, atomically, ownership of the lock is passed to one of the waiting threads, whose execution is immediately resumed.

- Signaling thread is resumed if former waiter exits monitor, or if it waits again
Producer/Consumer with Bounded Buffer

```python
import hoare

def BoundedBuffer(size) returns buffer:
    buffer = {
        .mon: hoare.Monitor(),
        .prod: hoare.Condition(), .cons: hoare.Condition(),
        .buf: { x:() for x in {1..size} },
        .head: 1, .tail: 1,
        .count: 0, .size: size
    }

    def put(bb, item):
        hoare.enter(?bb→mon)
        if bb→count == bb→size:
            hoare.wait(?bb→prod, ?bb→mon)
        bb→buf[bb→tail] = item
        bb→tail = (bb→tail % bb→size) + 1
        bb→count += 1
        hoare.signal(?bb→cons, ?bb→mon)
        hoare.exit(?bb→mon)
```
Producer/Consumer with Bounded Buffer

```python
import hoare

def BoundedBuffer(size) returns buffer:
    buffer = {
        .mon: hoare.Monitor(),
        .prod: hoare.Condition(), .cons: hoare.Condition(),
        .buf: { x(): for x in {1..size} },
        .head: 1, .tail: 1,
        .count: 0, .size: size
    }

    def put(bb, item):
        hoare.enter(?bb→mon)
        if bb→count == bb→size:
            hoare.wait(?bb→prod, ?bb→mon)
        bb→buf[bb→tail] = item
        bb→tail = (bb→tail % bb→size) +
        bb→count += 1
        hoare.signal(?bb→cons, ?bb→mon)
        hoare.exit(?bb→mon)
```
Producer/Consumer with Bounded Buffer

def get(bb) returns next:
    hoare.enter(?bb→mon)
    if bb→count == 0:
        hoare.wait(?bb→cons, ?bb→mon)
    next = bb→buf[bb→head]
    bb→head = (bb→head % bb→size) + 1
    bb→count -= 1
    hoare.signal(?bb→prod, ?bb→mon)
    hoare.exit(?bb→mon)

enter monitor

wait if empty

exit monitor

signal() passes the baton immediately if there are waiting producers
Mesa Monitors
Mesa Language, Xerox PAak 1980

- Syntactically similar to Hoare monitors
  - monitors and condition variables

- Semantically closer to busy waiting
  - `wait(cond. var.)`: wait for condition, but may get back the CPU when condition is not satisfied (!)
  - `notify(cond. var.)`: move to ready queue a thread waiting for the condition, if any, but don’t transfer control (i.e., give the CPU) to it
  - `notifyAll(cond. var.)`: move to ready queue all threads waiting for the condition, but don’t transfer control (i.e., give the CPU) to any of them
What are the implications?

**Hoare**
- Signaling is atomic with the resumption of waiting thread
  - shared state cannot change before waiting thread is resumed
  - safety requires to signal only when condition holds
- Shared state can be checked using an if statement
- Makes it easier to prove liveness
- Tricky to implement

**Mesa**
- notify() and notifyAll() are hints
  - adding them affects performance, never safety
- Shared state must be checked in a loop (the condition could have changed since the thread was notified!)
- Simple implementation
- Resilient to spurious wakeup
# Hoare vs Mesa Monitors

<table>
<thead>
<tr>
<th>Hoare Monitors</th>
<th>Mesa Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baton passing approach</td>
<td>If at first you don’t succeed... sleep &amp; try again when the stars seem aligned!</td>
</tr>
<tr>
<td>signal passes baton</td>
<td>notify(all) moves waiting threads back to ready queue</td>
</tr>
<tr>
<td>used by most books</td>
<td>used by most real systems</td>
</tr>
</tbody>
</table>

Mesa monitors won the test of time...
def Condition() returns condition:
  condition = bag.empty()

def wait(c, lk):
  var cnt = 0
  let _, ctx = save():
    atomically:
      cnt = bag.multiplicity(!c, ctx)
      !c = bag.add(!c, ctx)
      !lk = False
    atomically when (not !lk) and (bag.multiplicity(!c, ctx) <= cnt):
      !lk = True

def notify(c):
  atomically if !c != bag.empty():
    !c = bag.remove(!c, bag.bchoose(!c))

def notifyAll(c):
  !c = bag.empty()
Reader/Writer Lock Specification (again)

```python
def RWlock() returns lock:
    lock = { .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    atomically when rw→nwriters == 0:
        rw→nreaders += 1

def read_release(rw):
    atomically rw→nreaders -= 1

def write_acquire(rw):
    atomically when (rw→nreaders + rw→nwriters) == 0:
        rw→nwriters = 1

def write_release(rw):
    atomically rw→nwriters = 0
```

Better to assert rw → nreaders > 0
Reader/Writer lock with Mesa monitors

Invariants

- If $n$ readers in the critical section, then $n_{\text{readers}} \geq n$
- If $n$ writers in the critical section, then $n_{\text{writers}} \geq n$
- $(n_{\text{readers}} \geq 0 \land n_{\text{writers}} = 0) \lor (n_{\text{readers}} = 0 \land n_{\text{writers}} = \leq 1)$

```python
from synch import *

def RWlock() returns lock:
    lock = {
        .nreaders: 0, .nwriters: 0, .mutex: Lock(),
        .r_cond: Condition(), .w_cond: Condition()
    }
```

It is the mutex that protects $n_{\text{readers}}$ and $n_{\text{writers}}$, not the R/W lock!
def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
        rw→nreaders += 1
    release(?rw→mutex)

def read_release(rw):
    acquire(?rw→mutex)
    rw→nreaders -= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→mutex)

Similar to Busy Waiting

but needs this
R/W Lock, Writer

```python
def write_acquire(rw):
    acquire(rw->mutex)
    while (rw->nreaders + rw->nwriters) > 0:
        wait(rw->w_cond, rw->mutex)
    rw->nwriters = 1
    release(rw->mutex)

def write_release(rw):
    acquire(rw->mutex)
    rw->nwriters = 0
    notifyAll(rw->r_cond)
    notify(rw->w_cond)
    release(rw->mutex)
```

Similar to Busy Waiting

don't forget anyone!
## Conditional Critical Sections

Let me count the ways...

<table>
<thead>
<tr>
<th>Busy Waiting</th>
<th>Split Binary Semaphores</th>
<th>Mesa Monitors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use a lock and a loop</td>
<td>Use a collection of binary semaphores</td>
<td>Use a lock, a collection of condition variables, and a loop</td>
</tr>
<tr>
<td>Easy to write the code</td>
<td>Just follow the recipe</td>
<td>Notifying is tricky</td>
</tr>
<tr>
<td>Easy to understand the code</td>
<td>Tricky to understand if you don’t know the recipe</td>
<td>Easy to understand the code</td>
</tr>
<tr>
<td>Ok-ish for true multicore, but bad for virtual threads</td>
<td>Good for virtual threading. Thread only runs when it can make progress</td>
<td>Good for both multicore and virtual threading</td>
</tr>
</tbody>
</table>