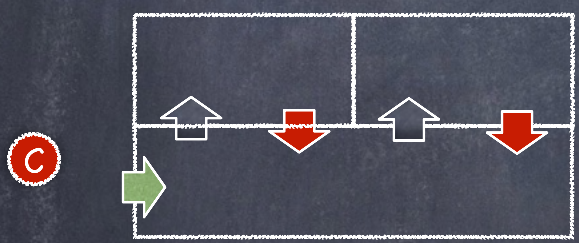
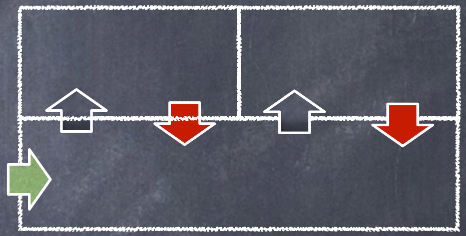
# Previously, on CS4410...

# Back to
# Split Binary Semaphores



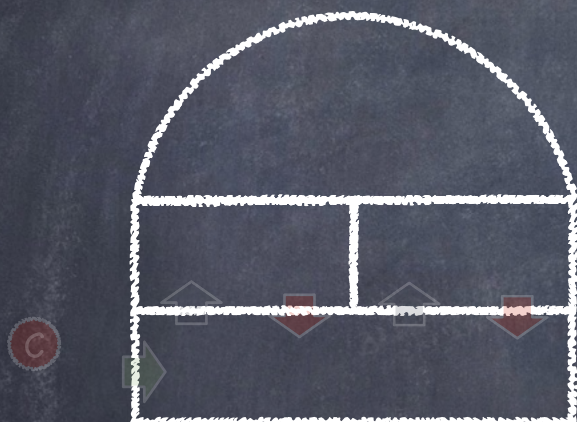



Nurse's office: critical section protecting variables that determine when to wait

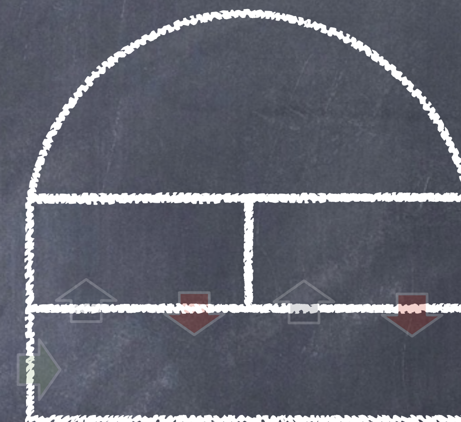
Rooms: waiting conditions

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

# Back to
# Split Binary Semaphores



R/W Lock



R/W Lock

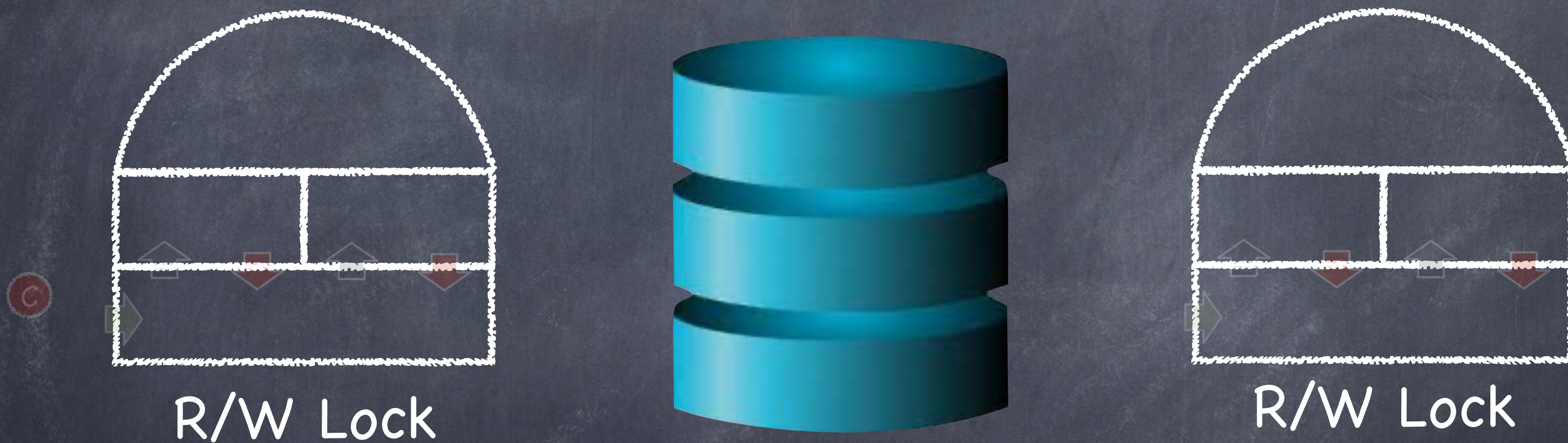
Nurse's office: critical section protecting variables that determine when to wait

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 $nreaders \geq n$

**WHY?**




$nreaders$ incremented inside R/W lock before entering the CS (i.e., the database)

# Two Types of Monitors

## Hoare Monitors

## Mesa Monitors

Tony Hoare


Butler Lampson

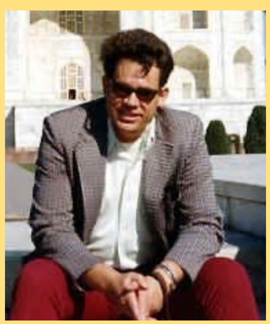
Different semantics as to what happens when a thread waiting on a condition is alerted that the condition holds

# Hoare Monitors

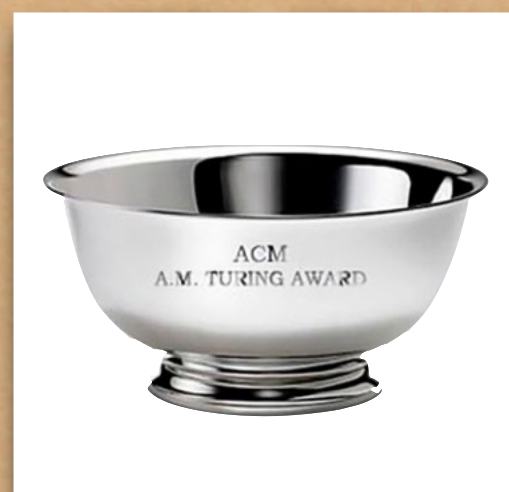
Tony Hoare, 1974

- 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*



# Hoare Monitors in Harmony

```
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)
```

main gate

waiting gate

passes control immediately

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

```
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)
```

circular buffer

enter monitor

wait if full

signal a consumer

exit monitor

# Producer/Consumer with Bounded Buffer

```
import hoare

def BoundedBuffer(size) returns buffer:
    buffer = {
            .mon: hoare.Monitor(),
            .prod: hoare.Condition(), .cons: h
            .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)
```

circular buffer

enter monitor

wait if full

signal() passes the baton immediately if there are waiting consumers

exit monitor

# 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

signal a producer

exit monitor

signal() passes the baton immediately if there are waiting producers

# Mesa Monitors

Mesa Language, Xerox PAak 1980

Very different from Hoare's monitors

- 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

| Hoare Monitors | Mesa Monitors |
| --- | --- |
| Baton passing approach | If at first you don't succeed... sleep & try again when the stars seem aligned! |
| signal passes baton | notify(all) moves waiting threads back to ready queue |
| used by most books | used by most real systems |

*Mesa monitors won the test of time...*

# Mesa Monitors in Harmony

```
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()
```

Condition: consists of a bag of threads waiting

wait: unlock+add thread context to bag of waiters

notify: remove one waiter from the bag of suspended threads

notifyAll: remove all waiters from the bag of suspended threads

# Reader/Writer Lock Specification (again)

```
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* $rw \rightarrow nreaders > 0$

# Reader/Writer lock with Mesa monitors

```
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 nreaders and nwriters, not the R/W lock!*

## Invariants

- If $n$ readers in the critical section, then $nreaders \geq n$
- If $n$ writers in the critical section, then $nwriters \geq n$
- $(nreaders \geq 0 \wedge nwriters = 0) \vee (nreaders = 0 \wedge nwriters = \leq 1)$

# R/W Lock, Reader

```
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

```
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…

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