Conditional Waiting
Review

- Concurrent Programming is Hard!
  - Non-Determinism
  - Non-Atomicity
- **Critical Sections** simplify things by avoiding data races
  - mutual exclusion
  - progress
- Need both mutual exclusion and progress!
- Critical Sections use a **lock**
  - Thread needs lock to enter the critical section
  - Only one thread can get the section’s lock
How to get more concurrency?

**Idea:** allow multiple read-only operations to execute concurrently

- Still no data races
- In many cases, reads are much more frequent than writes

**reader/writer lock**

Either:
- multiple readers, or
- a single writer

thus not:
- a reader and a writer, nor
- multiple writers
Conditional Waiting

• Thus far we’ve shown how threads can wait for one another to avoid multiple threads in the critical section

• Sometimes there are other reasons:
  • Wait until queue is non-empty
  • Wait until there are no readers (or writers) in a reader/writer lock
  • …
def RWlock():
    result = { .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
Reader/Writer Lock Specification

Invariants:

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

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

def read_acquire(rw):
    atomically when $rw\rightarrow n\text{writers} == 0$:
    $rw\rightarrow n\text{readers} += 1$

def read_release(rw):
    atomically $rw\rightarrow n\text{readers} -= 1$

def write_acquire(rw):
    atomically when $(rw\rightarrow n\text{readers} + rw\rightarrow n\text{writers}) == 0$:
    $rw\rightarrow n\text{writers} = 1$

def write_release(rw):
    atomically $rw\rightarrow n\text{writers} = 0$
```
import RW

const NOPS = 3

rw = RW.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ "read", "write" }) == "read":
            RW.read_acquire(?rw)
            rcs: assert (countLabel(rcs) >= 1) and (countLabel(wcs) == 0)
            RW.read_release(?rw)
        else:  # write
            RW.write_acquire(?rw)
            wcs: assert (countLabel(rcs) == 0) and (countLabel(wcs) == 1)
            RW.write_release(?rw)

    for i in {1..NOPS}:
        spawn thread()
Cheating R/W lock implementation

```python
import synch

def RWlock():
    result = synch.Lock()

def read_acquire(rw):
    synch.acquire(rw);

def read_release(rw):
    synch.release(rw);

def write_acquire(rw):
    synch.acquire(rw);

def write_release(rw):
    synch.release(rw);
```

The lock protects the application’s critical section
Cheating R/W lock implementation

```python
import synch

def RWlock():
    result = synch.Lock()

def read_acquire(rw):
    synch.acquire(rw);

def read_release(rw):
    synch.release(rw);

def write_acquire(rw):
    synch.acquire(rw);

def write_release(rw):
    synch.release(rw);
```

Allows only one reader to get the lock at a time

Does *not* have the same behavior as the specification
- it is missing behaviors
- no bad behaviors though
Busy Waiting Implementation

```python
from synch import Lock, acquire, release

def RWlock():
    result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders -= 1
    release(?rw→lock)

def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

def write_release(rw):
    acquire(?rw→lock)
    rw→nwriters = 0
    release(?rw→lock)
```

The lock protects nreaders and nwriters, not the critical section of the application.

waiting conditions
Busy Waiting Implementation

```python
from synch import Lock, acquire, release

def RWlock():
    result = { .lock: Lock(), .nreaders: 0, .nwriters: 0 }

def read_acquire(rw):
    acquire(?rw->lock)
    while rw->nwriters > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nreaders += 1
    release(?rw->lock)

def read_release(rw):
    acquire(?rw->lock)
    rw->nreaders -= 1
    release(?rw->lock)

def write_acquire(rw):
    acquire(?rw->lock)
    while (rw->nreaders + rw->nwriters) > 0:
        release(?rw->lock)
        acquire(?rw->lock)
    rw->nwriters = 1
    release(?rw->lock)

def write_release(rw):
    acquire(?rw->lock)
    rw->nwriters = 0
    release(?rw->lock)
```

Good: has the same behaviors as the implementation

Bad: process is continuously scheduled to try to get the lock even if it’s not available

(Harmony complains about this as well)
Mesa Condition Variables

• A lock can have one or more *condition variables*
• A thread that holds the lock but wants to wait for some condition to hold can temporarily release the lock by *waiting* on some condition variable
• Associate a condition variable with each “waiting condition”
  • reader: no writer in the critical section
  • writer: no readers nor writers in the c.s.
Mesa Condition Variables, cont’d

- When a thread that holds the lock notices that some waiting condition is satisfied it should *notify* the corresponding condition variable.
from synch import *

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

r_cond: used by readers to wait on nwriters == 0
w_cond: used by writers to wait on nreaders == 0 == nwriters
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)
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)
R/W Lock, reader part

```python
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)
```
R/W Lock, reader part

```python
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):
        if rw->nreaders == 0:
            notify(rw->w_cond)
        release(rw->mutex)
```

NEVER EVER USE WAIT WITHOUT AN ENCLOSING WHILE ("naked wait")
R/W Lock, reader part

compare with busy waiting

```python
def read_acquire(rw):
    acquire(rw→lock)
    while rw→nwriters > 0:
        release(rw→lock)
        acquire(rw→lock)
    rw→nreaders += 1
    release(rw→lock)

def read_release(rw):
    acquire(rw→lock)
    rw→nreaders -= 1
    if rw→nreaders == 0:
        notify(rw→w_cond)
    release(rw→mutex)
```
R/W Lock, reader part

compare with busy waiting

def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
        rw→nreaders += 1
    release(?rw→lock)

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

def read_acquire(rw):
    acquire(?rw→mutex)
    while rw→nwriters > 0:
        wait(?rw→r_cond, ?rw→mutex)
        rw→nreaders += 1
    release(?rw→mutex)
R/W Lock, reader part

compare with busy waiting

```python
def read_acquire(rw):
    acquire(?rw→lock)
    while rw→nwriters > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nreaders += 1
    release(?rw→lock)

def read_release(rw):
    acquire(?rw→lock)
    rw→nreaders -= 1
    if rw→nreaders == 0:
        notify(?rw→w_cond)
    release(?rw→lock)
```
R/W Lock, reader part

compare with busy waiting

```python
def read_acquire(rw):
    acquire(rw→lock)
    while rw→nwriters > 0:
        release(rw→lock)
        acquire(rw→lock)
    rw→nreaders += 1
    release(rw→lock)

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

```python
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)
```
R/W Lock, writer part

```python
23  def write_acquire(rw):
24      acquire(?rw→mutex)
25      while (rw→nreaders + rw→nwriters) > 0:
26          wait(?rw→w_cond, ?rw→mutex)
27          rw→nwriters = 1
28      release(?rw→mutex)
29
30  def write_release(rw):
31      acquire(?rw→mutex)
32      rw→nwriters = 0
33      notifyAll(?rw→r_cond)
34      notify(?rw→w_cond)
35      release(?rw→mutex)
```
R/W Lock, writer part

```
def write_acquire(rw):
    acquire(\(rw\rightarrow\text{mutex}\))
    while \(rw\rightarrow\text{nreaders} + rw\rightarrow\text{nwriters} > 0\):
        wait(\(rw\rightarrow\text{w\_cond}, rw\rightarrow\text{mutex}\))
    rw\rightarrow\text{nwriters} = 1
    release(\(rw\rightarrow\text{mutex}\))

    def write_release(rw):
        acquire(\(rw\rightarrow\text{mutex}\))
        rw\rightarrow\text{nwriters} = 0
        notifyAll(\(rw\rightarrow\text{r\_cond}\))
        notify(\(rw\rightarrow\text{w\_cond}\))
        release(\(rw\rightarrow\text{mutex}\))
```
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)
R/W Lock, writer part

```python
def write_acquire(rw):
    acquire(?rw→lock)
    while (rw→nreaders + rw→nwriters) > 0:
        release(?rw→lock)
        acquire(?rw→lock)
    rw→nwriters = 1
    release(?rw→lock)

def write_release(rw):
    acquire(?rw→lock)
    rw→nwriters = 0
    release(?rw→lock)
```

```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)
```
Why not use “if” instead of “while” around `wait()`?

- By the time waiter gets the lock back, condition may no longer hold
  - Given three threads, W1, R2, W3
  - W1 enters as a writer
  - R2 waits as a reader
  - W1 leaves, notifying R2
  - W3 enters as a writer
  - R2 wakes up
    - If R2 doesn’t check condition again, R2 and W3 would both be in the critical section
Why not use “if” instead of “while” around $wait()$?

- When notifying, be safe rather than sorry
- it’s better to notify too many threads than too few
- in case of doubt, use notifyAll() instead of just notify()
- But this too can lead to some threads waking up when their condition is no longer satisfied
Why not use “if” instead of “while” around `wait()`?

• Because you **should** use while around `wait`, many condition variable implementation allow “**spurious wakeups**”
• `wait()` resumes even although condition variable was not notified
Naked waits: just don’t do it
# Busy Waiting vs Condition Variables

<table>
<thead>
<tr>
<th>Busy Waiting</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use a lock and a loop</td>
<td>Use a lock <em>and a collection of condition variables</em> and a loop</td>
</tr>
<tr>
<td>Easy to write the code</td>
<td>Notifying is tricky</td>
</tr>
<tr>
<td>Easy to understand the code</td>
<td>Easy to understand the code</td>
</tr>
<tr>
<td>Progress property is easy</td>
<td>Progress requires careful consideration (both for correctness and efficiency)</td>
</tr>
<tr>
<td>Ok-ish for true multi-core, but bad for virtual threads</td>
<td>Good for both multi-core and virtual threading</td>
</tr>
</tbody>
</table>
Mesa Monitors in Harmony

```python
def Condition():
    result = 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 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 list of suspended threads