Conditional Waiting

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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
  - …

Reader/Writer Lock Specification

```
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
```

Reader/Writer Lock Specification

```
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
```

Invariants:

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

R/W Locks: test for mutual exclusion

```
import rwlock

nreaders = nwriters = 0
invariant ((nreaders >= 0) and (nwriters == 0)) or
          ((nreaders == 0) and (0 <= nwriters <= 1))

const NOPS = 3

rw = rwlock.RWlock()

def thread():
    while choose({ False, True }):
        if choose({ "read", "write" }) == "read":
            rwlock.read_acquire(?rw)
            atomically nreaders += 1
            atomically nreaders -= 1
            rwlock.read_release(?rw)
        else:                          # write
            rwlock.write_acquire(?rw)
            atomically nwriters += 1
            atomically nwriters -= 1
            rwlock.write_release(?rw)

for i in {1..NOPS}:
    spawn thread()
```

no writer, one or more readers

one writer, no readers

Cheating R/W lock implementation

```
import synch

def RWlock() returns lock:
    lock = 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

```
import synch

def RWlock() returns lock:
    lock = 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

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

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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

Good: has the same behaviors as the implemention

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

R/W lock with *Mesa condition variables*

```
from synch import *

def RWlock() returns lock:
    lock = {
              .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*

R/W Lock, reader part

```
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

```
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

R/W Lock, reader part

```
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 need this

R/W Lock, reader part

```
def read_acquire(rw):
    acquire(?rw->mutex)
    while rw->nwriters > 0:
        wait(?rw->r_cond, ?rw->mutex)
    rw->nreade        1
    releas

def read_r
    acquir
    rw->nr
    if rw->nreaders == 0:
        notify(?rw->w_cond)
    release(?rw->mutex)
```

similar to busy waiting

but need this

- Always use **while**
- Never just **if** (or nothing)
- **wait** without **while** is called a “naked wait”

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→lock)
    rw→nreaders −= 1
    release(?rw→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)
```

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→lock)
    rw→nreaders −= 1
    release(?rw→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)
```

R/W Lock, writer part

```
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)
```

don't forget anybody!

Condition Variable interface

- **wait**(*cv, lock*)
  - may only be called while holding *lock*
  - temporarily releases *lock*
    - but re-acquires it before resuming
  - if *cv* not notified, may block indefinitely
    - but wait() may resume ”on its own”
- **notify**(*cv*)
  - no-op if nobody is waiting on *cv*
  - otherwise wakes up at least one thread waiting on *cv*
- **notifyAll**(*cv*)
  - wakes up all threads currently waiting on *cv*

Busy Waiting or?

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomically !lk = False
```

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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)
```

Busy Waiting or?

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomically !lk = False
```

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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)
```

Busy Waiting or?

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomically !lk = False
```

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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)
```

State unchanged while condition does not hold. This thread only "observes" the state until condition holds

State conditionally changes while condition does not hold. This thread actively changes the state until the condition hold

Busy Waiting or?

```
def test_and_set(s) returns result:
    atomically:
        result = !s
        !s = True

def Lock() returns result:
    result = False

def acquire(lk):
    while test_and_set(lk):
        pass

def release(lk):
    atomically !lk = False
```

```
from synch import Lock, acquire, release

def RWlock() returns lock:
    lock = { .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)
```

Busy Waiting

State unchanged while condition does not hold. This thread only "observes" the state until condition holds

State conditionally changes while condition does not hold. This thread actively changes the state until the condition hold

Why is busy waiting bad?

- Consider a timesharing setting
- Threads T1 and T2 take turns on the CPU
  - switch every 100 milliseconds
- Suppose T1 has the lock and is running
- Now suppose a clock interrupt occurs, T2 starts running and tries to acquire the lock
- Non-busy-waiting acquisition:
  - T2 is put on a waiting queue and T1 resumes and runs until T1 releases the lock (which puts T2 back on the run queue)
- Busy-waiting acquisition:
  - T2 keeps running (wasting CPU) until the lock is available until the next clock interrupt
  - T1 and T2 switch back and forth until T1 releases the lock

Busy Waiting vs Condition Variables

Busy Waiting	Condition Variables
Use a lock and a loop	Use a lock *and a collection of condition variables* and a loop
Easy to write the code	Notifying is tricky
Easy to understand the code	Easy to understand the code
Progress property is easy	Progress requires careful consideration (both for correctness and efficiency)
Ok-ish for true multi-core, but bad for virtual threads	Good for both multi-core and virtual threading

Busy Waiting: just don’t do it

Why no naked waits? (reason 1)

- 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 no naked waits? (reason 2)

- 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 no naked waits? (reason 3)

- Because you should use **while** around **wait**, many condition variable implementations allow "spurious wakeups"
  - wait() resumes even though condition variable was not notified

Naked waits: just don’t do it

Hints for reducing unneeded wakeups

- Use separate condition variables for each waiting condition
- Don’t use **notifyAll** when **notify** suffices
  - but be safe rather than sorry
  - sometimes you can even use N calls to **notify** if you know at most N nodes can continue after a waiting condition holds