[537] Locks and Condition Variables

Tyler Harter
10/13/14

P2B recap

Remember non-determinism makes testing harder!

Build incrementally.

Separate policy from mechanism.

Bad

```
scheduler() {
  …
  lots of logic for choosing process    policy
  …
  proc = p;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&cpu->scheduler, proc->context);    mechanism
  switchkvm();
  …
}
```

Worse

```
scheduler() {
  …
  lots of logic for choosing process
  …
  proc = p;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&cpu->scheduler, proc->context);
  switchkvm();
  …
}
```

copy this many places…

```
struct proc *choose_proc() {
  lots of logic for choosing process
}

scheduler() {
  …
  p = choose_proc();
  if (p) {
    proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&cpu->scheduler, proc->context);
    switchkvm();
  }
  …
}
```

Better: separation
of policy/mechanism

Review: using and designing basic locks

Problem 1

Do it.

Lock Evaluation

How to tell if a lock implementation is good?

Lock Evaluation

How to tell if a lock implementation is good?

Fairness: does everybody get a chance to use the lock?

Performance
- high contention (many threads per CPU, each contending)
- low contention (fewer threads, fewer locking attempts)

Lock Evaluation

How to tell if a lock implementation is good?

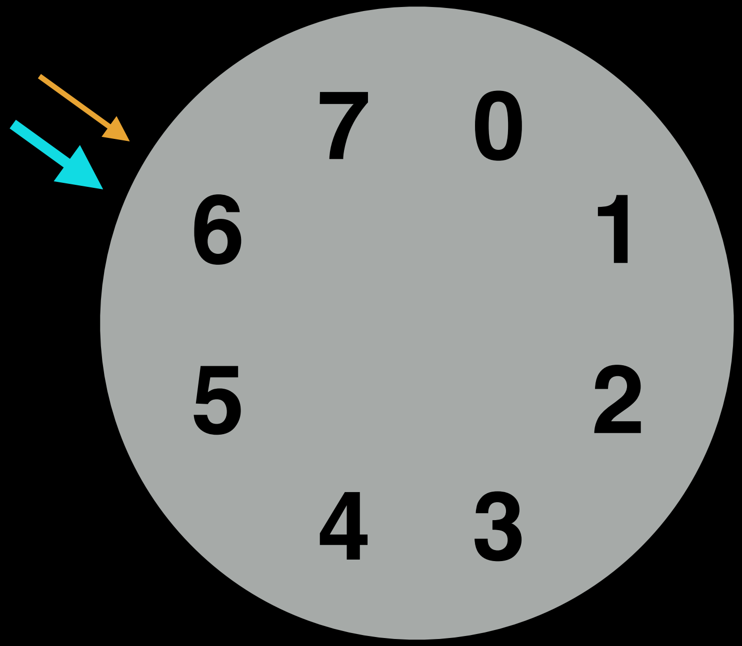
Fairness: does everybody get a chance to use the lock?

Performance

- high contention (many threads per CPU, each contending)
- low contention (fewer threads, fewer locking attempts)

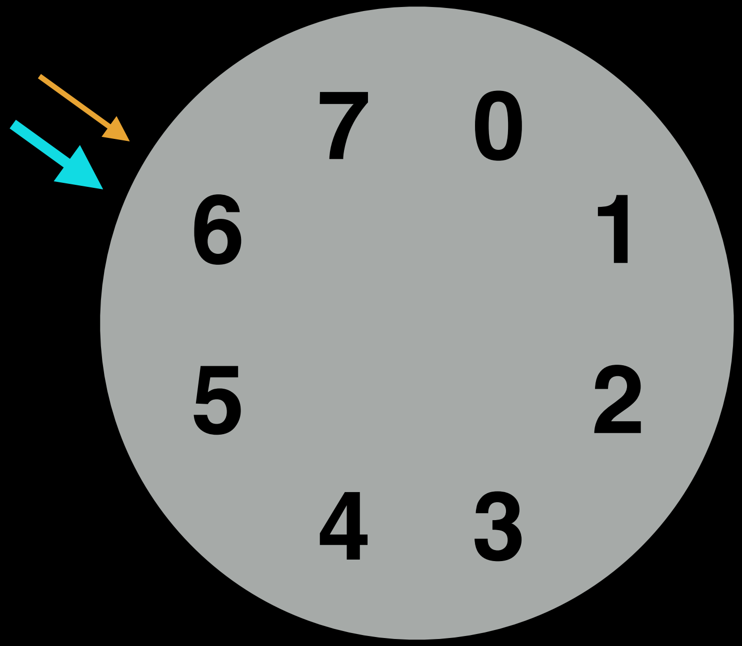
which are spinlocks better for?

Ticket Lock Review



turn = 6

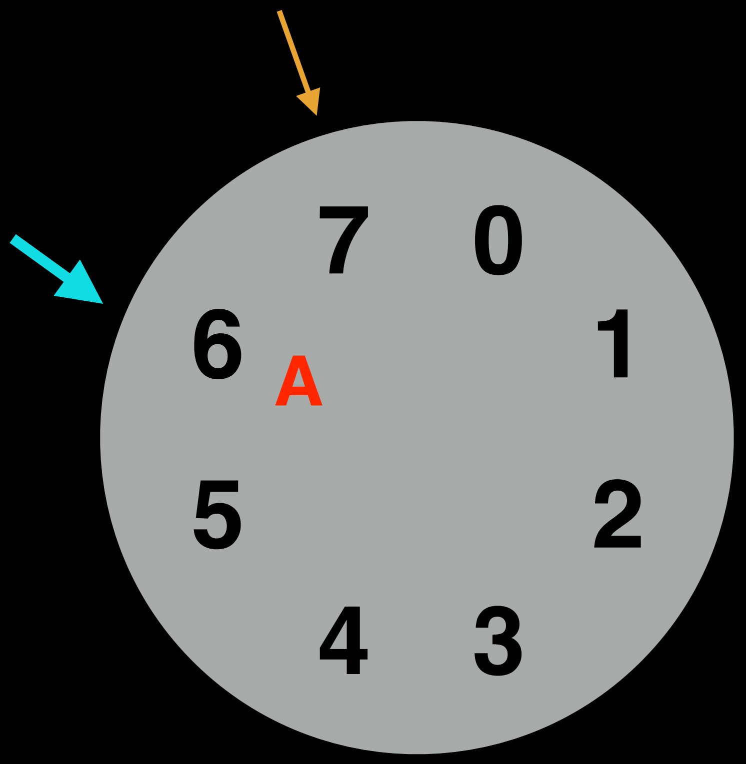
ticket = 6



turn = 6

ticket = 6

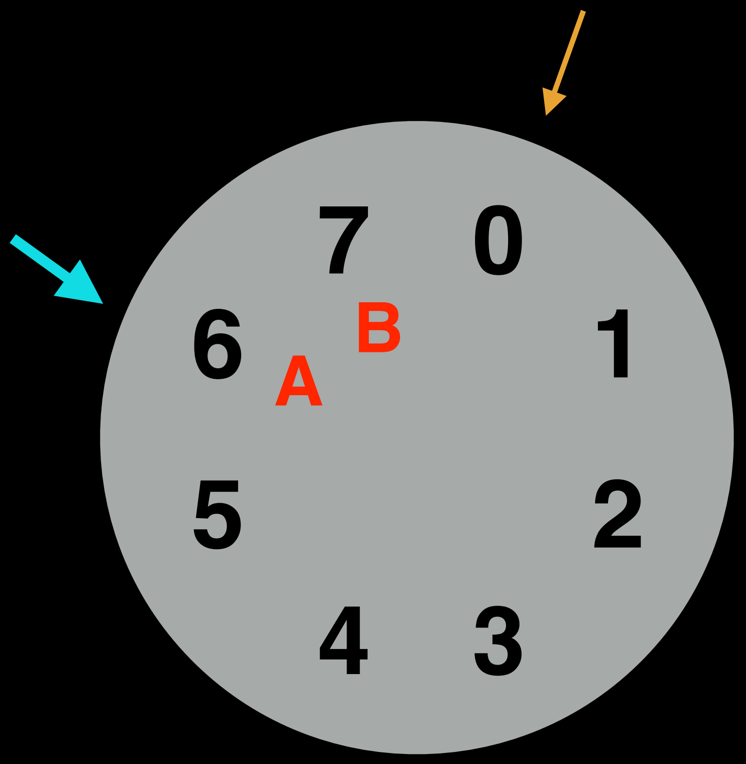
A lock(): gets ticket 6, runs



turn = 6

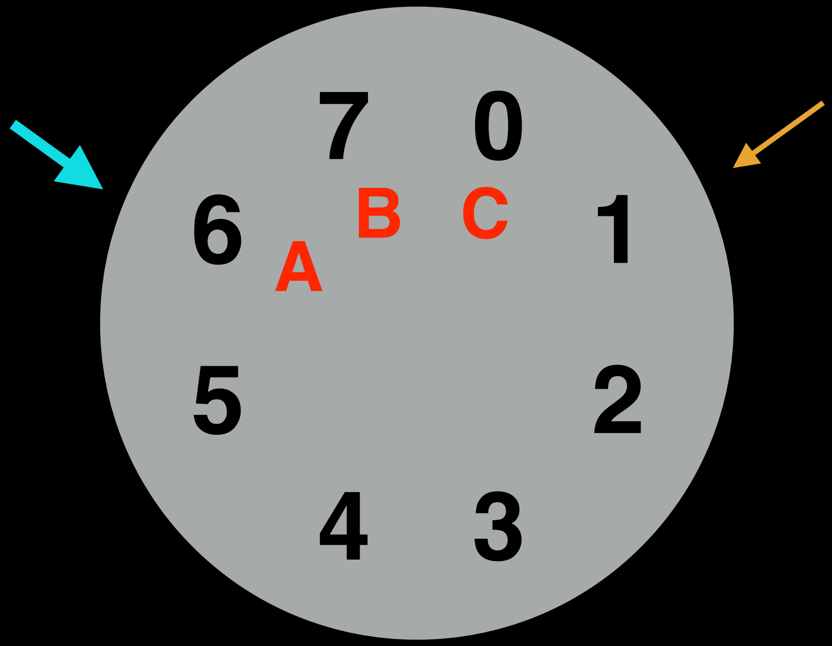
ticket = 7

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7



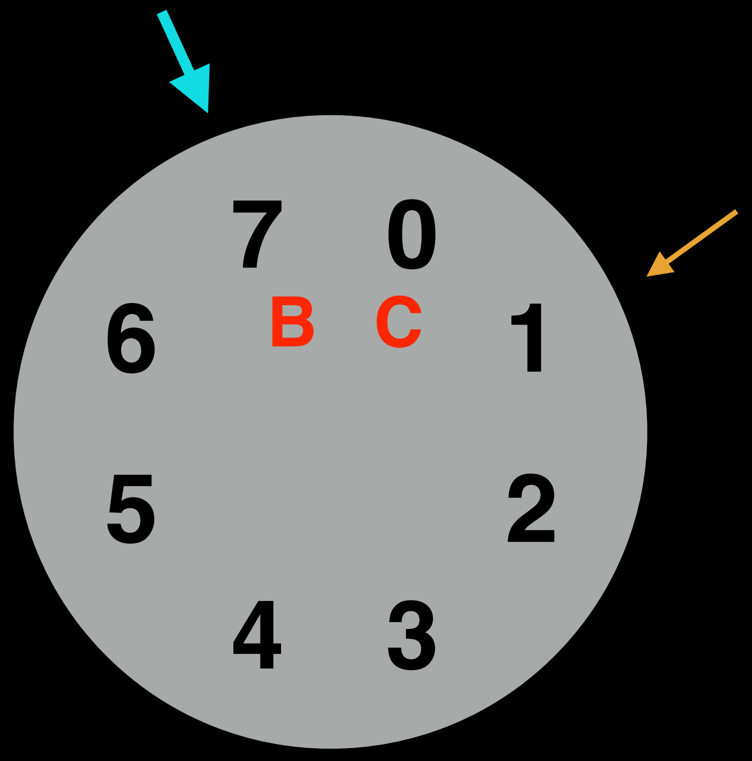
turn = 6
ticket = 0

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0



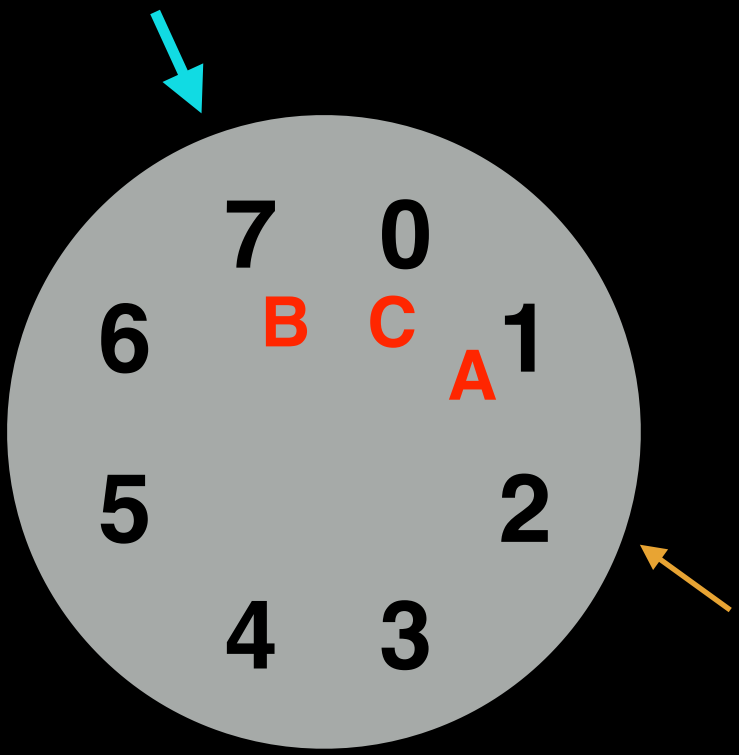
turn = 6
ticket = 1

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs



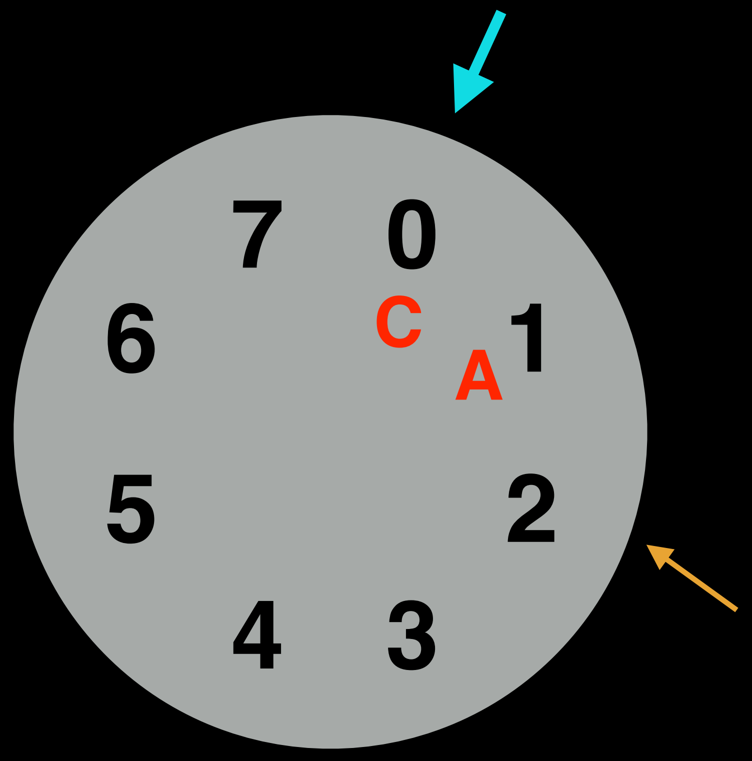
turn = 7
ticket = 1

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1



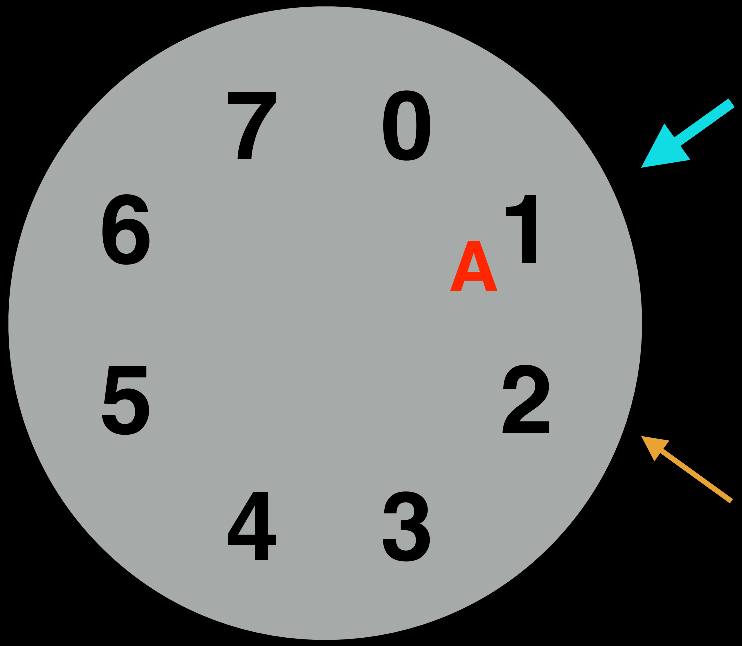
turn = 7
ticket = 2

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs



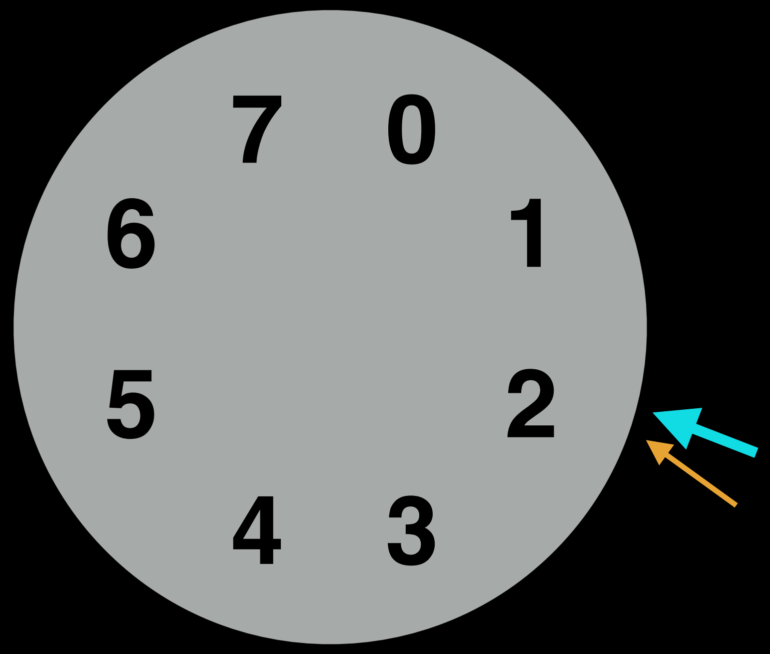
turn = 0
ticket = 2

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs



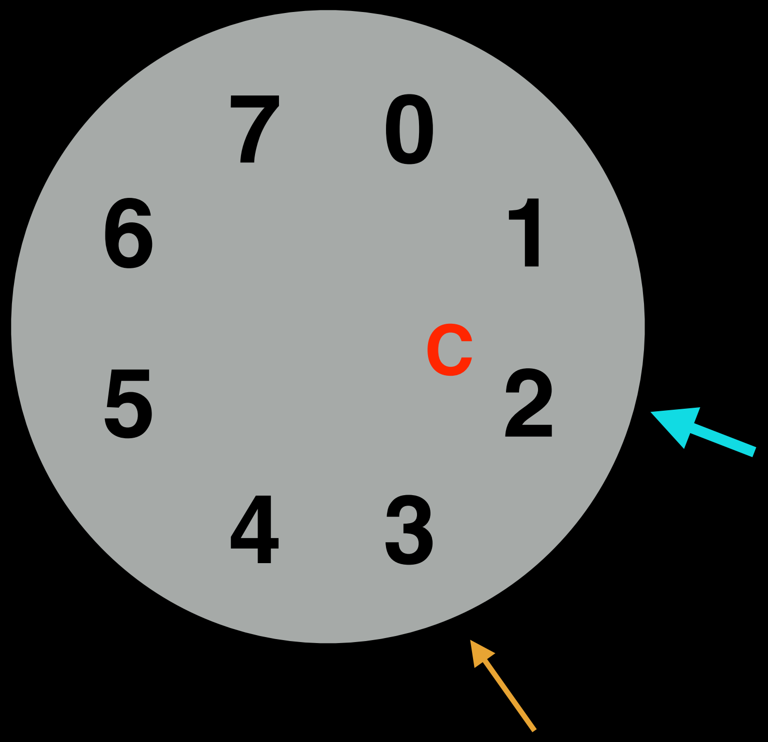
turn = 1
ticket = 2

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++



turn = 2
ticket = 2

A lock(): gets ticket 6, runs
B lock(): gets ticket 7, spins until turn=7
C lock(): gets ticket 0, spins until turn=0
A unlock(): turn++
B runs
A lock(): gets ticket 1, spins until turn=1
B unlock(): turn++
C runs
C unlock(): turn++
A runs
A unlock(): turn++
C lock(): gets ticket 2, runs



turn = 2
ticket = 3

Problem 2

Do it.

Ticket Lock (1)

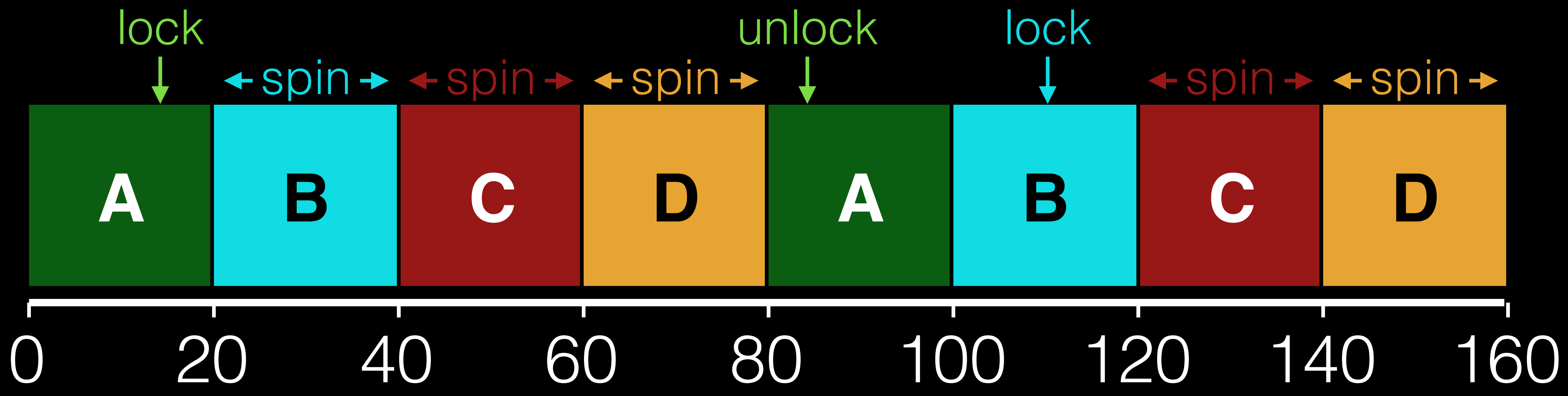
```
typedef struct __lock_t {
    int ticket;
    int turn;
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

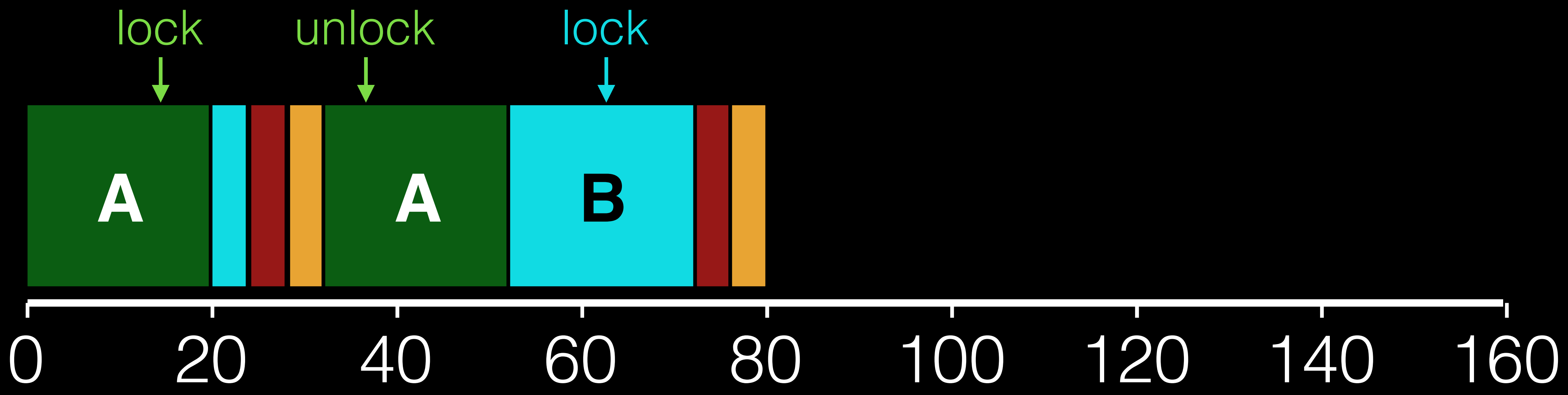
```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while(lock->turn != myturn)
        ; // spin
}

void release (lock_t *lock) {
    FAA(&lock->turn);
}
```

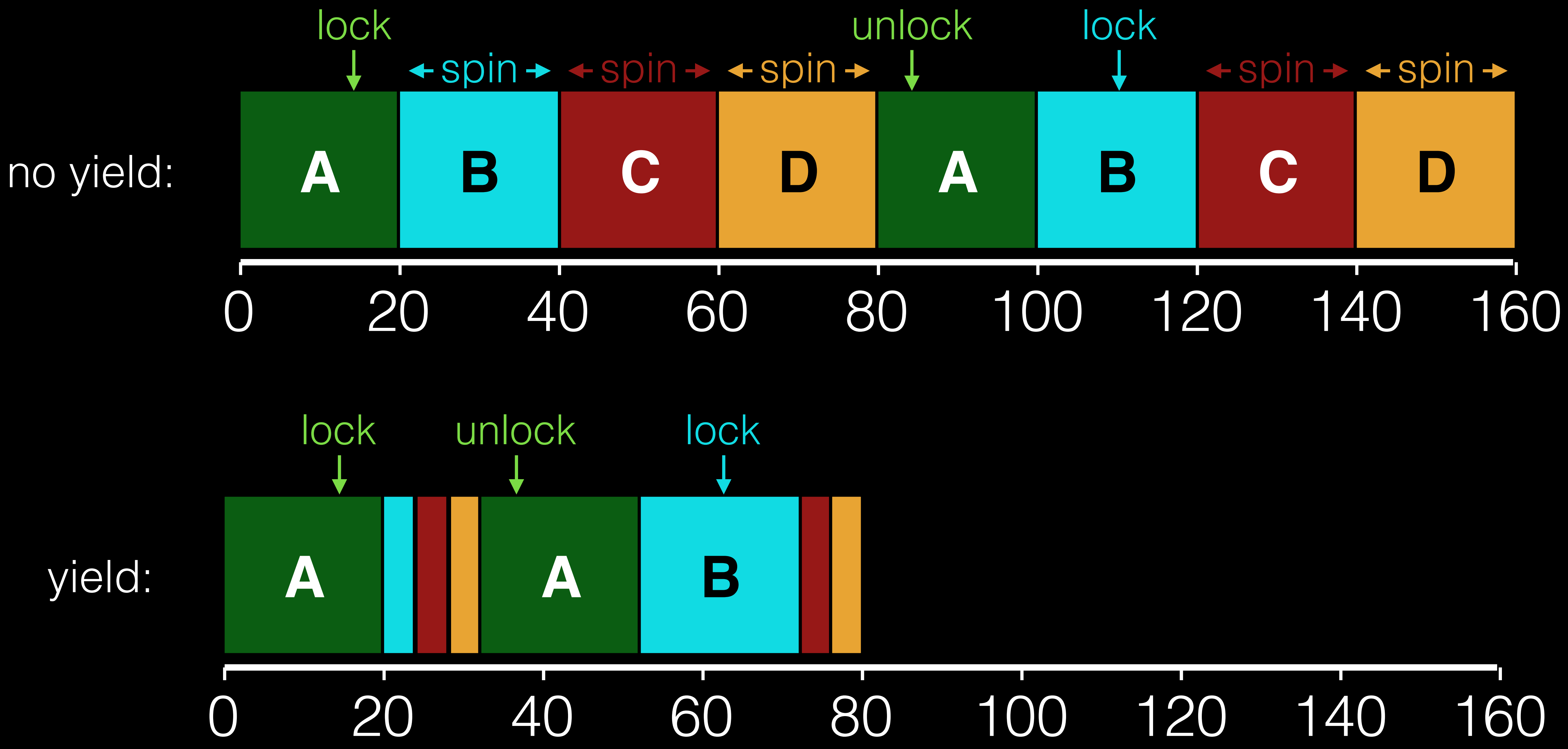
Ticket Lock (2)

```
typedef struct __lock_t {
    int ticket;
    int turn;
}

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    while(lock->turn != myturn)
        yield(); // spin
}

void release (lock_t *lock) {
    FAA(&lock->turn);
}
```


lock
spin
spin
spin
unlock
lock
spin
spin
A
B
C
D
A
B
C
D
0
20
40
60
80
100
120
140
160


lock
unlock
lock
A
A
B
0
20
40
60
80
100
120
140
160


lock
spin
spin
spin
unlock
lock
spin
spin
no yield:
A
B
C
D
A
B
C
D
0
20
40
60
80
100
120
140
160
lock
unlock
lock
yield:
A
A
B
0
20
40
60
80
100
120
140
160

Spinlock Performance

Waste…

Without yield: O(threads * context_switch)

With yield: O(threads * time_slice)

Spinlock Performance

Waste…

Without yield: O(threads * time_slice)

With yield: O(threads * context_switch)

So even with yield, we’re slow with high contention.

Problem 3

Do it.

Problem 3

Do it.

(a) This spins on guard — why? (what is protected? what is not?)
(b) This still spins. Why is it better than a simple spin lock?
(c) In unlock, there is no setting of flag=0 when we unpark. Why?
(d) What is the race-condition bug in this code?

Race Condition

Thread 1	**Thread 2**
`if (lock->flag == 0)`	
`queue_push(lock->q, gettid());`	
`lock->guard = 0;`	
	`while (xchg(&lock->guard, 1) == 1)`
	`if (queue_empty(lock->q))`
	`unpark(queue_pop(lock->q));`
	`lock->guard = 0;`
`park();`	
(in lock)	(in unlock)

Incorrect Code

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {               // lock not free: sleep
            queue_push(lock->q, gettid());
            lock->guard = 0;
            park();           // put self to sleep
    }
}
```

Incorrect Code

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {                 // lock not free: sleep
            queue_push(lock->q, gettid());
            lock->guard = 0;
            park();            // put self to sleep
    }
}
```

Race!

Incorrect Code

```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {                // lock not free: sleep
            queue_push(lock->q, gettid());
            lock->guard = 0;
            park();           // put self to sleep
    }
}
```

Correct Code

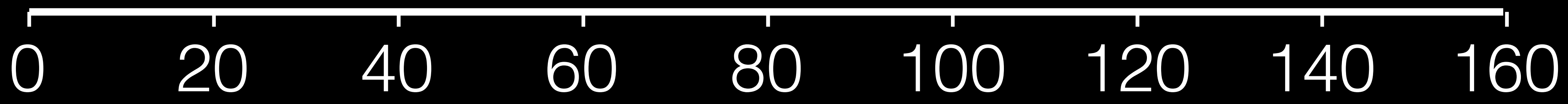
```
void lock(lock_t *lock) {
    while (xchg(&lock->guard, 1) == 1)
            ; // spin
    if (lock->flag == 0) { // lock is free: grab it!
            lock->flag = 1;
            lock->guard = 0;
    } else {              // lock not free: sleep
            queue_push(lock->q, gettid());
            setpark();
            lock->guard = 0;
            park();          // put self to sleep
    }
}
```

Queue Lock

RUNNABLE: A, B, C, D

RUNNING: <empty>

WAITING: <empty>

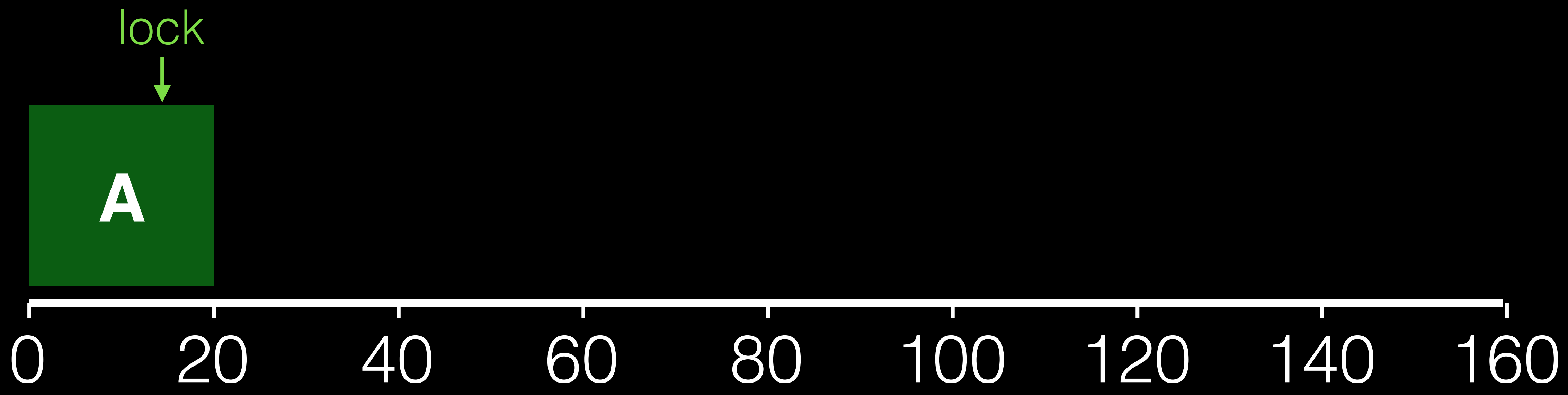


Queue Lock

RUNNABLE: B, C, D

RUNNING: A

WAITING: <empty>

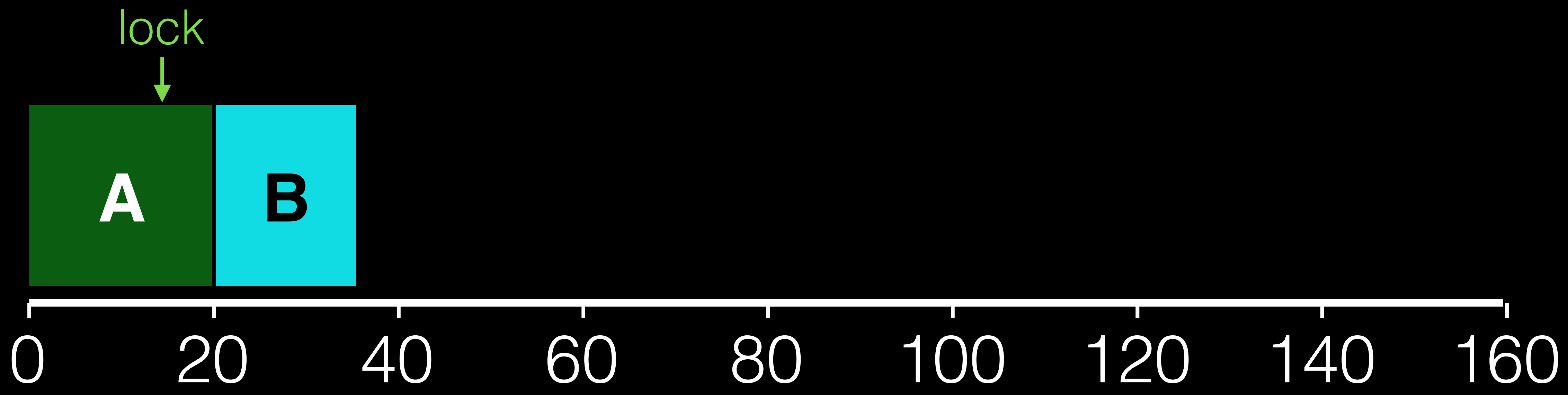


Queue Lock

RUNNABLE: C, D, A

RUNNING: B

WAITING: <empty>

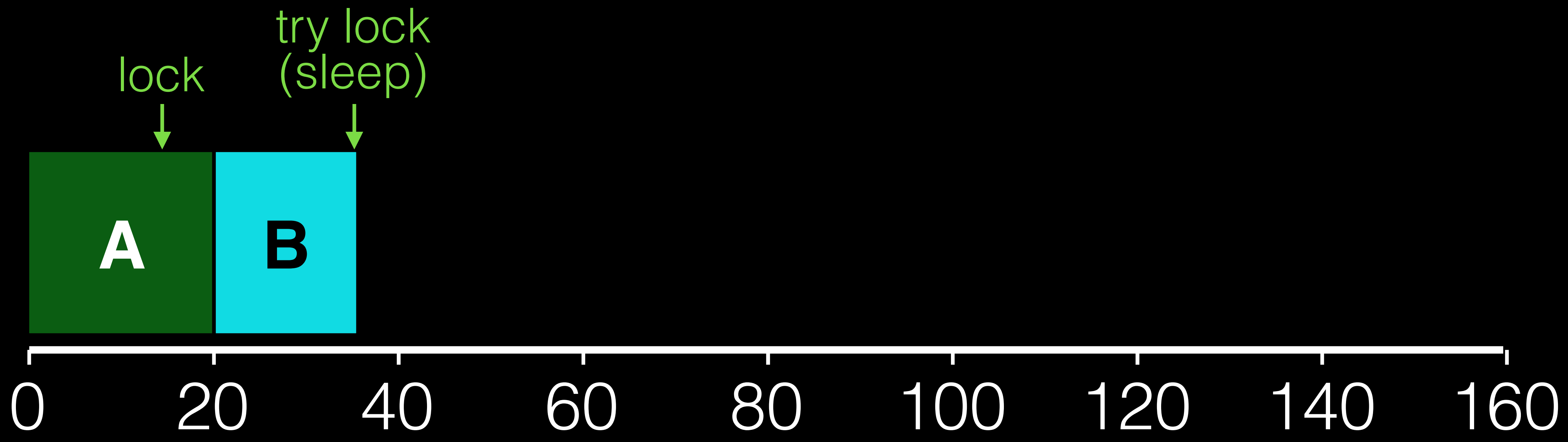


Queue Lock

RUNNABLE: C, D, A

RUNNING:

WAITING: B

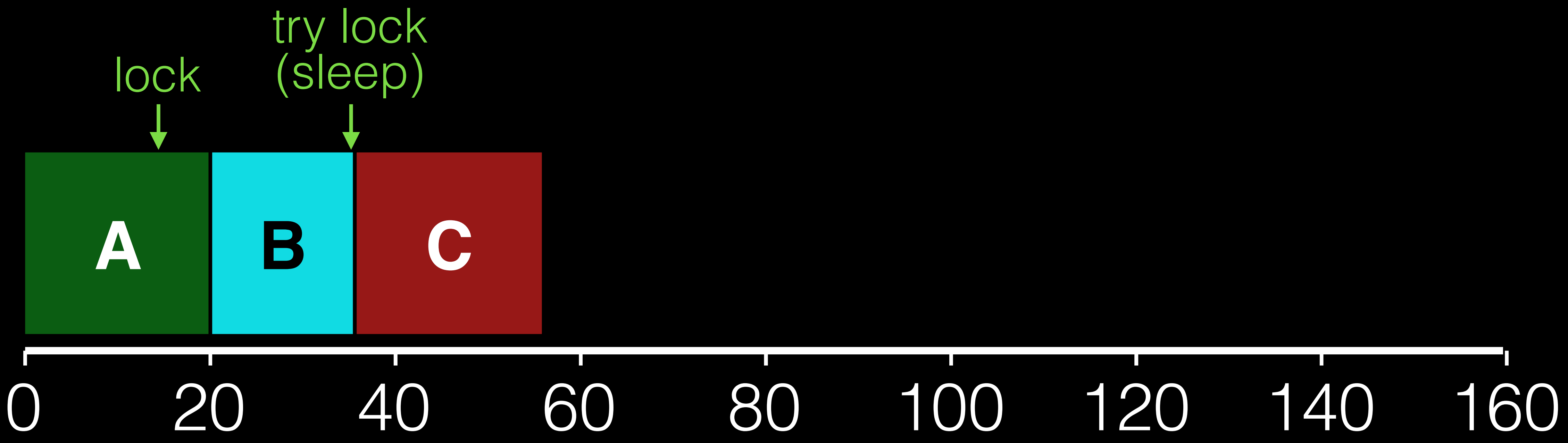


Queue Lock

RUNNABLE: D, A

RUNNING: C

WAITING: B

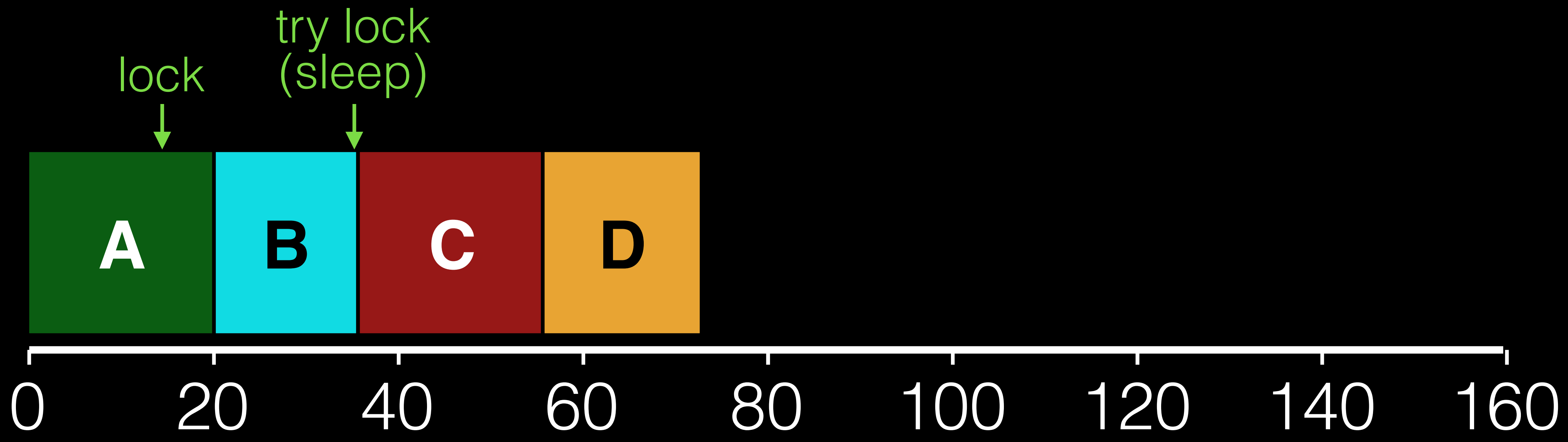


Queue Lock

RUNNABLE: A, C

RUNNING: D

WAITING: B

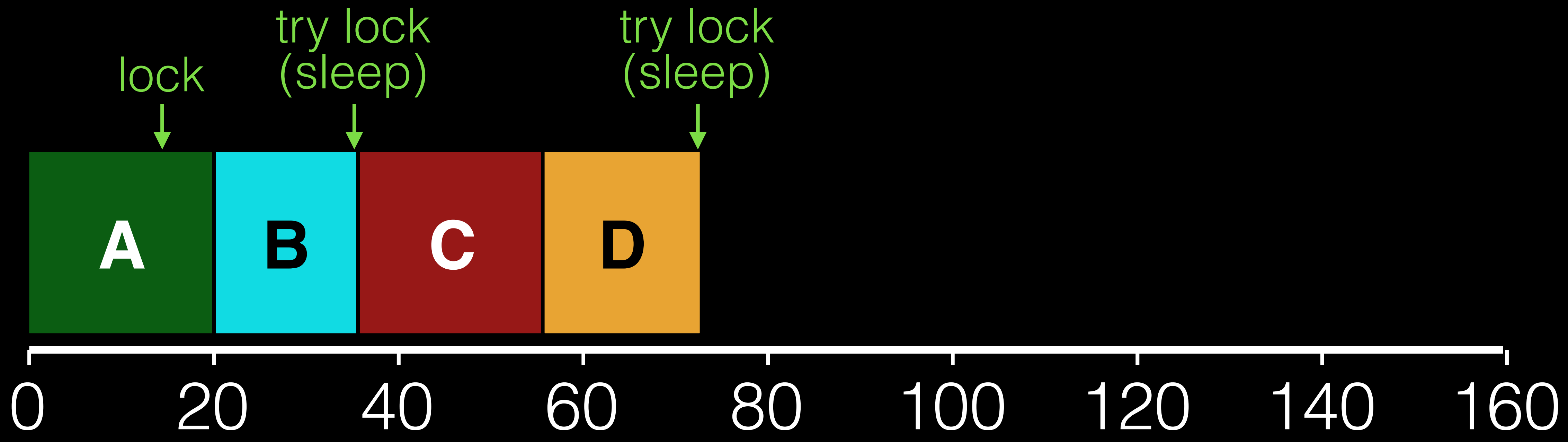


Queue Lock

RUNNABLE: A, C

RUNNING:

WAITING: B, D

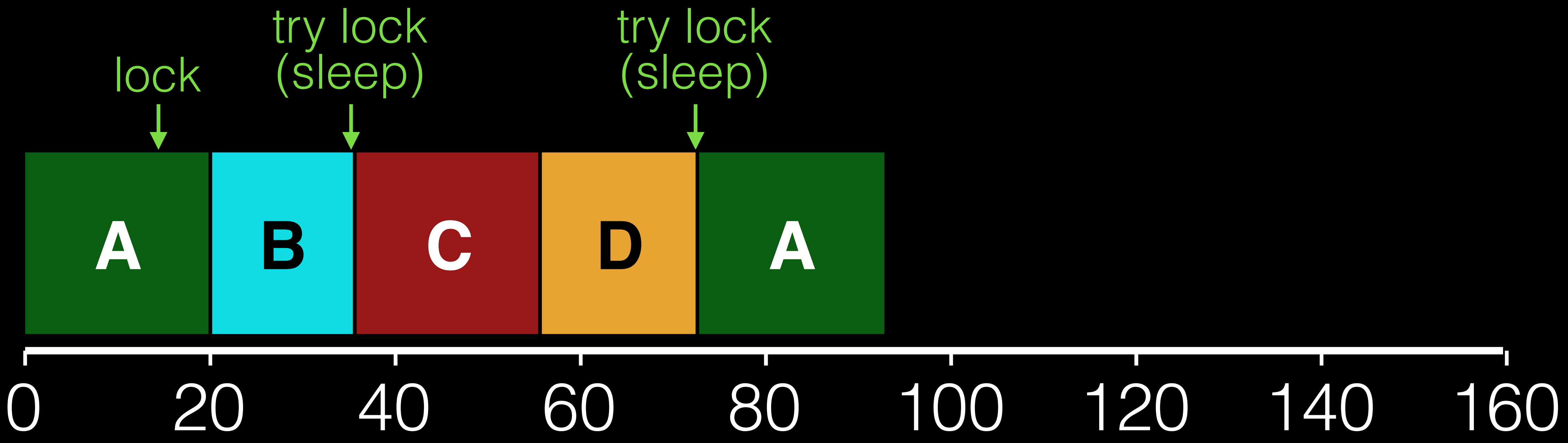


Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D

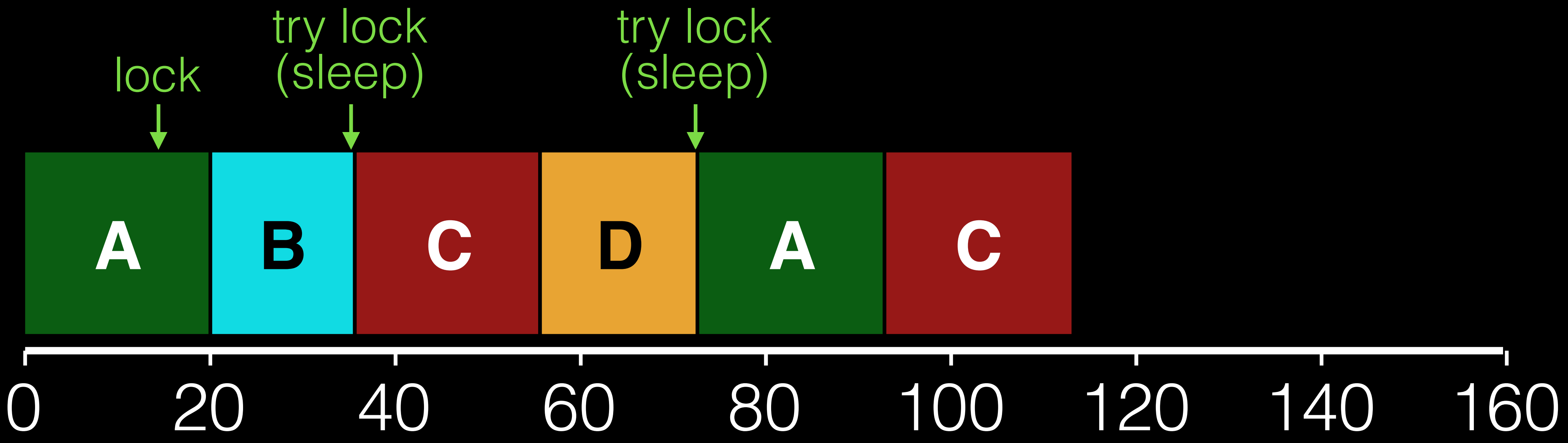


Queue Lock

RUNNABLE: A

RUNNING: C

WAITING: B, D

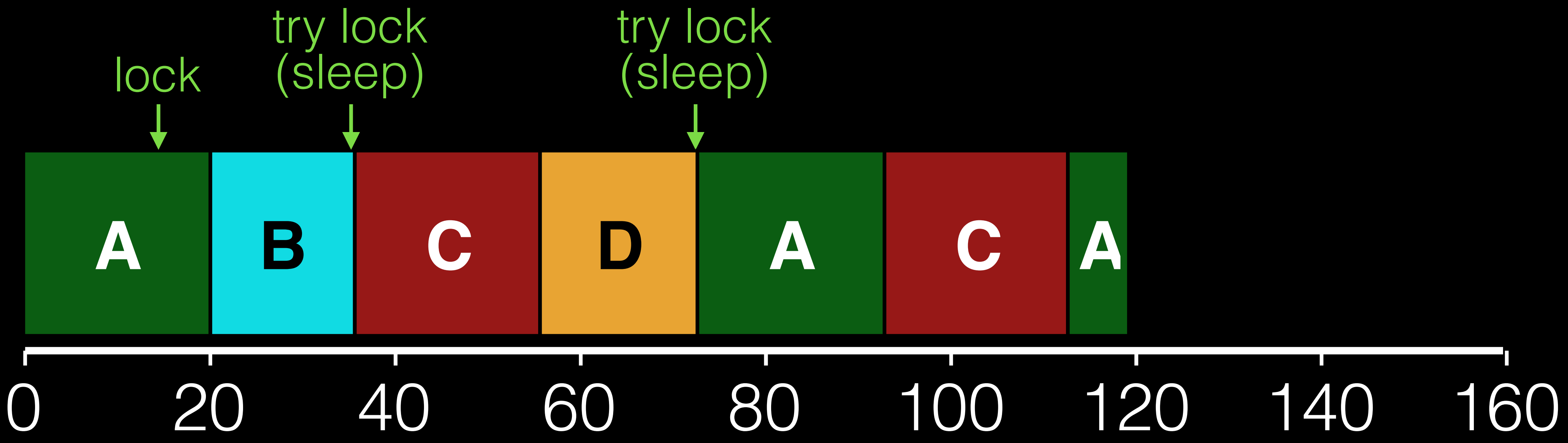


Queue Lock

RUNNABLE: C

RUNNING: A

WAITING: B, D

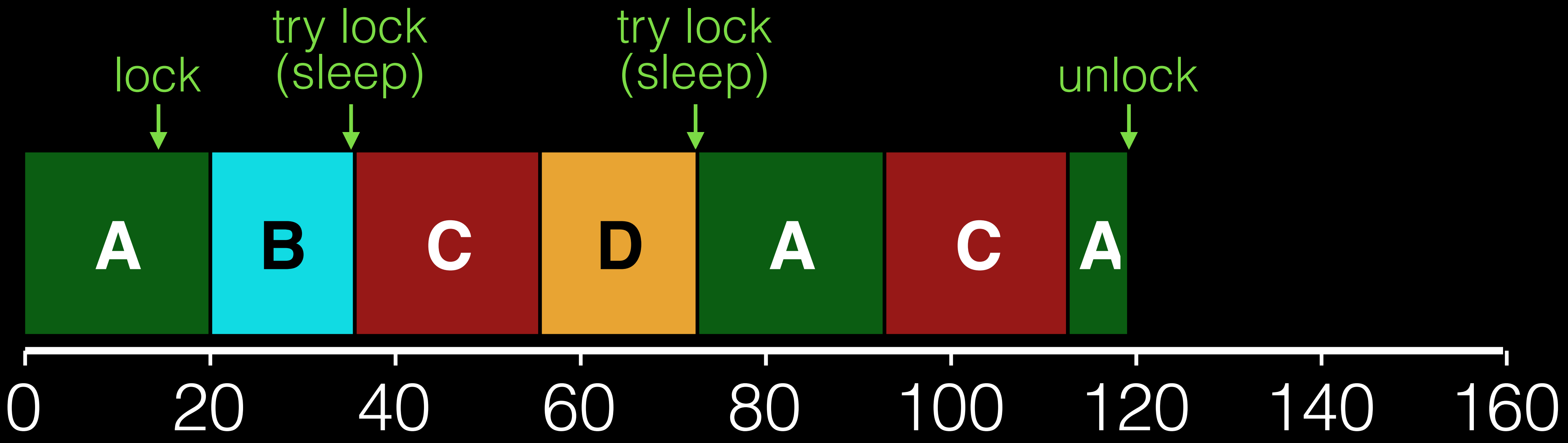


Queue Lock

RUNNABLE: B, D, C

RUNNING: A

WAITING:

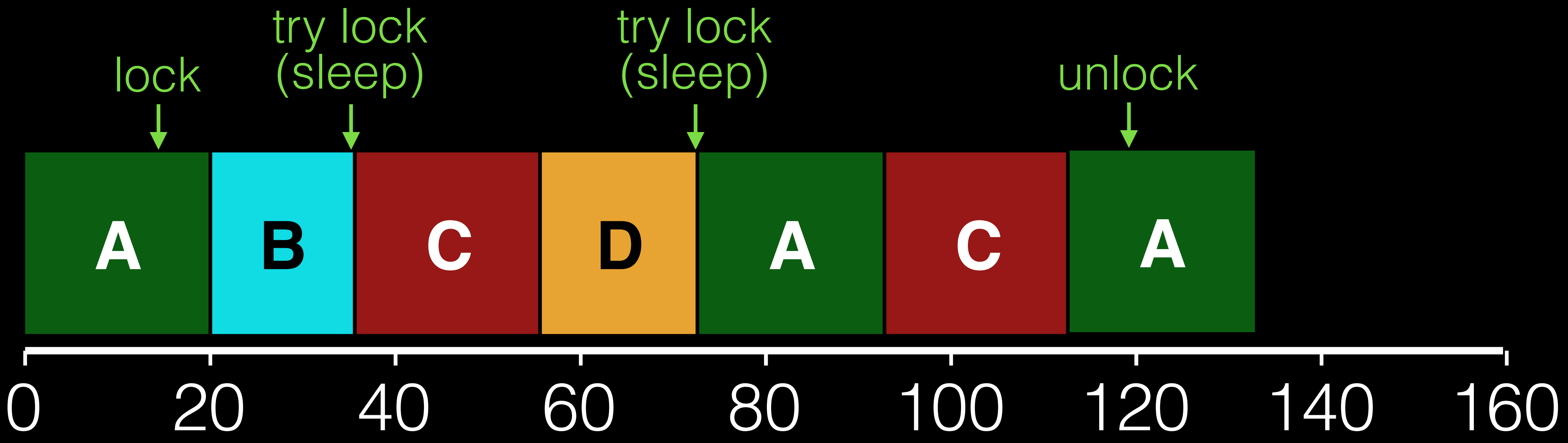


Queue Lock

RUNNABLE: B, D, C

RUNNING: A

WAITING:

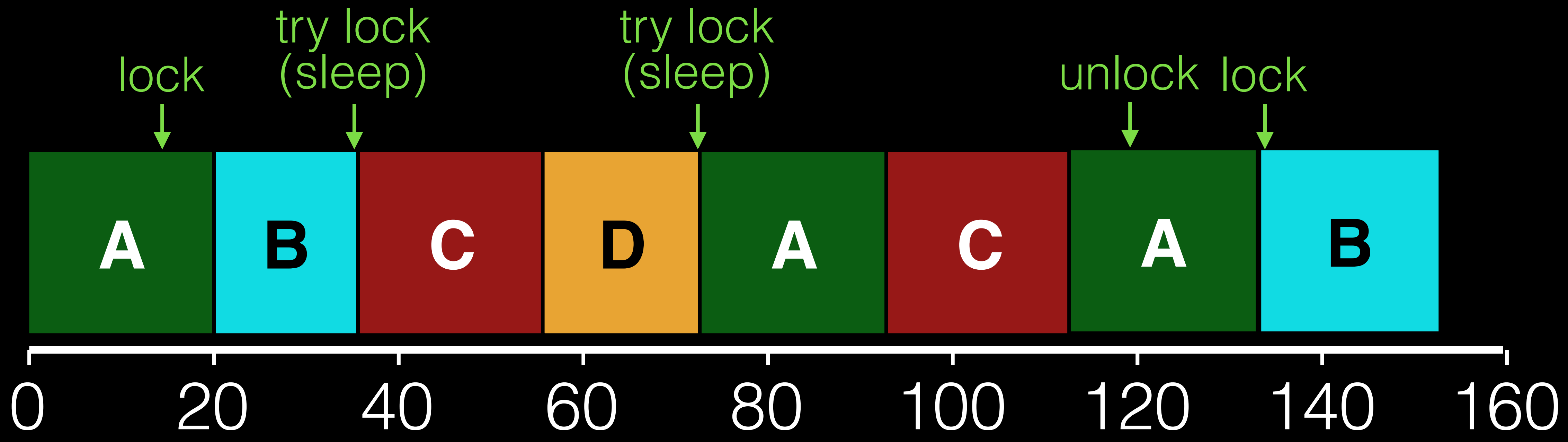


Queue Lock

RUNNABLE: D, C, A

RUNNING: B

WAITING:



Condition Variables

Concurrency Objectives

Mutual exclusion (e.g., A and B don't run at same time)
 - solved with *locks*

Ordering (e.g., B runs after A)
 - solved with *condition variables*

Ordering Example: Join

```
pthread_t p1, p2;
printf("main: begin [balance = %d]\n", balance);
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
   balance, max*2);
return 0;
```

Ordering Example: Join

```
pthread_t p1, p2;
printf("main: begin [balance = %d]\n", balance);
Pthread_create(&p1, NULL, mythread, "A");
Pthread_create(&p2, NULL, mythread, "B");
// join waits for the threads to finish
Pthread_join(p1, NULL);
Pthread_join(p2, NULL);
printf("main: done\n [balance: %d]\n [should: %d]\n",
   balance, max*2);
return 0;
```

how to implement join?

Condition Variables

CV’s are more like channels than variables.

B waits for a signal on channel before running.

A sends signal when it is time for ***B*** to run.

Condition Variables

CV's are more like channels than variables.
B waits for a signal on channel before running.
A sends signal when it is time for ***B*** to run.

A CV also has a queue of waiting threads.

Broken CV’s

wait(cond_t *cv)
- puts caller to sleep (and on queue)

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

Broken CV's

wait(cond_t *cv)
- puts caller to sleep (and on queue)

when to call?

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

Way 1

```
if (!ready)
    wait(&cv);

lock(&mutex);
// critical section
unlock(&mutex);
```

Way 1

```
if (!ready)
    wait(&cv);

lock(&mutex);
// critical section
unlock(&mutex);
```

what if another thread sets ready=1 here?

Way 2

```
lock(&mutex);
// critical section
if (!ready)
    wait(&cv);
unlock(&mutex);
```

Way 2

```
lock(&mutex);
// critical section
if (!ready)
    wait(&cv);
unlock(&mutex);
```

nobody can wake us up because we hold mutex

Broken CV’s

wait(cond_t *cv)
- puts caller to sleep

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

Correct CV's

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

Correct CV's

requires kernel
support!

wait(cond_t *cv, mutex_t *lock)
- assumes the lock is held when wait() is called
- puts caller to sleep + releases the lock (atomically)
- when awoken, reacquires lock before returning

signal(cond_t *cv)
- wake a single waiting thread (if >= 1 thread is waiting)
- if there is no waiting thread, just return w/o doing anything

Code Examples

NOTE: Handout reference defines CV’s.

CV’s in xv6 code

proc.c:
- sleep() is like cond_wait()
- wakeup() is like cond_signal()

Example use case:
- piperead() and pipewrite() in pipe.c

Announcements

Exam this Friday.
- Oct 17, 7-9pm, in CHEM 1351.
- Covers all material until that day.
- Read OSTEP!

Review this Wednesday.
- Oct 15, 7-9pm, room TBD, come with questions!

p3a posted.

Office hours after class in **Galapagos lab**.