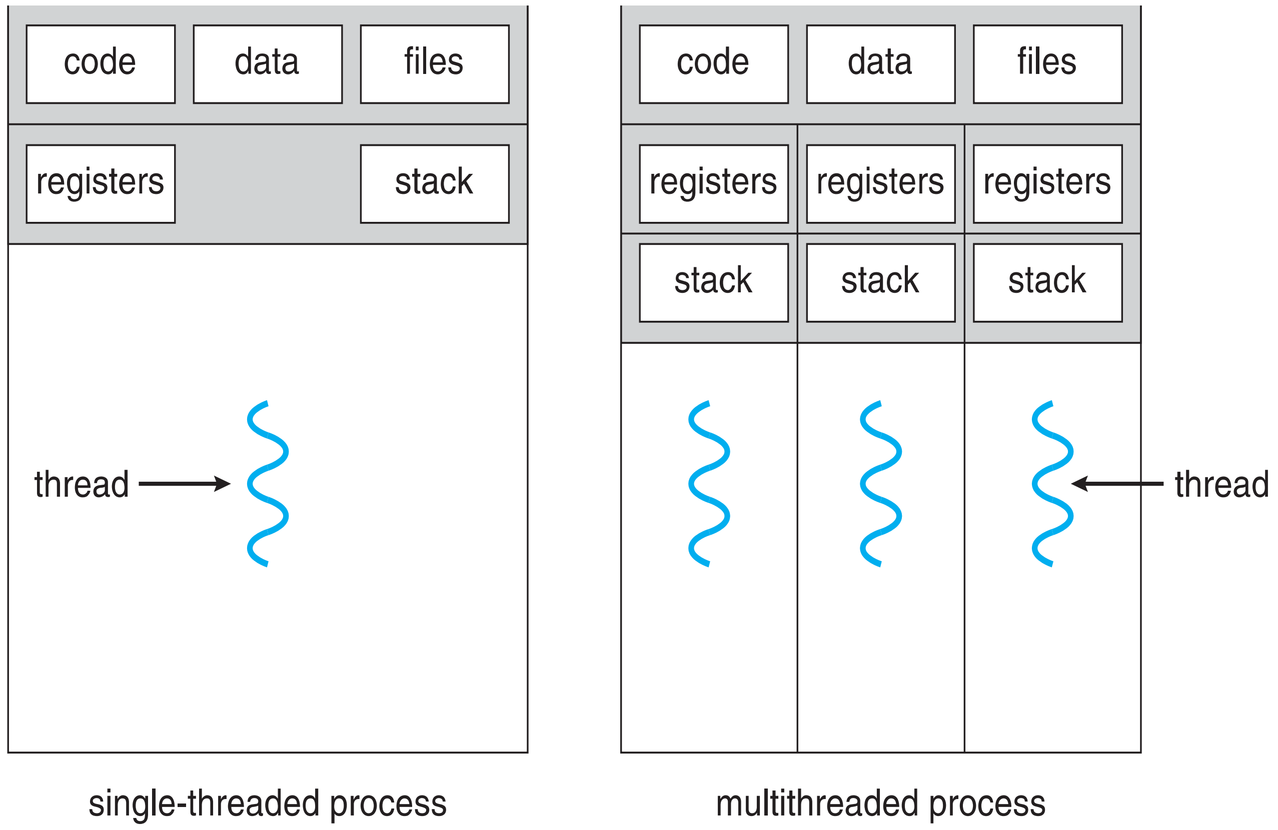
Concurrency: Locks

Sridhar Alagar

What do threads share?



Concurrency Issues

- Concurrent threads access shared variable/memory
- While one thread is accessing the shared memory another thread accessing the shared memory
  - Due to context switching, or
  - Multi-processor environments
- Due to this race condition arises and results in non-deterministic and incorrect execution

Handling Critical Section

counter = counter + 1; counter is a shared variable

- Instructions updating shared memory must execute as an uninterruptable group

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

critical section

- Need mutual exclusion for critical sections
- if process A is in critical section, process B can't be in CS

Criteria for a Solution to CS Problem

- Mutual Exclusion
  - only one thread in CS at a given time
- Fairness
  - Does each thread waiting to enter CS get a fair shot at entering CS? (Starvation free)
- Performance
  - The time overhead added by the solution

Locks

- Use locks to ensure mutually exclusion for CS

```
lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks Semantics

- A variable holds the state of the lock
- available/free: no thread has acquired the lock
- acquired/locked: exactly one thread has acquired the lock
- lock()
  - Acquire the lock if it is free

Pthread Locks

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // acquire the lock
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

How to implement Locks?

Control Interrupts

- Disable interrupts before entering CS and enable after exiting CS
  - one of the earliest solution

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Problems with disabling interrupts

- Application can monopolize the processor
- Does not work on multi-processors
- Interrupts can be lost
- Masking/unmasking very slow

A software solution - use load/store

```c
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
  // 0 → lock is available, 1 → held
  mutex->flag = 0;
}
void lock(lock_t *mutex) {
  while (mutex->flag == 1)   // TEST the flag
      ;  // spin-wait (do nothing)
  mutex->flag = 1;  // now SET it !
}
void unlock(lock_t *mutex) {
  mutex->flag = 0;
}
```

Problems with software solution

- No mutual exclusion

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

- Spin-waiting wastes CPU cycles

Hardware is your friend

- Need a hardware supported atomic instruction for test and set

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;// fetch old value at ptr
    *ptr = new;     // store 'new' into ptr
    return old;     // return the old value
}
```

Spin Lock using test-and-set

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
          ;     // spin-wait
}
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Compare and Swap (H/W-atomic)

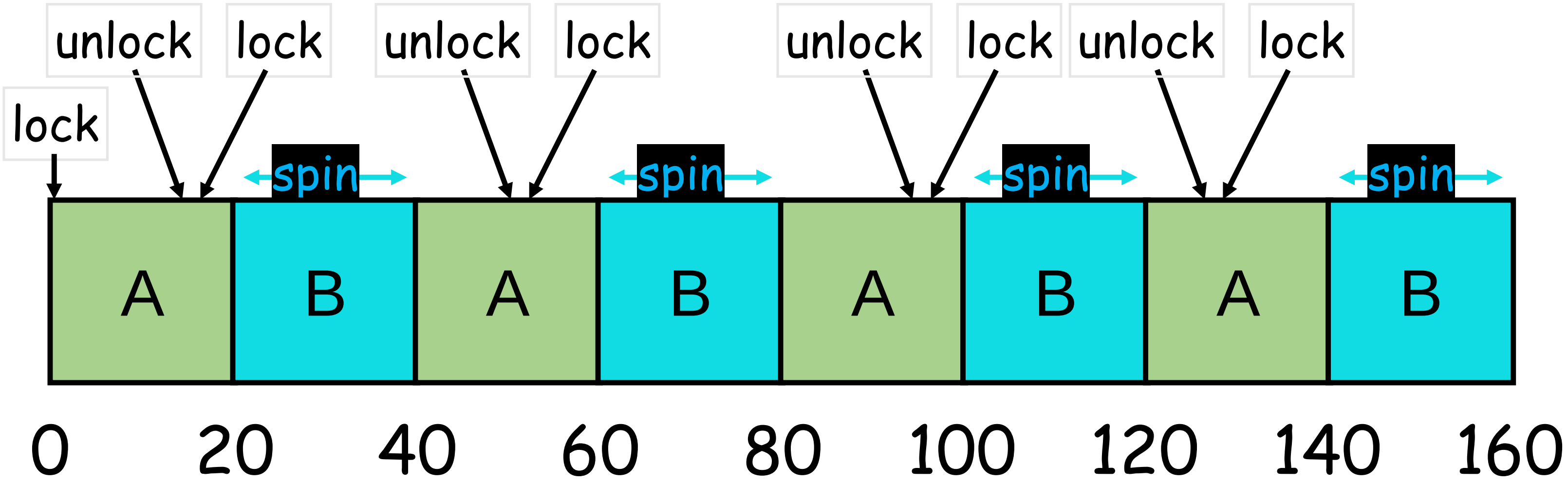
```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Evaluating Spin Locks

- Correctness:
  - Yes: Only one thread is allowed to enter CS
- Fairness:
  - No: a thread may spin forever
- Performance
  - Can be bad, especially, in a single processor system

Basic Spinlocks are Unfair



Scheduler is independent of locks/unlocks

Fetch-and-add

- Atomically increment value and return old value

```
int FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

Ticket lock using fetch-and-add

```
typedef struct __lock_t {
  int ticket;
  int turn;
} lock_t;
void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn = 0;
}
void lock(lock_t *lock) {
  …
}
void unlock(lock_t *lock) {
  …
}
```

Ticket lock using fetch-and-add

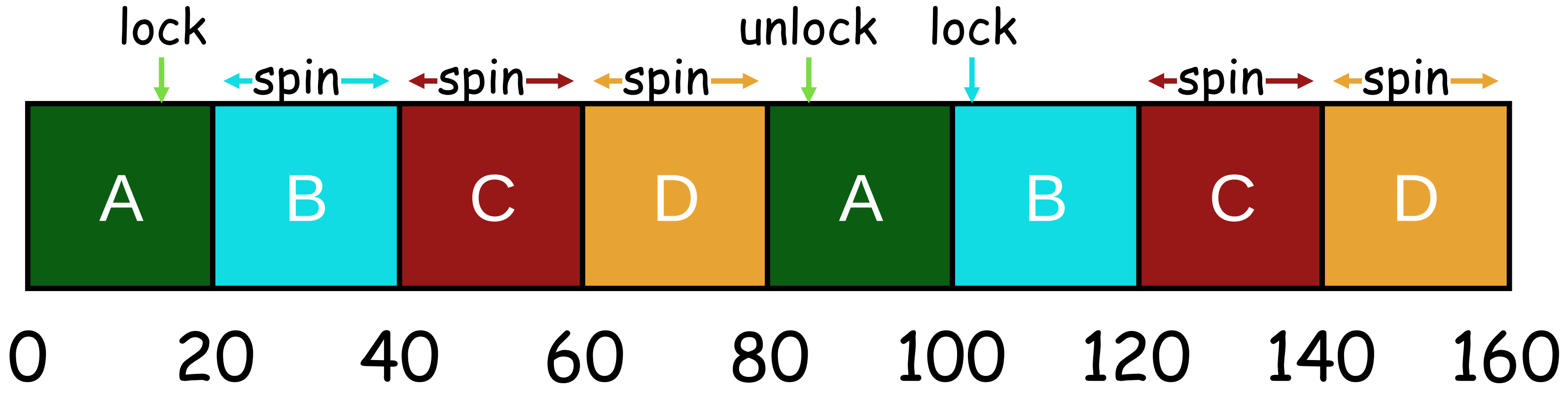
```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn); // spin
}

void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

Ensures fairness

Too much spinning

- Hardware based spin locks are simple
- But CPU cycles wasted by spinning

CPU Scheduler is Ignorant

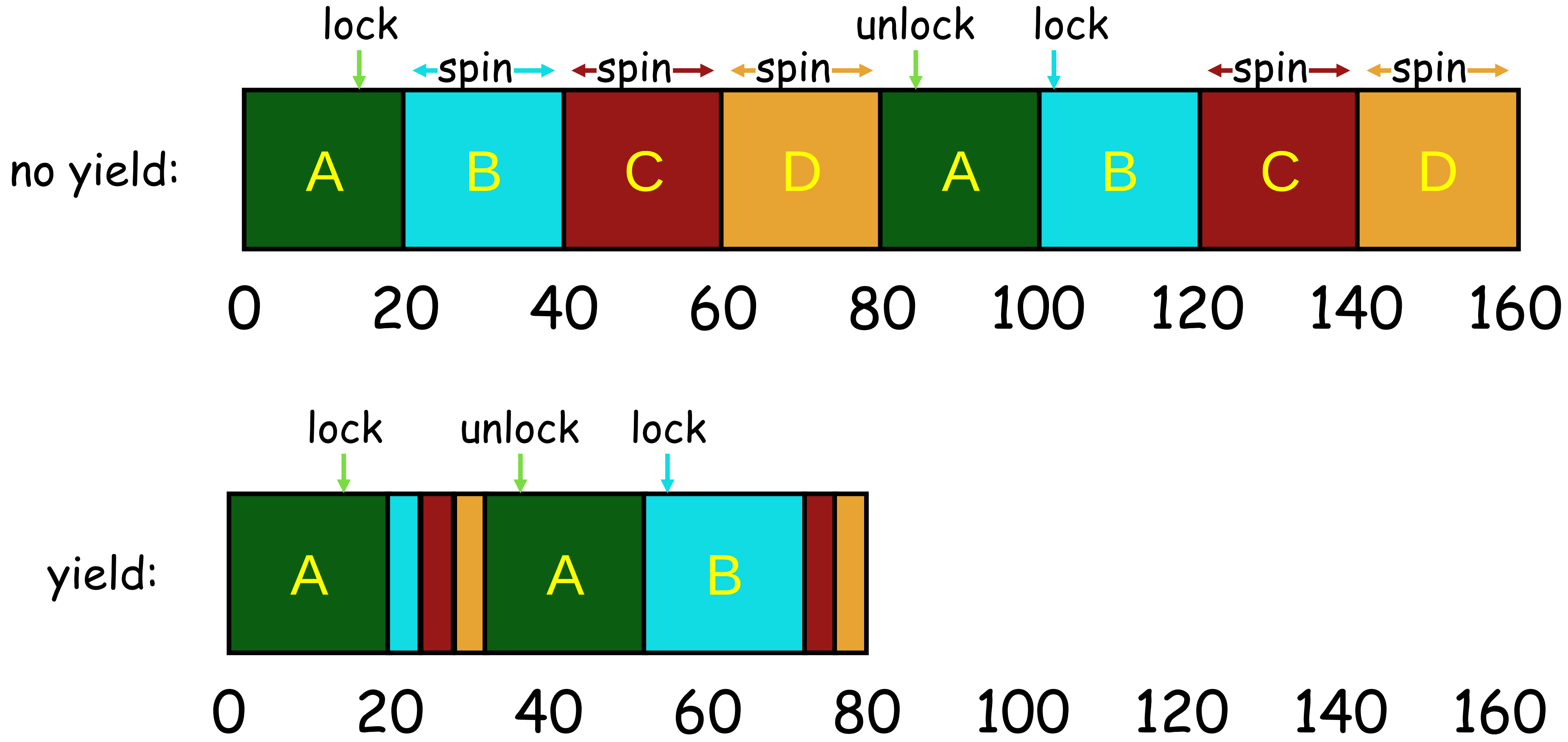


CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Ticket lock with yield

```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
            yield();
 }
 void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
 }
```

Yield Instead of Spin



Spinlock Performance

- Waste…
  - without yield: O(threads * **time_slice**)
  - with yield: O(threads * **context_switch**)
  - So even with yield, performance is slow with high thread contention

- Next improvement: Block and put thread on waiting queue instead of spinning

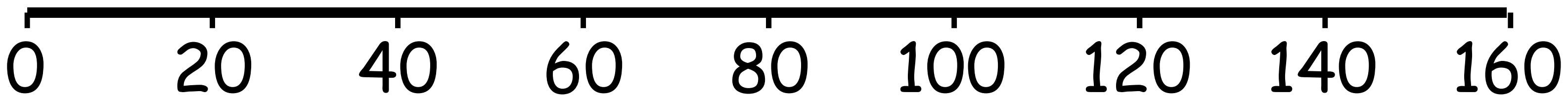
Lock Implementation: Block when Waiting

- Lock implementation removes waiting threads from scheduler ready queue (e.g., park() and unpark())
- **Scheduler** runs any thread that is **ready**
- Good separation of concerns

RUNNABLE: A, B, C, D

RUNNING: <empty>

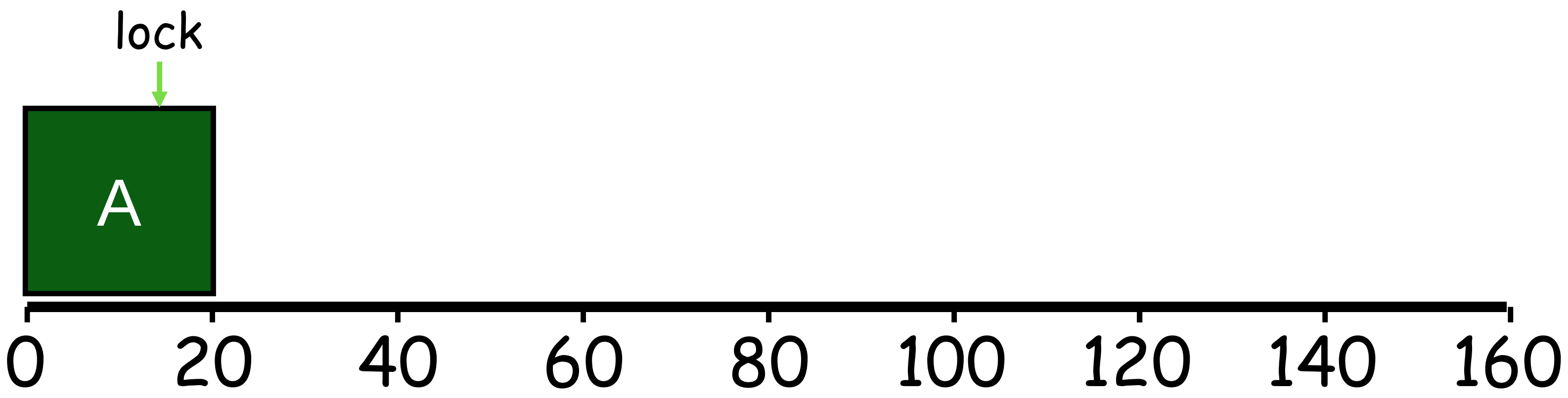
WAITING: <empty>



RUNNABLE: B, C, D

RUNNING: A

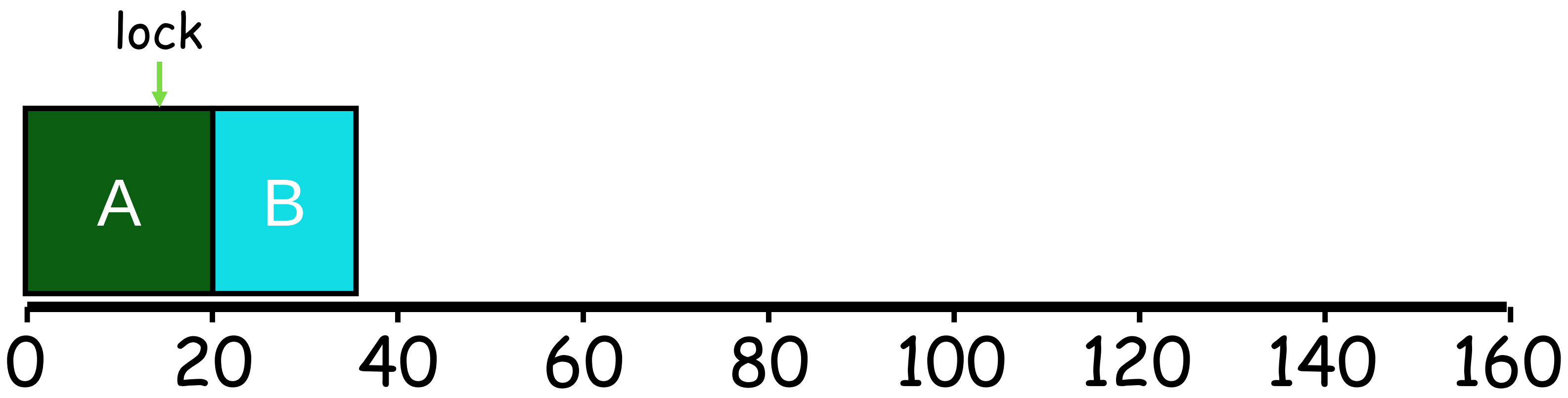
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING: B

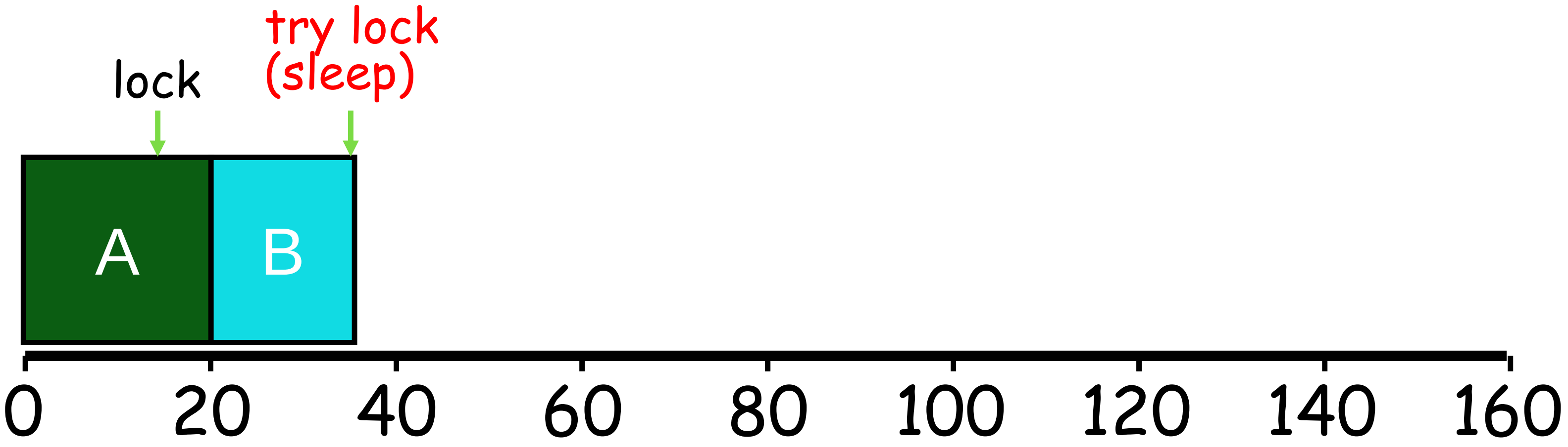
WAITING: <empty>



RUNNABLE: C, D, A

RUNNING:

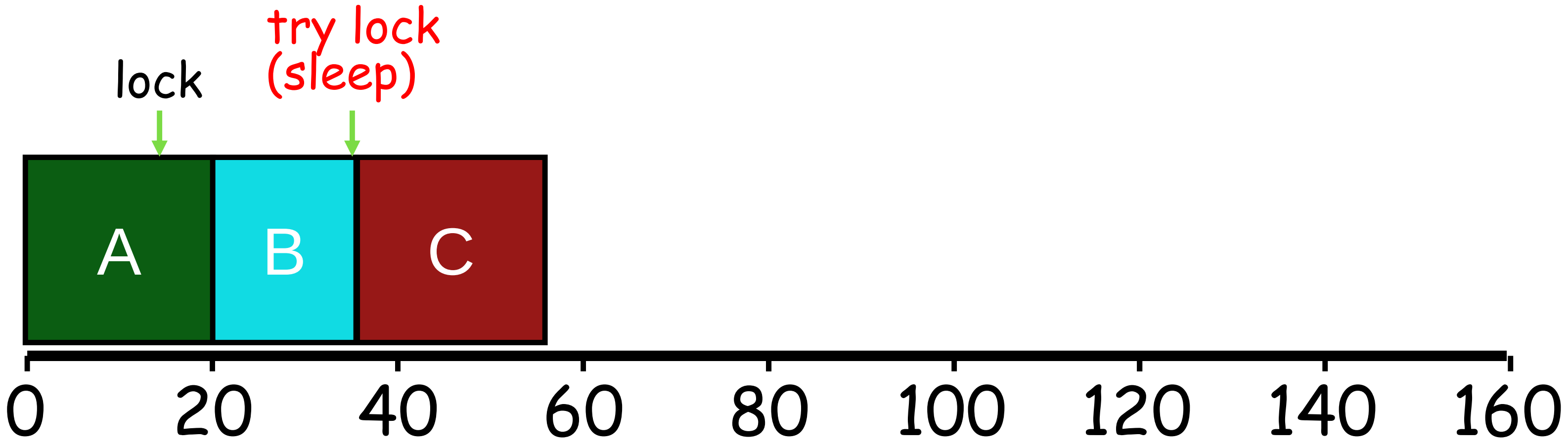
WAITING: B



RUNNABLE: D, A

RUNNING: C

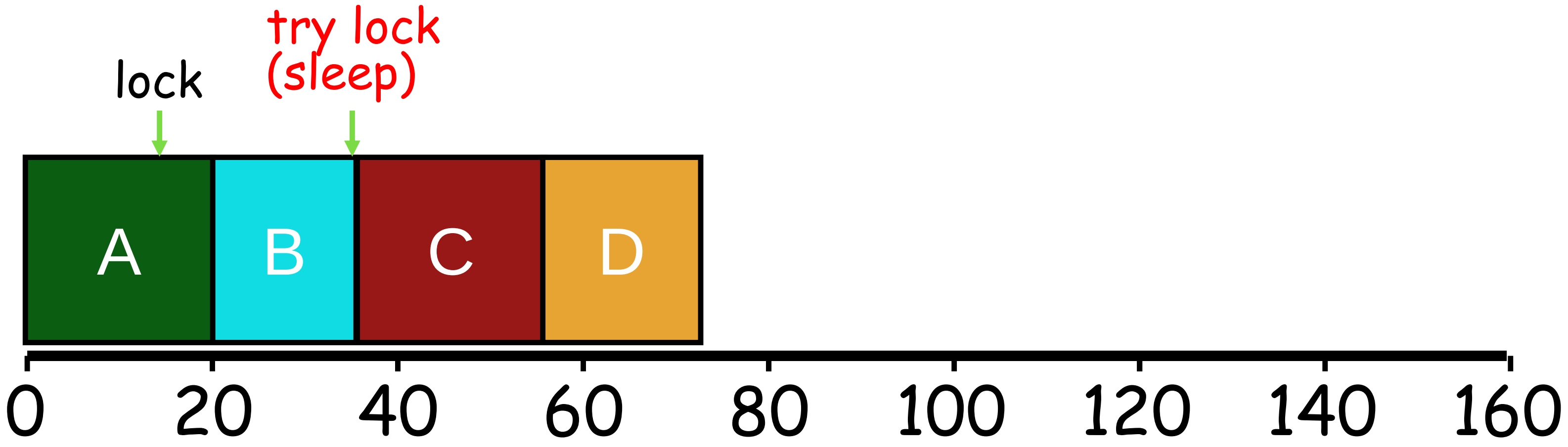
WAITING: B



RUNNABLE: A, C

RUNNING: D

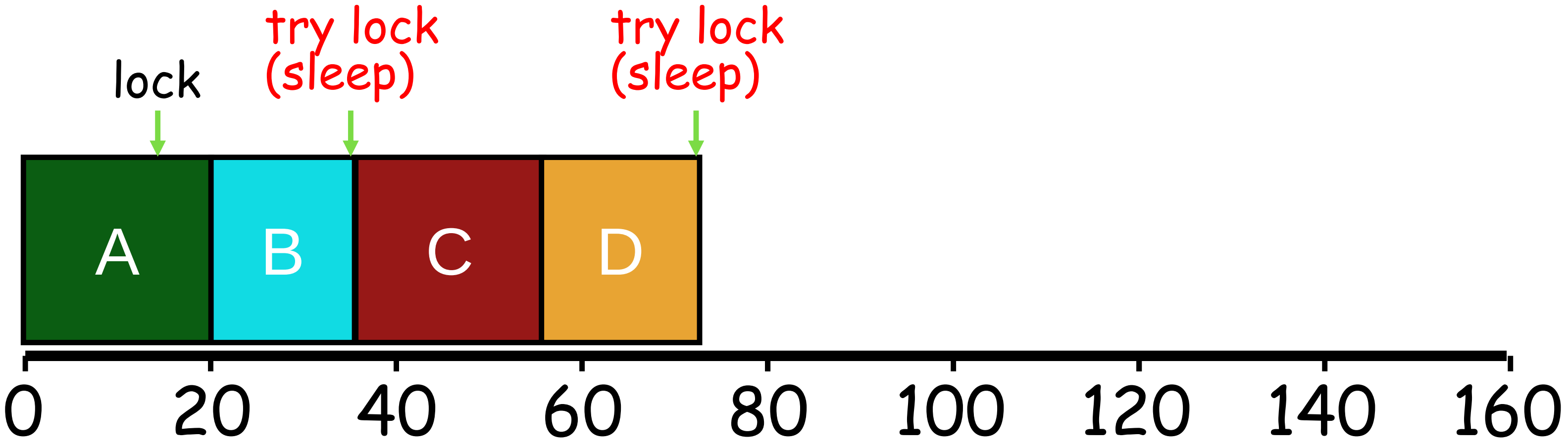
WAITING: B



RUNNABLE: A, C

RUNNING:

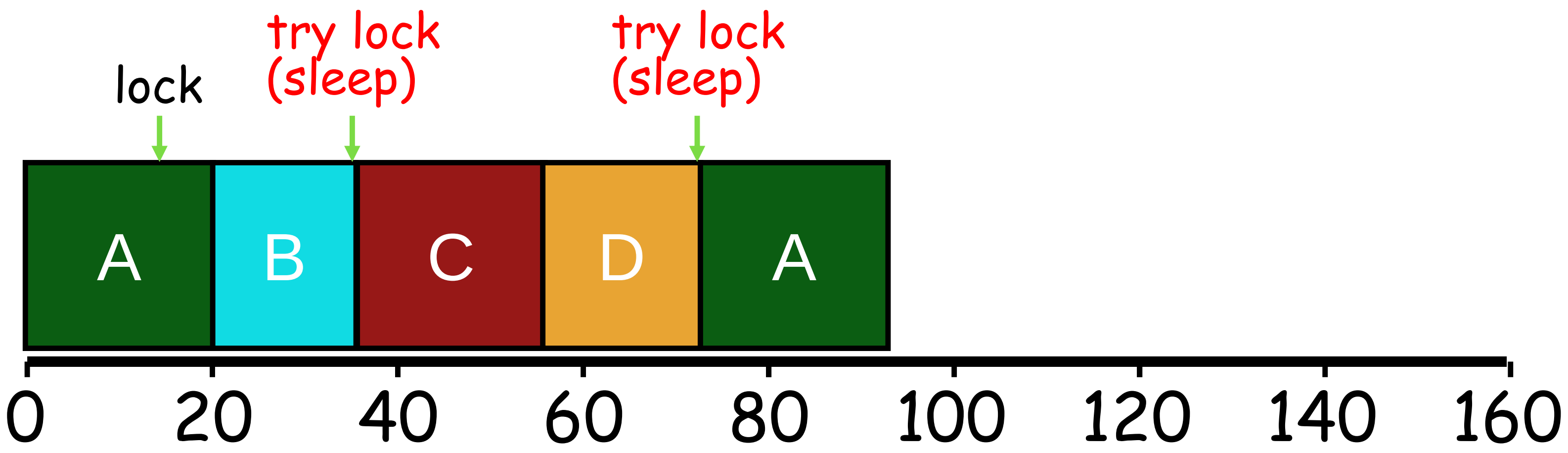
WAITING: B, D



RUNNABLE: C

RUNNING: A

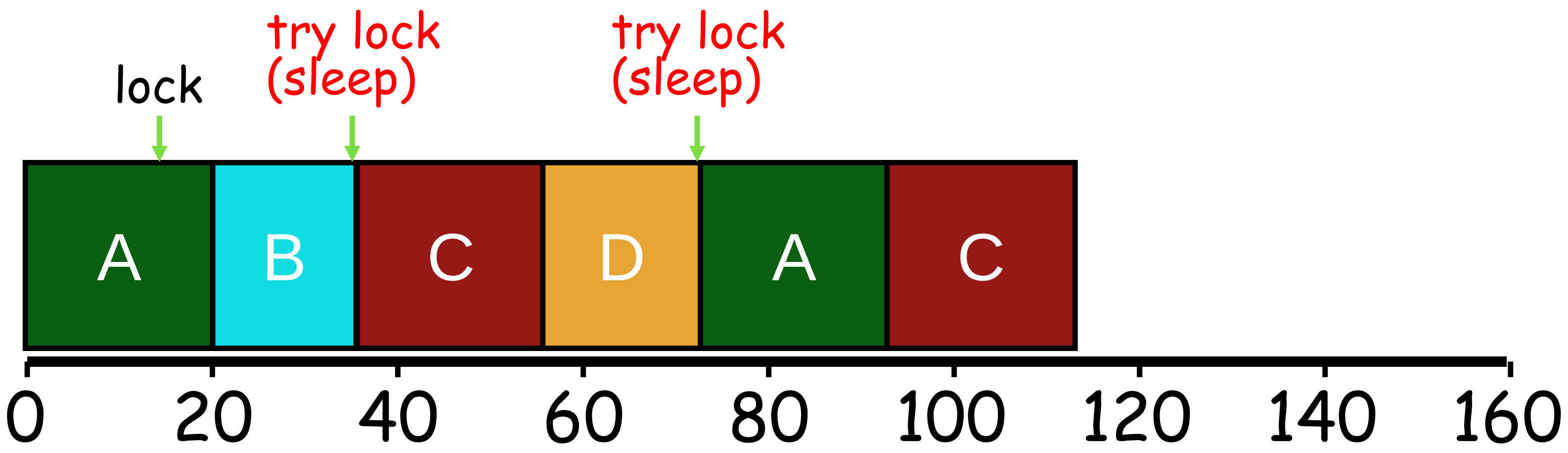
WAITING: B, D



RUNNABLE: A

RUNNING: C

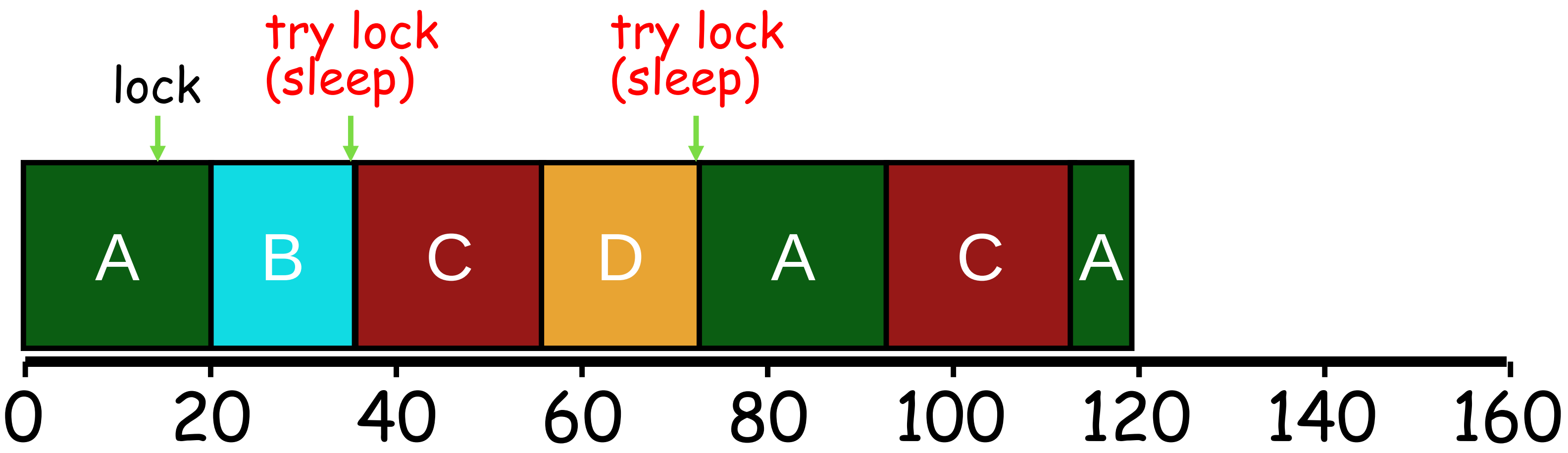
WAITING: B, D



RUNNABLE: C

RUNNING: A

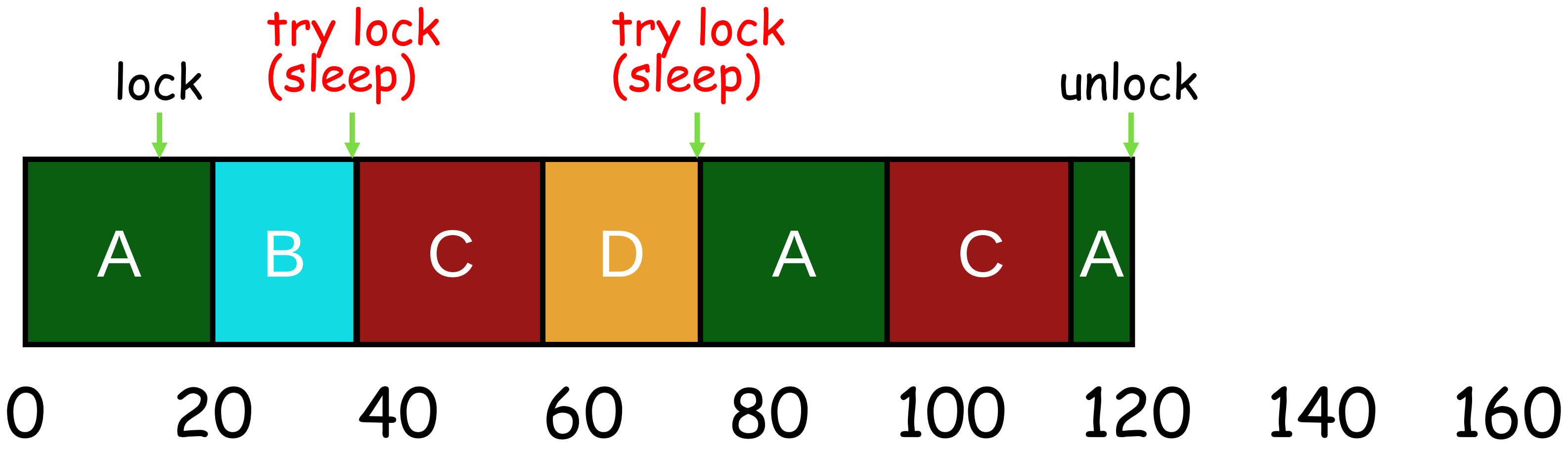
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

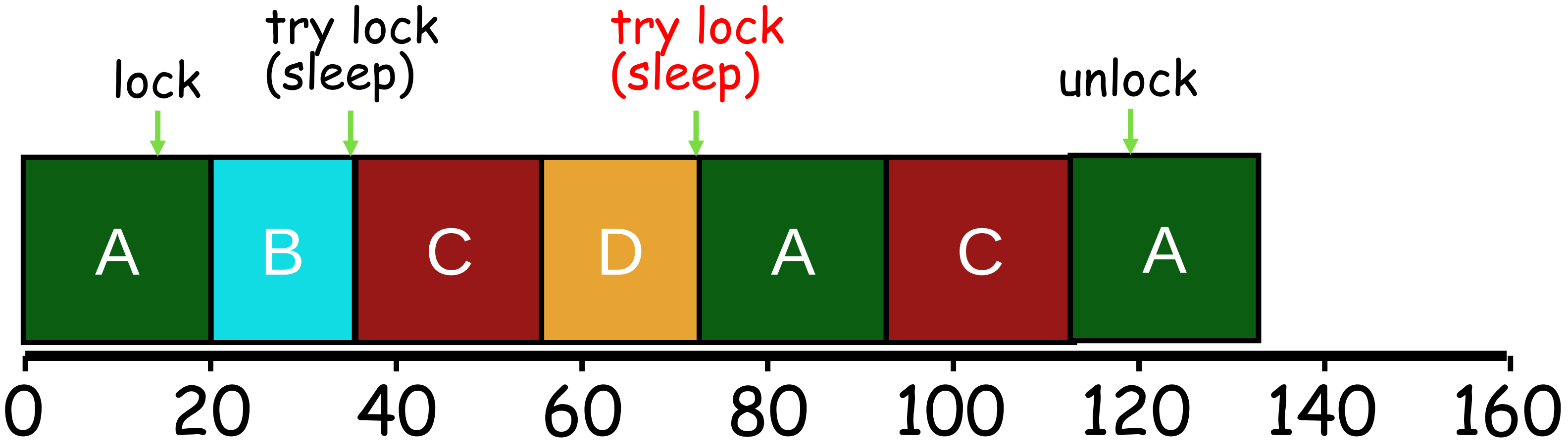
WAITING: D



RUNNABLE: B, C

RUNNING: A

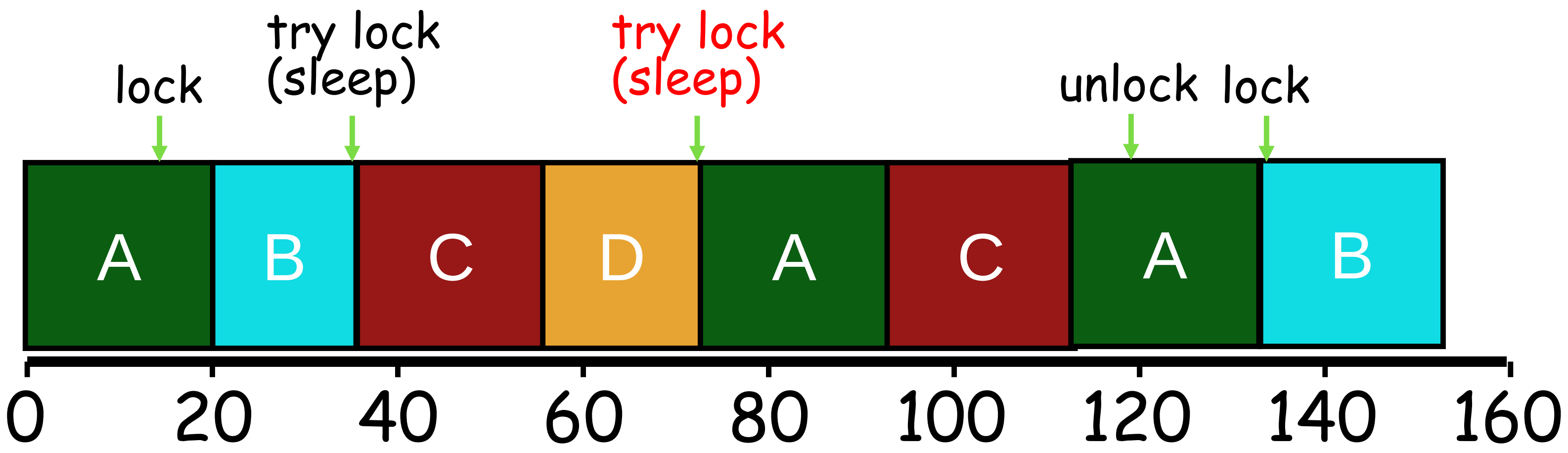
WAITING: D



RUNNABLE: C, A

RUNNING: B

WAITING: D



Lock Implementation: Block when Waiting

```
typedef struct {
  bool lock = false;
  queue_t q;
} LockT;
```

```
void lock(LockT *l) {
   if(testandset(&l->lock, 1)){
        qadd(l->q, tid);
        park();       // blocked
   }
}

void unlock(LockT *l) {
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
}
```

Problem: q is shared

Solution: lock q before modifying it

Lock Implementation: Block when Waiting

```
typedef struct {
  bool guard = false;
  bool lock = false;
  queue_t q;
} LockT;
```

```
void lock(LockT *l) {
   while (testandset(&l->guard, true));
   if (l->lock) {
             qadd(l->q, tid);
             l->guard = false;
             park();      // blocked
   } else {
             l->lock = true;
             l->guard = false;
   }
}

void unlock(LockT *l) {
   while (testandset(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

(a) use guard to lock q
(b) Why okay to **spin** on guard?
(c) In unlock(), why not set lock=false when unpark?
(d) What is the race condition?

Race Condition

Thread 1 (in lock)

```
if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;



        park();     // block
```

Thread 2 (in unlock)

```
while (testandset(&l->guard, true));
if (qempty(l->q)) // false!!
else unpark(qremove(l->q));
l->guard = false;
```

Problem: Guard not held when call park()
Unlocking thread may unpark() before other park()

Block when waiting: correct LOCK

```
Typedef struct {
  bool lock = false;
  bool guard = false;
  queue_t q;
} LockT;
```

setpark() fixes race condition

```
void lock(LockT *l) {
   while (TAS(&l->guard, true));
   if (l->lock) {
            qadd(l->q, tid);
            setpark(); // notify of plan
            l->guard = false;
            park(); // unless unpark()
   } else {
            l->lock = true;
            l->guard = false;
   }
}

void unlock(LockT *l) {
   while (TAS(&l->guard, true));
   if (qempty(l->q)) l->lock=false;
   else unpark(qremove(l->q));
   l->guard = false;
}
```

Spin Waiting vs Blocking

- Each approach is better under different circumstances
- Uniprocessor
  - Waiting process is scheduled --> Process holding lock isn't
  - Waiting process should always relinquish processor
  - Associate queue of waiters with each lock (as in previous implementation)
- Multiprocessor
  - Waiting process is scheduled --> Process holding lock might be
  - Spin or block depends on how long, t, before lock is released
    - Lock released quickly --> Spin-wait
    - Lock released slowly --> Block
    - Quick and slow are relative to context-switch cost, C

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Arpaci, Prof. Youjip, and other educators. Thanks to all of them.