Threads Locks

Synchronization Primitives

- Locks or Mutex or Mutex Lock
  - Used to guard a critical region
  - Guarantees mutual exclusion to the critical region
- Condition Variables and Semaphores
  - Used to order the execution of threads. Guarantee that thread Y waits until thread X finishes
  - Semaphores can be used for mutual exclusion and for ordering

pThread Code

```
pthread_mutex_t m;
volatile int balance = 0;
int loops = 2000;
void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&m);
        balance++;
        pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

Only one thread at a time

Two workers in critical region

```
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    pthread_mutex_init(&m, NULL);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

Critical Region

- When two threads update the same static variables. The update of one thread must be complete before the second thread. C code

```
balance++
```

critical region

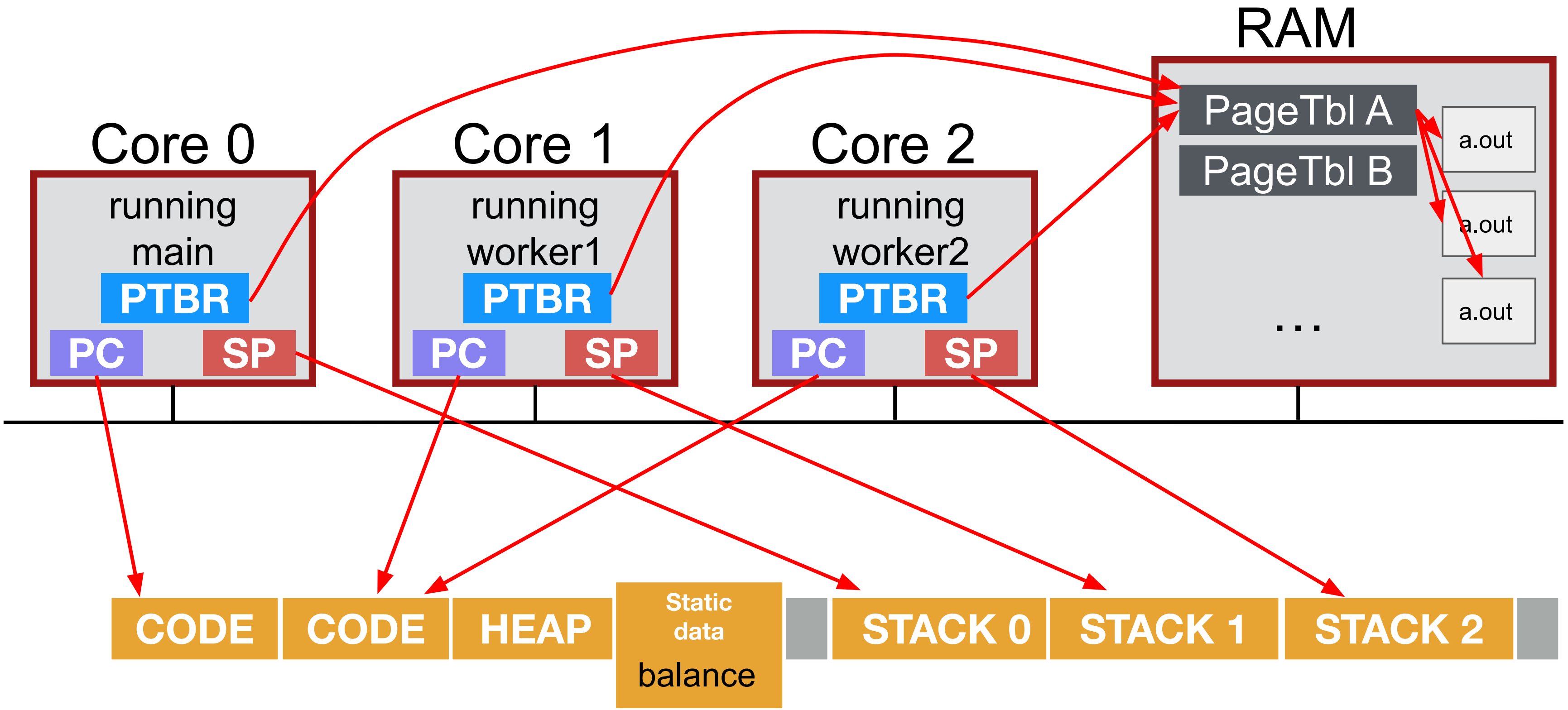
- The assembly instructions corresponding to the C code must be completed without getting interrupted

```
balance++
```

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

critical region

- Each thread must have mutual exclusive access to critical regions




```
Pthread_mutex_lock(&m);
          balance++;
Pthread_mutex_unlock(&m);
```

Locking Algorithm - Incorrect

```c
int lock = 0; // 0 means free

void acquire(int *lock) {
  while (*lock);
  *lock = 1;
}

void release(int *lock) {
  *lock = 0;
}
```

← spin till lock == 0

```
release:
   addi  sp,sp,-32
   sd    s0,24(sp)
   addi  s0,sp,32
   sd    a0,-24(s0)
   ld    a5,-24(s0)
   sw    zero,0(a5)
   ld    s0,24(sp)
   addi  sp,sp,32
   jr    ra
```

```
acquire:
   addi  sp,sp,-32
   sd    s0,24(sp)
   addi  s0,sp,32
   sd    a0,-24(s0)
.L2:
   ld    a5,-24(s0)
   lw    a5,0(a5)
   bne   a5,zero,.L2
   ld    a5,-24(s0)
   li    a4,1
   sw    a4,0(a5)
   ld    s0,24(sp)
   addi  sp,sp,32
   jr    ra
```

Race Condition with Load and Store

`*lock == 0`

<u>Thread 1</u>

```
while(*lock)
```

interrupted here

```
    *lock = 1
```

```
acquire:
  addi   sp,sp,-32
  sd   s0,24(sp)
  addi   s0,sp,32
  sd   a0,-24(s0)
.L2:
  ld   a5,-24(s0)
  lw   a5,0(a5)
  bne   a5,zero,.L2
  ld   a5,-24(s0)
  li   a4,1
  sw   a4,0(a5)
  ld   s0,24(sp)
  addi   sp,sp,32
  jr   ra
```

<u>Thread 2</u>

```
while(*lock)
    *lock = 1
```

Both threads grab lock!
Testing lock and setting lock are not atomic

Atomic Assembly Instructions

```
for (i = loop; i > 0; i++) {
   lock(mutex);
    balance = balance + 1;
   unlock(mutex)
 }
```

Spin Loop

Thread Snuck In

Critical Region

```
.var mutex
.var balance
.main
.top
.acquire
mov  mutex, %ax      # loop until mutex is 0
test $0, %ax         # we can be interrupted in the loop
jne .acquire
mov  $1, %ax         # maybe another thread got mutex as 0
xchg %ax, mutex      # atomic swap of 1 (in ax) and mutex
test $0, %ax         # if we get 0 back: lock is free!
jne .acquire         # if not, try again
# critical section - must have lock to enter
mov  balance, %ax    # get the value at the address
add  $1, %ax         # increment it
mov  %ax, balance    # store it back
# release lock
mov  $0, mutex
# see if we're still looping
sub  $1, %bx         # reg bx has i
test $0, %bx
jgt .top
halt
```

Spin Lock Implementation with xchg

```
typedef struct lock {
    int locked;
} lock;
void init(lock *lk) {
    lk->locked = 0;
}
void acquire(lock *lk) {
    while(xchg(&lk->locked, 1));
}
void release(lock *lk) {
    lk->locked = 0;
}
```

```
int xchg(int *adr, int new) {
    int old = *adr;
    *adr = new;
    return old;
}
```

```
xchg(volatile unsigned int *addr, unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m"
           (*addr), "=a" (result) : "1" (newval) : "cc");
    return result;
}
```

Lock Algorithm - Wrong

```
struct lock { int locked; }
acquire(l) {
  while(1){
    if(l->locked == 0){ // A
      l->locked = 1;    // B
      return;
    }
  }
}
```

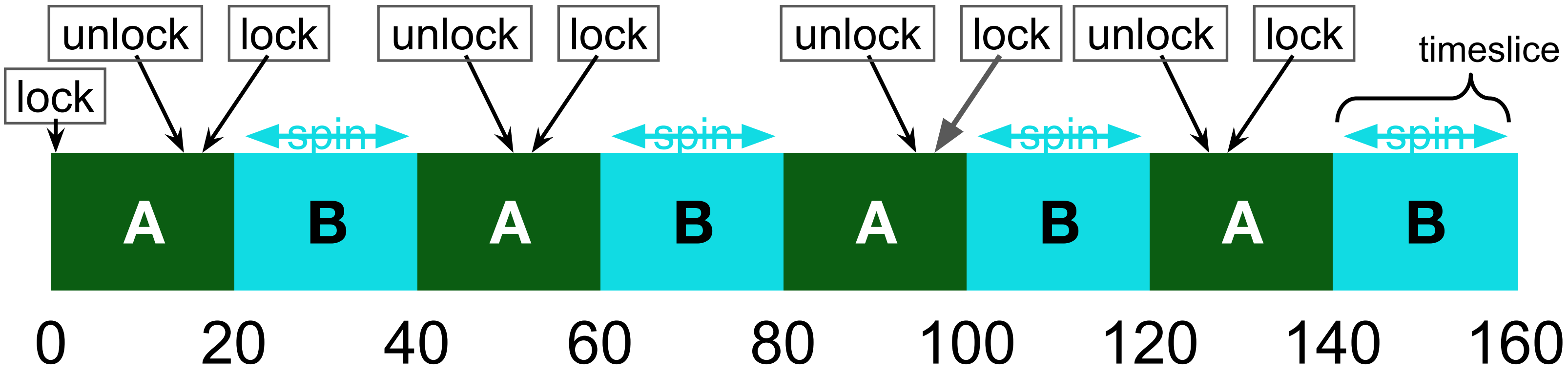
oops: race between lines A and B
how can we do A and B atomically?

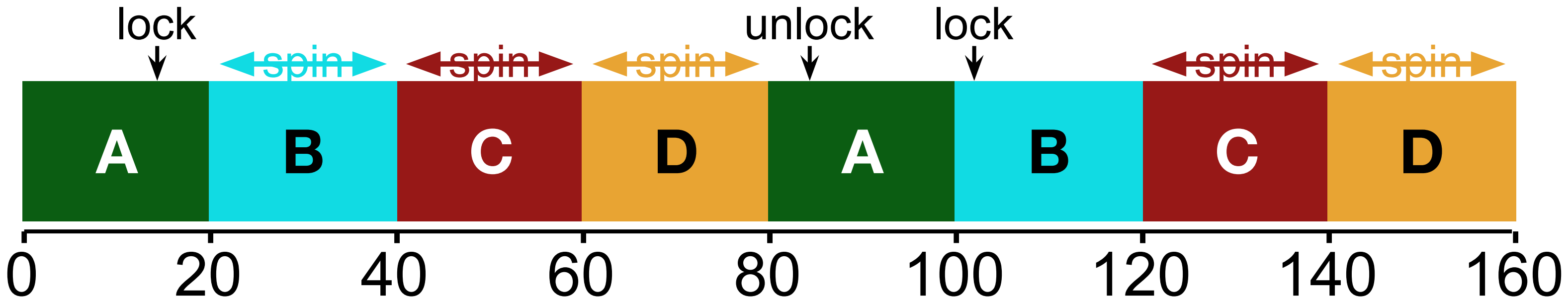
Xv6 Spin Lock Implementation

```
acquire(lk){
  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
}
```

- if lk->locked was already 1, sync_lock_test_and_set sets to 1 (again), returns 1, and the loop continues to spin
- if lk->locked was 0, at most one lock_test_and_set will see the 0; it will set it to 1 and return 0; other test_an_set will return 1
- this is a "spin lock", since waiting cores "spin" in acquire loop

Spinlocks are not fair - threads wait equally





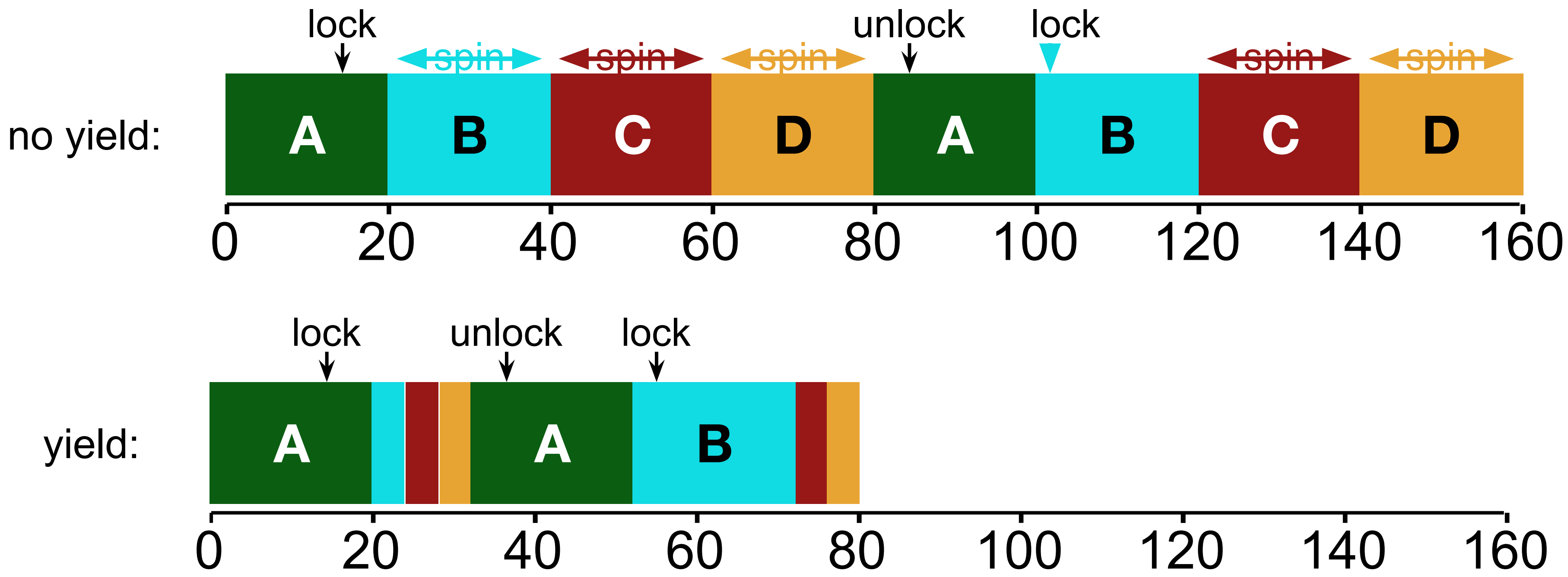
Scheduler is not aware of locks/unlocks

Spin Lock Implementation with `yield`

```
typedef struct lock {
    int locked;
} lock;
void init(lock *lk) {
    lk->locked = 0;
}
void acquire(lock *lk) {
    while(xchg(&lk->locked, 1))
      yield();
}
void release(lock *lk) {
    lk->locked = 0;
}
```

```
int xchg(int *adr, int new) {
    int old = *adr;
    *adr = new;
    return old;
}
```

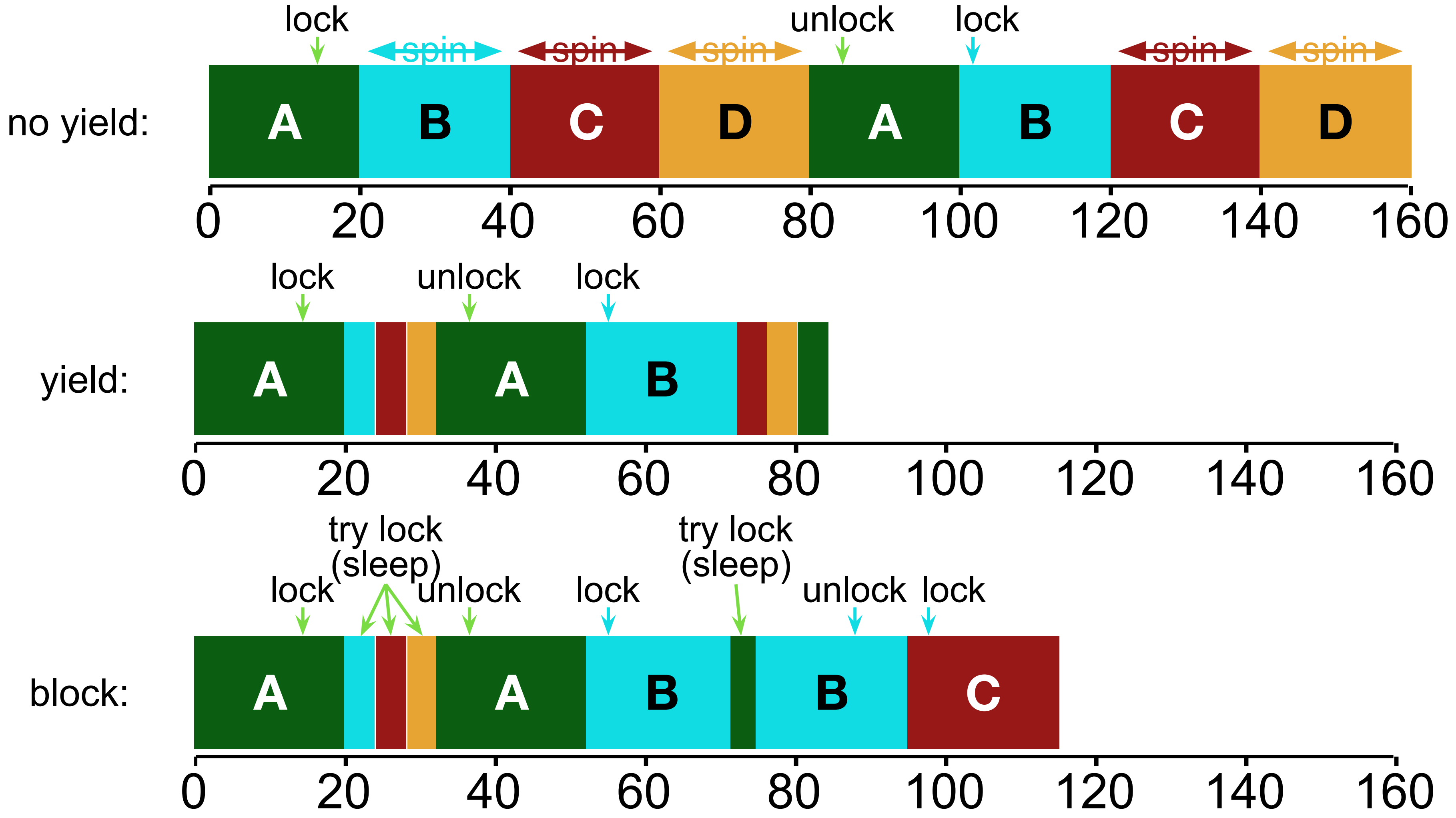
Yield Instead of Spin



Spinlock Performance

- Fast when…
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch
- Slow when…
  - one CPU
  - locks held a long time
    - Without yield: O(num_threads * time_slice)
    - With yield: O(num_threads * context_switch_time)
  - disadvantage: spinning is wasteful

- **Next**: Block thread on queue instead of spinning



no yield:
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
yield:
lock
unlock
lock
A
A
B
0
20
40
60
80
100
120
140
160
block:
try lock
(sleep)
try lock
(sleep)
lock
unlock
lock
unlock
lock
A
A
B
B
C
0
20
40
60
80
100
120
140
160

Waiting on Mutex - Spin vs Blocking

- Uniprocessor - thread waiting on mutex is scheduled
  - Blocking is better - does not waste CPU cycles
  - Associate queue of waiters with each lock
- Multiprocessor - thread waiting on mutex is scheduled
  - Spin or block depends on how long, t, before lock is released (proc spins) and the time, C, of a context switch
    - if t < C (i.e., lock released quickly) - spinning is better
    - if t > C (i.e., lock released slowly) - blocking is better
    - Quick and slow are relative to the length of time, t, a lock is held and context-switch time, C
    - You must understand your system design and implementation to know length a lock is held is

Spinning vs Blocking Analysis

- If we know how lock held time, we can determine optimal behavior
- CPU time wasted when spin-waiting is **t**, time spent spinning
- CPU time wasted when blocking is **C**, time of a context switch
- **Optimal**: spin time **t** is less that context time **C (t < C)**, it is best to spin
- **Optimal**: spin time **t** is greater than context time **C (t>C)**, it is best to block
- **Optimal**: achieved by knowing time lock will be held, which we don't know
- **Algorithm**: Spin-wait for **C** then block. This is 2 times the optimal
- **Case 1**: **t < C**: optimal spin-waits for **t**; Algorithm spin-waits **t** too
- **Case 2**: **t > C**: optimal blocks immediately (cost of **C**); we pay spin **C** then block (cost of 2 **C**); **2C / C**, yields a algorithm that is 2 times the optimal

Concurrency Tools and Problems

- Concurrency Tools (Primitives)
  - Locks, Condition Variables, Semaphores
- Problems
  - Mutual exclusion
    - Threads A and B run separately in a critical region
    - Solved with locks
  - Ordering
    - Thread B runs after thread A (or A runs after B)
    - Solved with condition variables and semaphores