Concurrency: Locks

Questions answered in this lecture:

Review threads and mutual exclusion for critical sections

How can locks be used to protect shared data structures such as linked lists?

Can locks be implemented by disabling interrupts?

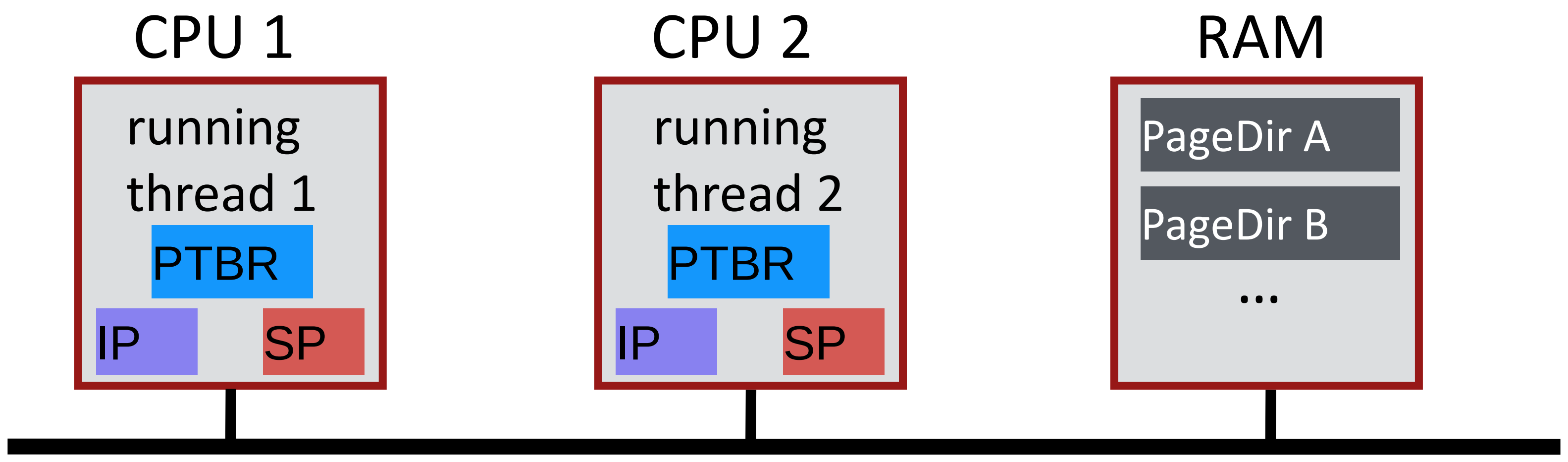
Can locks be implemented with loads and stores?

Can locks be implemented with atomic hardware instructions?

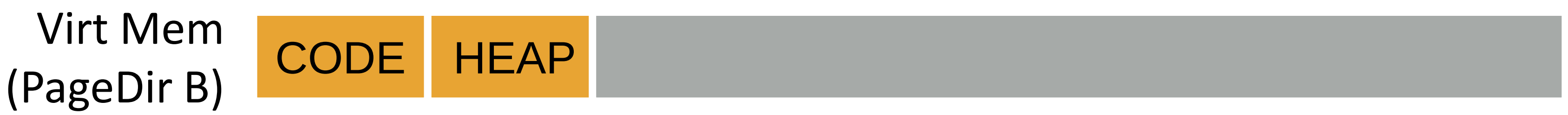
Are spinlocks a good idea?

How can threads block instead of spin-waiting while waiting for a lock?

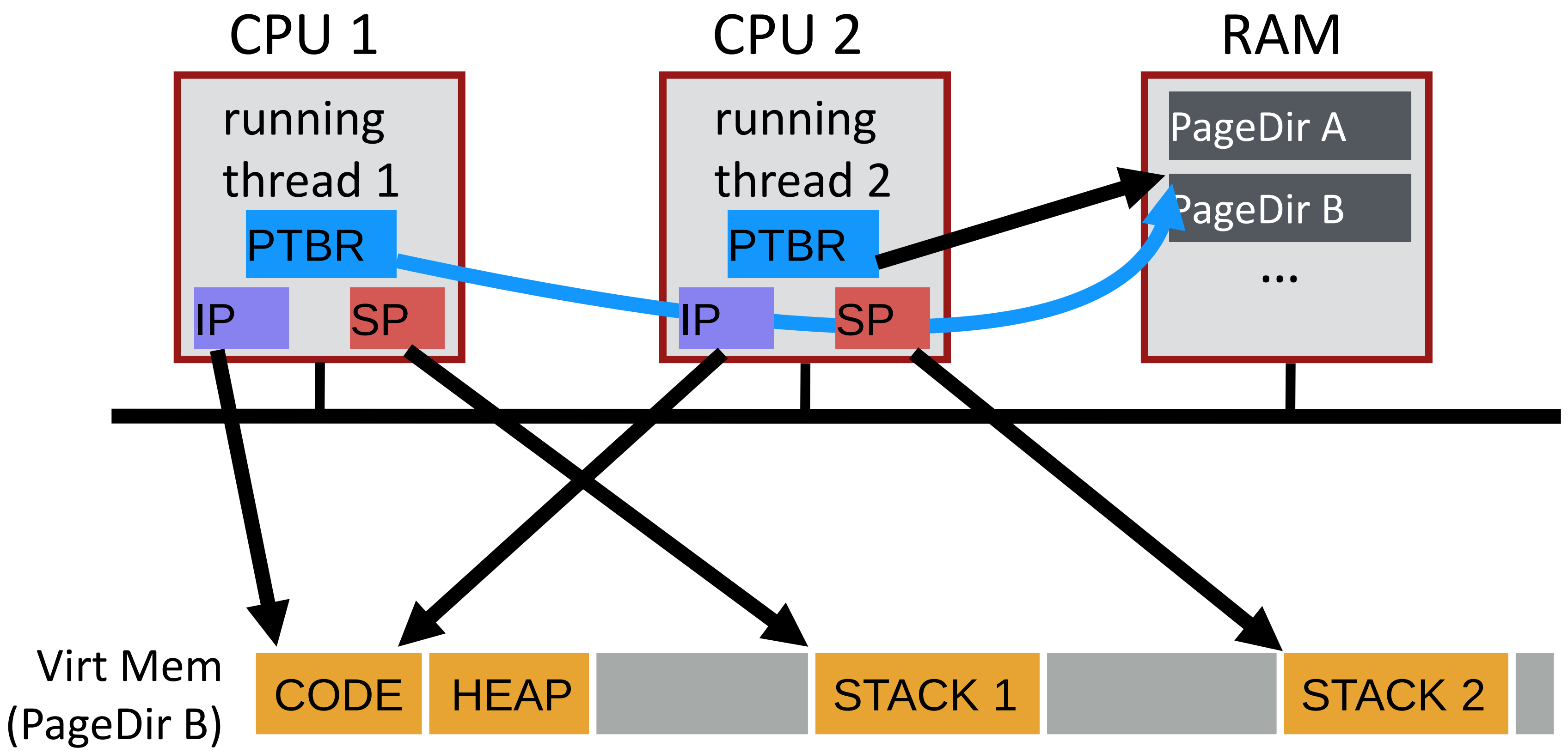
When should a waiting thread block and when should it spin?



PTBRs (Page Table Base Register) are the same;
IPs (Instruction pointer) and SPs (Stack pointer) are different!



Review:
Which registers store the same/different values across threads?


CPU 1
running thread 1
PTBR
IP
SP
CPU 2
running thread 2
PTBR
IP
SP
RAM
PageDir A
PageDir B
...
Virt Mem (PageDir B)
CODE
HEAP
STACK 1
STACK 2

Review: What is needed for Correctness?

- **`Balance = balance + 1;`**
- **Instructions accessing shared memory must execute as uninterruptable group**
  - Need instructions to be atomic

```
mov (0x123), %eax
add $0x1, %eax        — critical section
mov %eax, (0x123)
```

- More general:
- Need mutual exclusion for critical sections
- if process A is in critical section C, process B can’t
  - (okay if other processes do unrelated work)

badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
               *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

How can we fix this using semaphores?

goodcnt.c: Mutex Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

- Surround critical section with *lock* and *unlock*:

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
 }
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Function	badcnt	goodcnt
Time (ms) niters = 10^6	12	214
Slowdown	1.0	17.8

Other Examples

- **Consider multi-threaded applications that do more than increment shared balance**

- **Multi-threaded application with shared linked-list**
  - All concurrent:
    - Thread A inserting element a
    - Thread B inserting element b
    - Thread C looking up element c

Shared Linked List

```
Void List_Insert(list_t *L, int key)
{
        node_t *new =
        malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L, int key)
{
        node_t *tmp = L->head;
        while (tmp) {
            if (tmp->key == key)
                return 1;
            tmp = tmp->next;
        }
        return 0;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

What can go wrong?
Find schedule that leads to problem?

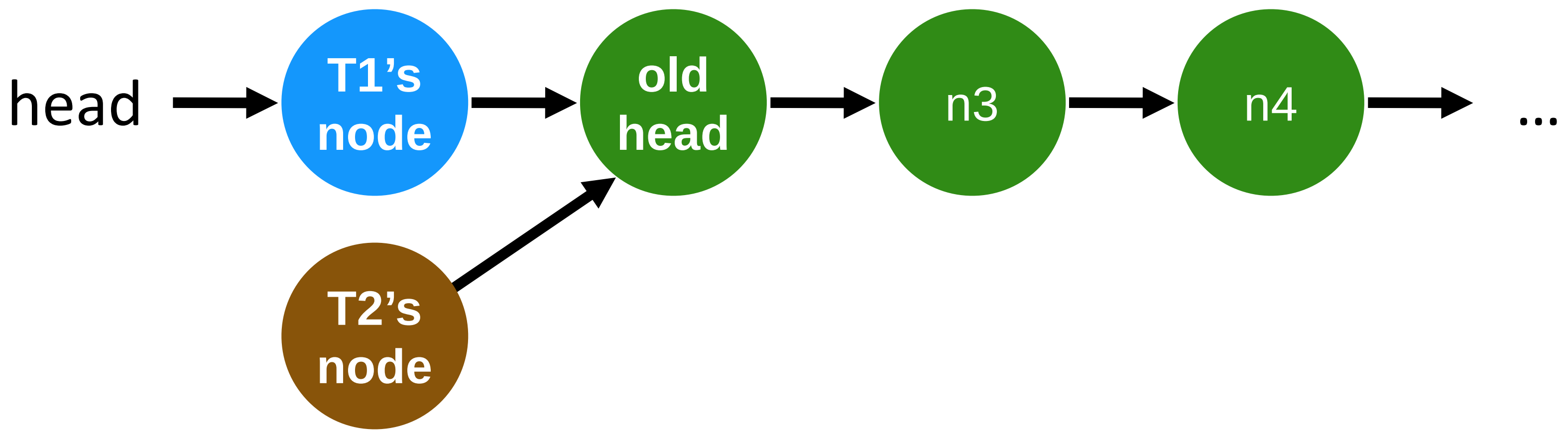
Linked-List Race

Thread 1	Thread 2
new->key = key	
new->next = L->head	
	new->key = key
	new->next = L->head
	L->head = new
L->head = new	

Both entries point to old head

Only the entry created by Thread 1 can be the new head.

Resulting Linked List



Locking Linked Lists

```
Void List_Insert(list_t *L, int key)
{
        node_t *new =

        malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
int List_Lookup(list_t *L, int key)
{
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                        return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

How to add locks?

Locking Linked Lists

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
}
```

```
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

Typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

Void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock,
                NULL);
}
```

`pthread_mutex_t lock;`

One lock per list

Locking Linked Lists : Approach #1

```
Pthread_mutex_lock(&L->lock);      →   Void List_Insert(list_t *L, int key) {
                                               node_t *new =
                                               malloc(sizeof(node_t));
                                               assert(new);
                                               new->key = key;
                                               new->next = L->head;
                                               L->head = new;
Pthread_mutex_unlock(&L->lock);    →   }
```

Consider everything critical section
Can critical section be smaller?

```
Pthread_mutex_lock(&L->lock);      →   int List_Lookup(list_t *L, int key) {
                                               node_t *tmp = L->head;
                                               while (tmp) {
                                                       if (tmp->key == key)
                                                       return 1;
                                                       tmp = tmp->next;
                                               }
Pthread_mutex_unlock(&L->lock);    →           return 0;
                                       }
```

Locking Linked Lists : Approach #2

Critical section small as possible

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
        malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}
```

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

```
int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Locking Linked Lists : Approach #3

What about Lookup()?

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Pthread_mutex_lock(&L->lock);

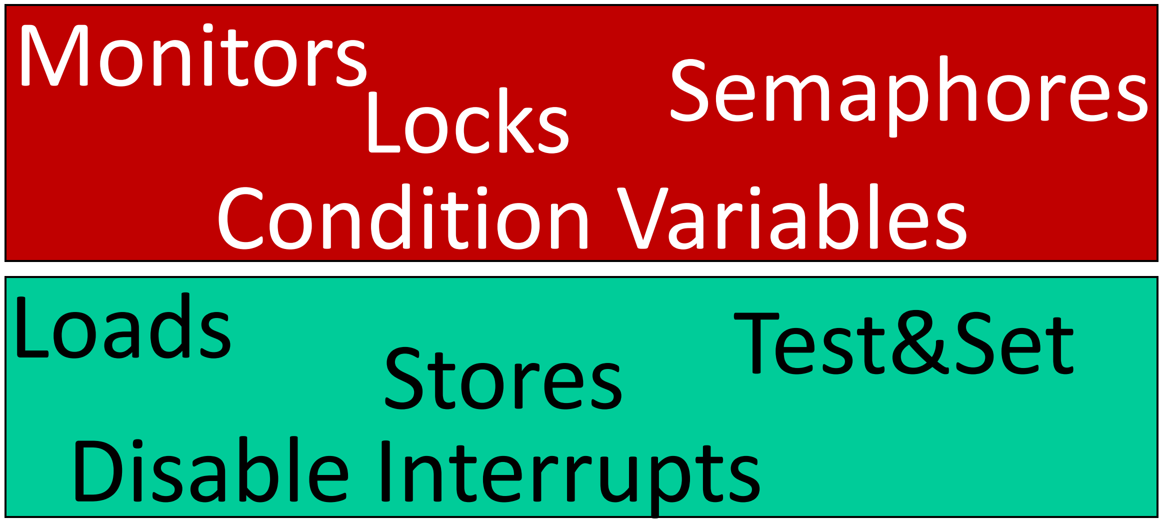
If no List_Delete(), locks not needed

Pthread_mutex_unlock(&L->lock);

```
Void List_Insert(list_t *L, int key) {
        node_t *new =
        malloc(sizeof(node_t));
        assert(new);
        new->key = key;
        new->next = L->head;
        L->head = new;
}

int List_Lookup(list_t *L, int key) {
        node_t *tmp = L->head;
        while (tmp) {
                if (tmp->key == key)
                return 1;
                tmp = tmp->next;
        }
        return 0;
}
```

Implementing Synchronization

- **Build higher-level synchronization primitives in OS**
  - Operations that ensure correct ordering of instructions across threads
- **Motivation: Build them once and get them right**



Lock Implementation Goals

- **Correctness**
  - Mutual exclusion
    - Only one thread in critical section at a time
  - Progress (deadlock-free)
    - If several simultaneous requests, must allow one to proceed
  - Bounded (starvation-free)
    - Must eventually allow each waiting thread to enter
- **Fairness**
  - Each thread waits for same amount of time
- **Performance**
  - CPU is not used unnecessarily (e.g., spinning)

Implementing Synchronization

- **To implement, need atomic operations**
- **Atomic operation: No other instructions can be interleaved**
- **Examples of atomic operations**
  - Code between interrupts on uniprocessors
    - Disable timer interrupts, don’t do any I/O
  - Loads and stores of words
    - Load r1, B
    - Store r1, A
  - **Special hw instructions**
    - **Test&Set**
    - **Compare&Swap**

Implementing Locks: w/ Interrupts

- **Turn off interrupts for critical sections**
  - Prevent dispatcher from running another thread
  - Code between interrupts executes atomically

```
Void acquire(lockT *l) {
      disableInterrupts();
}
Void release(lockT *l) {
      enableInterrupts();
}
```

- **Disadvantages?**
  - Privilege mode, malicious or buggy program run forever
  - Not work for multiprocessors, thread will be scheduled on other
  - Make CPUs miss other interrupt events, such as disk finishes read
  - Inefficient

Implementing locks: w/ Load+Store

- **Code uses a single shared lock variable**

```
boolean lock = false; // shared variable
void acquire(Boolean *lock) {
     while (*lock) /* wait */ ;
          *lock = true;
}
void release(Boolean *lock) {
     *lock = false;
}
```

- **Does it work?**

Race Condition with LOAD and STORE

```
*lock == 0 initially

Thread 1                        Thread 2
while(*lock == 1)
                                    while(*lock == 1)
                                    *lock = 1
*lock = 1
```

- Both threads grab lock!
- Problem: Testing lock and setting lock are not atomic

Can we implement a software lock without the help of hardware?

Peterson's Algorithm

- **Assume only two threads (tid = 0, 1) and use just loads and stores**

```
int turn = 0; // shared
Boolean lock[2] = {false, false};
Void acquire() {
      lock[tid] = true;
      turn = 1-tid;
      while (lock[1-tid] && turn == 1-tid) /* wait */ ;
}
Void release() {
      lock[tid] = false;
}
```

Enter if the other thread does not have lock or its your turn

Different Cases: All work

- Only thread 0 wants lock

```
lock[0] = true;
turn = 1;
while (lock[1] && turn == 1);
```

- Thread 0 and thread 1 both want lock;

```
lock[0] = true;

turn = 1;
                              lock[1] = true;

                              turn = 0;
while (lock[1] && turn == 1);
                              while (lock[0] && turn == 0);
```

Different Cases: All work

- Thread 0 and thread 1 both want lock

```
Lock[0] = true;

                              Lock[1] = true;

                              turn = 0;

turn = 1;

while (lock[1] && turn == 1);

                              while (lock[0] && turn == 0);
```

Different Cases: All work

- Thread 0 and thread 1 both want lock;

```
lock[0] = true;

turn = 1;
                               lock[1] = true;


while (lock[1] && turn == 1);
                               turn = 0;

                               while (lock[0] && turn == 0);


while (lock[1] && turn == 1);
```

Peterson’s Algorithm

- Try to let the other run first

- If the other does not lock, it will proceed
- If both locks, let the last turn assignment determine who will run
- After one finishes, unlock itself, and the other will proceed

Peterson’s Algorithm: Intuition

- Mutual exclusion

- Progress

- Bounded

Peterson's Algorithm: Intuition

- **Mutual exclusion: Enter critical section if and only if**
  - Other thread does not want to enter
  - Other thread wants to enter, but your turn
- **Progress: Both threads cannot wait forever at while() loop**
  - Completes if other process does not want to enter
  - Other process (matching turn) will eventually finish
- **Bounded waiting (not shown in examples, why?)**
  - Each process waits at most one critical section

- Problem?

Modern architecture such as x86 is not sequential consistency: Two observers (cores) might see two events (writes) in different order

xchg: atomic exchange, or test-and-set

```
// xchg(int *addr, int newval)
// return what was pointed to by addr
// at the same time, store newval into addr
int xchg(int *addr, int newval) {
    int old = *addr;
    *addr = newval;
    return old;
}

static inline uint
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 implementation with XCHG

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

void init(lock_t *lock) {
      lock->flag = ??;
}

void acquire(lock_t *lock) {
      ????;
      // spin-wait (do nothing)
}

void release(lock_t *lock) {
      lock->flag = ??;
}
```

int xchg(int *addr, int newval)

XCHG Implementation

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

void init(lock_t *lock) {
      lock->flag = 0;
}

void acquire(lock_t *lock) {
      while(xchg(&lock->flag, 1) == 1) ;
      // spin-wait (do nothing)
}

void release(lock_t *lock) {
      lock->flag = 0;
}
```

Other Atomic HW Instructions

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

```
 void acquire(lock_t *lock) {
              while(CompareAndSwap(&lock->flag, ?, ?)
                                                          == ?) ;
              // spin-wait (do nothing)
 }
```

Other Atomic HW Instructions

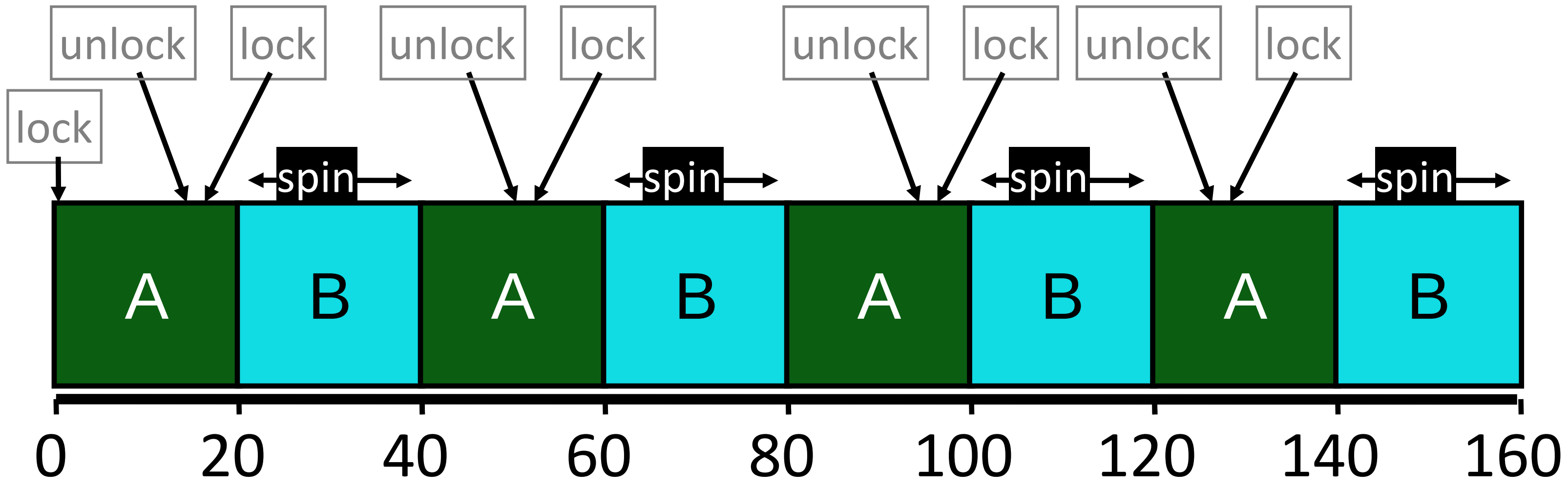
```
int CompareAndSwap(int *addr, int expected, int new) {
    int actual = *addr;
    if (actual == expected)
        *addr = new;
    return actual;
}
```

```
 void acquire(lock_t *lock) {
              while(CompareAndSwap(&lock->flag, 0, 1)
                                                  == 1) ;
              // spin-wait (do nothing)
 }
```

Lock Implementation Goals

- **Correctness**
  - Mutual exclusion
    - Only one thread in critical section at a time
  - Progress (deadlock-free)
    - If several simultaneous requests, must allow one to proceed
  - Bounded (starvation-free)
    - Must eventually allow each waiting thread to enter
- **Fairness**
  - **Each thread waits for same amount of time**
- **Performance**
  - CPU is not used unnecessarily

Basic Spinlocks are Unfair



Scheduler is independent of locks/unlocks

Fairness: How to implement?

- **The spins lead to competition**
- **Competition results in unpredictability and unfairness**
- **Any solution for a fair lock?**

- **Spin**
  - always waiting at a bank counter
  - fight with other customers when someone finishes services
  - you may never be served if weak

- **Any inspiration for implementing a fair lock?**

Fairness: Ticket Locks

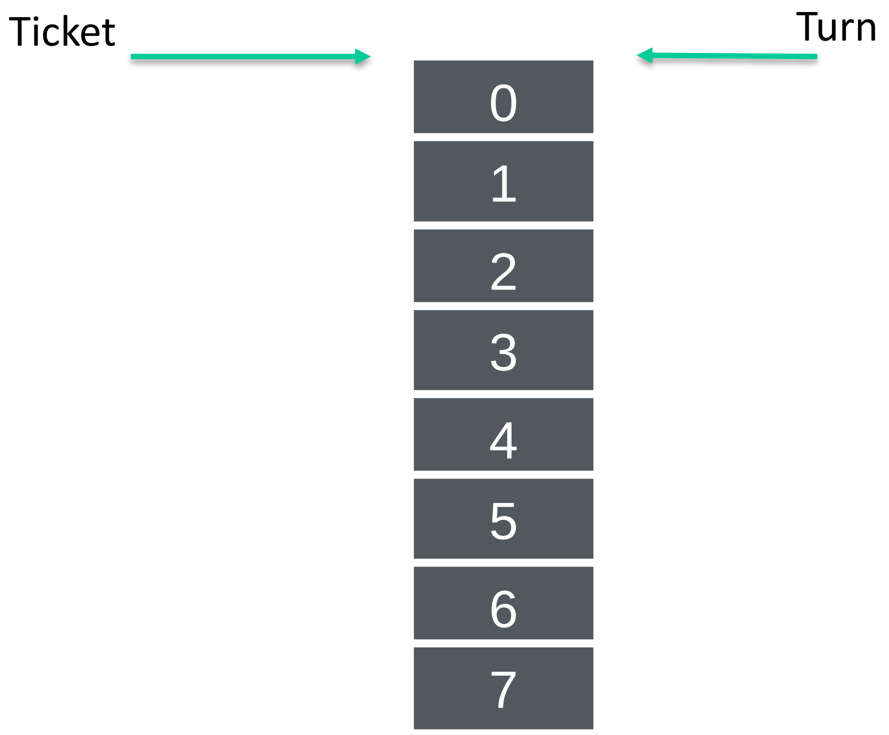
- **Idea: reserve each thread’s turn to use a lock.**
- **Each thread spins until their turn.**
- **Use new atomic primitive, fetch-and-add:**

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

- **Acquire: Grab ticket;**
  Spin while not thread’s ticket != turn
- **Release: Advance to next turn**

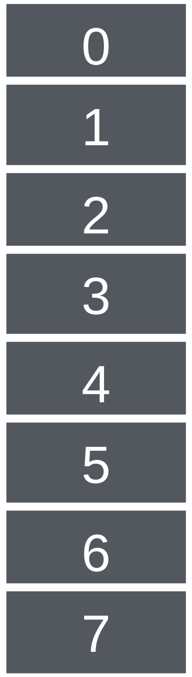
Ticket Lock Example

A lock():
B lock():
C lock():
A unlock():
B runs
A lock():
B unlock():
C runs
C unlock():
A runs
A unlock():
C lock():



Ticket Lock Example

A lock(): gets ticket 0, spins until turn = 0 →runs
B lock(): gets ticket 1, spins until turn=1
C lock(): gets ticket 2, spins until turn=2
A unlock(): turn++ (turn = 1)
B runs
A lock(): gets ticket 3, spins until turn=3
B unlock(): turn++ (turn = 2)
C runs
C unlock(): turn++ (turn = 3)
A runs
A unlock(): turn++ (turn = 4)
C lock(): gets ticket 4, runs



Ticket Lock Implementation

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

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

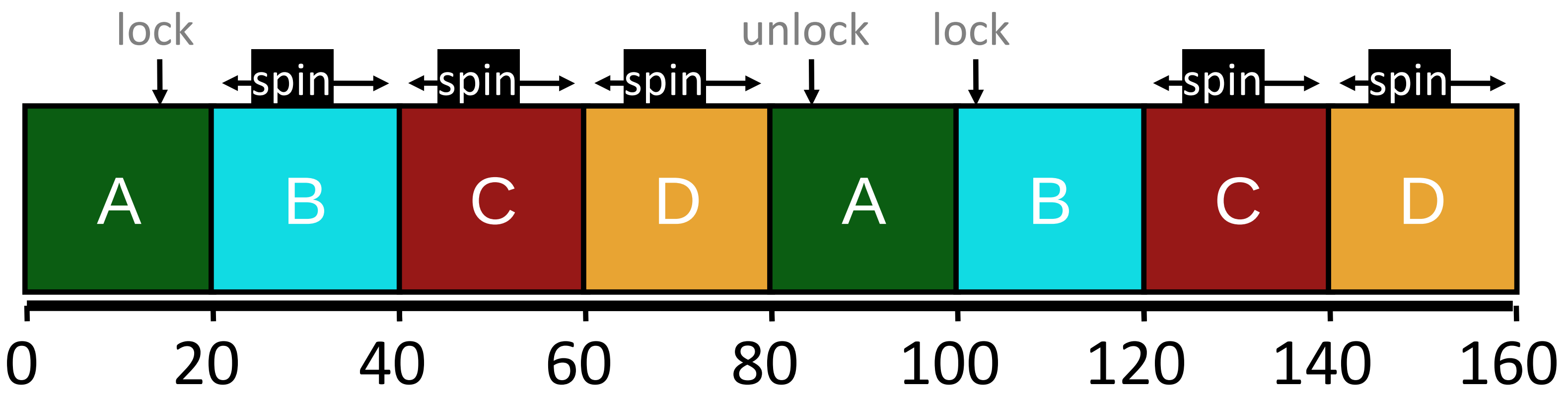
```
void acquire(lock_t *lock) {
        int myturn = FAA(&lock->ticket);
        while (lock->turn != myturn); // spin
}

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

Spinlock Performance

- **Fast when…**
  - many CPUs
  - locks held a short time
  - advantage: avoid context switch

- **Slow when…**
  - one CPU
  - locks held a long time
  - disadvantage: spinning is wasteful

CPU Scheduler is Ignorant



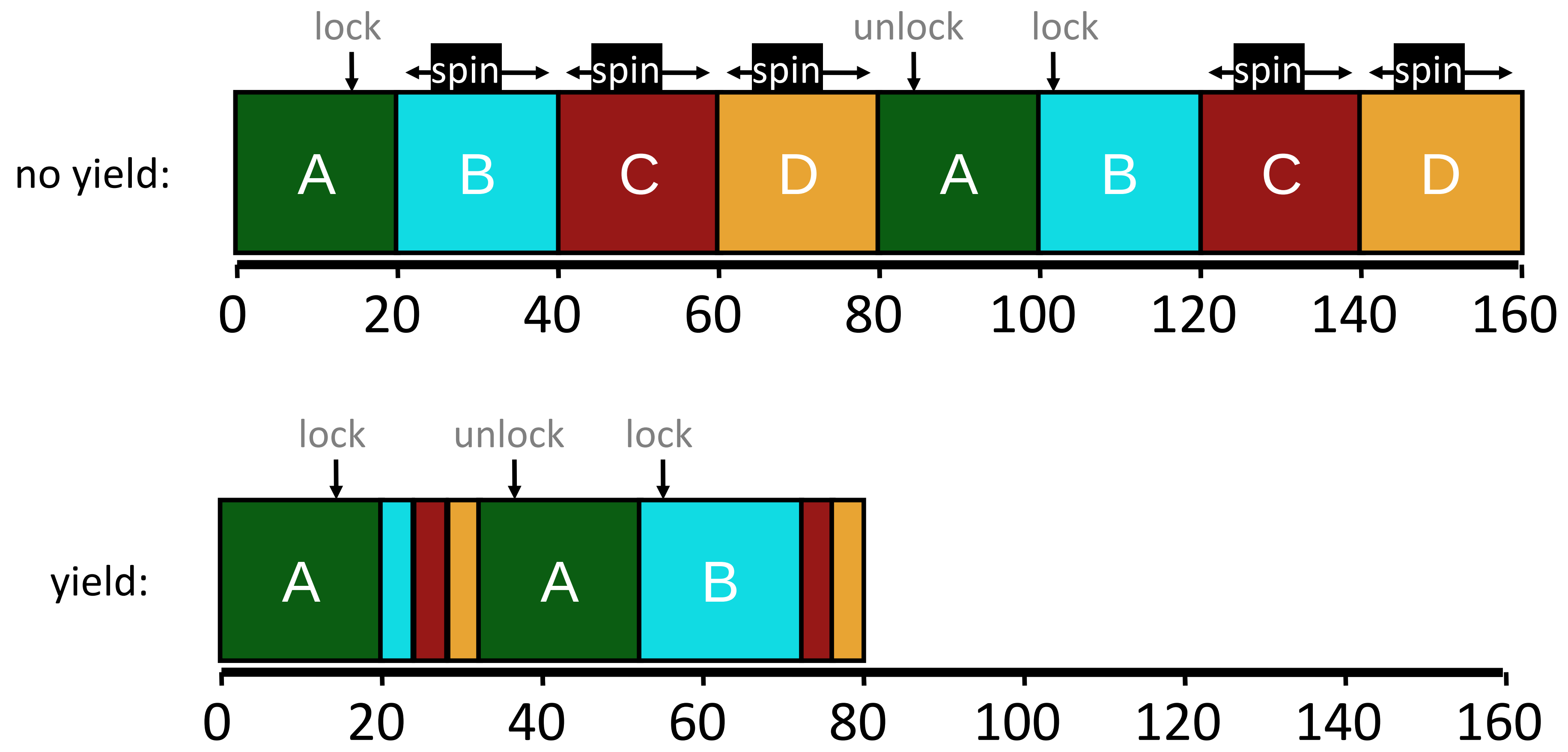
CPU scheduler may run **B** instead of **A**
even though **B** is waiting for **A**

Ticket Lock with Yield()

```
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();
}

void release (lock_t *lock) {
        FAA(&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, spinning is slow with high thread contention**

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

Lock Evaluation

- **How to tell if a lock implementation is good?**

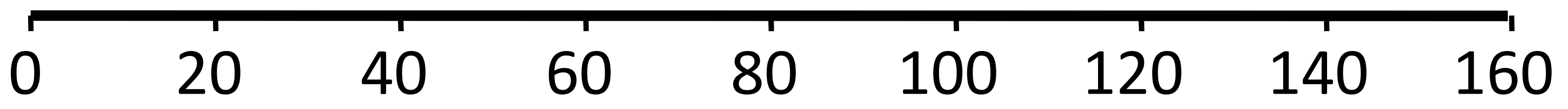
- **Fairness**
  - Do processes acquire lock in same order as requested?

- **Performance**
  - low contention (fewer threads, lock usually available)
  - high contention (many threads per CPU, each contending)

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

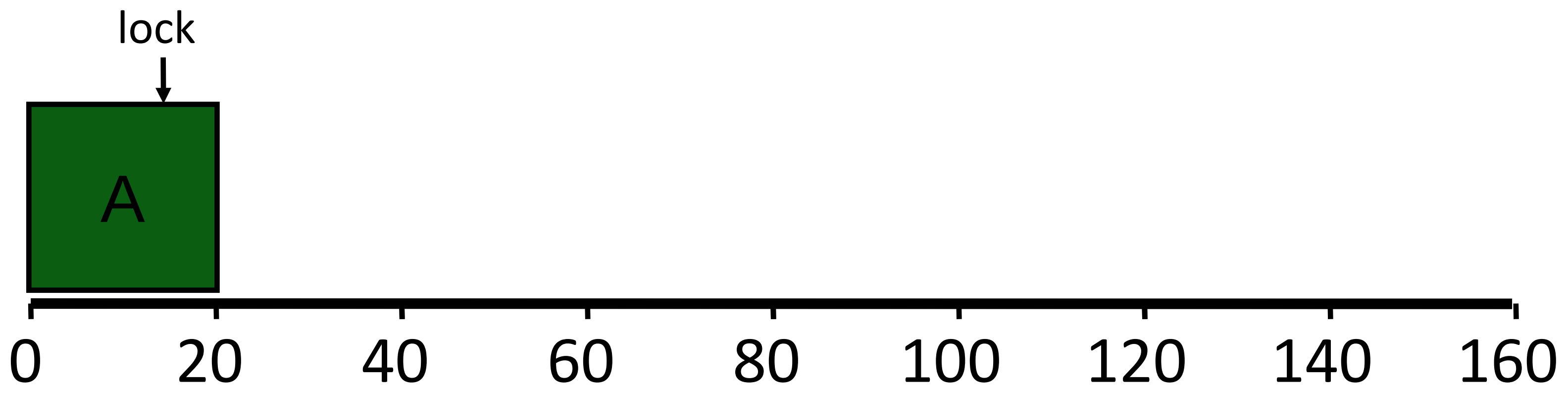
RUNNABLE:	A, B, C, D
RUNNING:	<empty>
WAITING:	<empty>



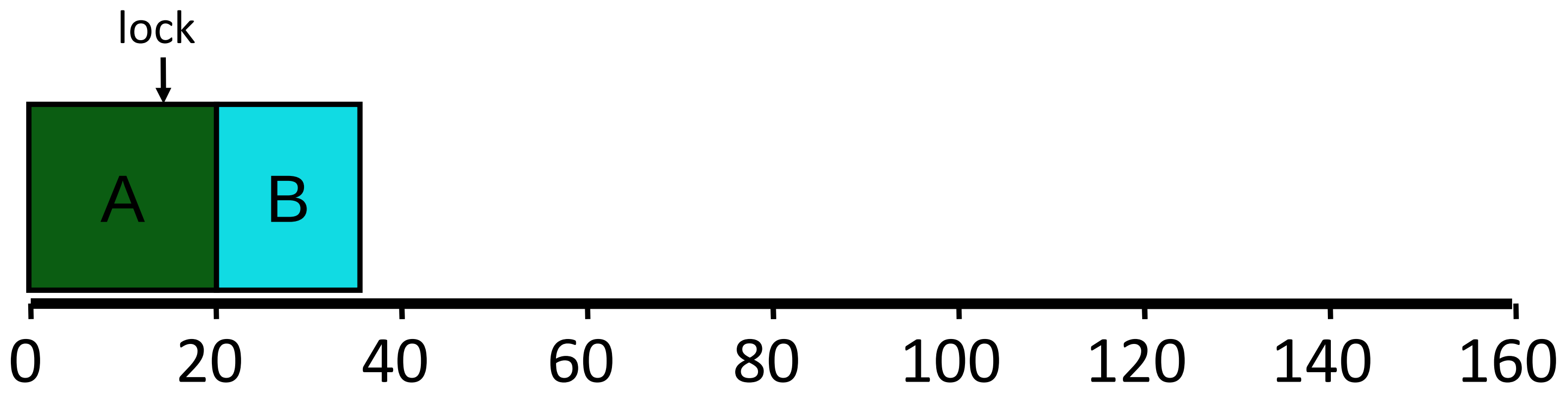
RUNNABLE: B, C, D

RUNNING: A

WAITING: <empty>



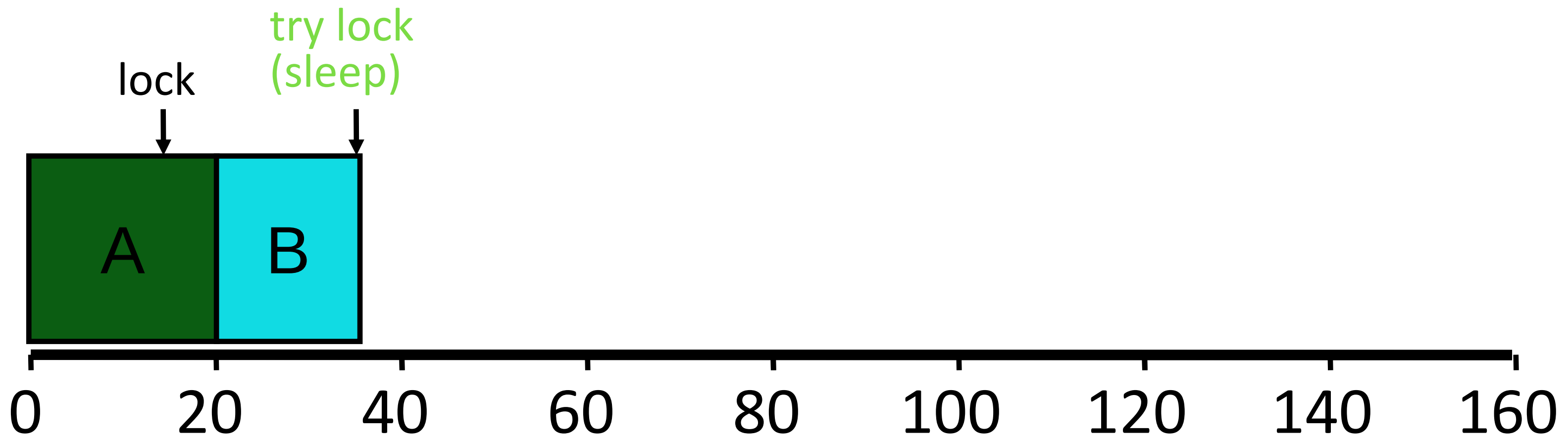
RUNNABLE:	C, D, A
RUNNING:	B
WAITING:	<empty>



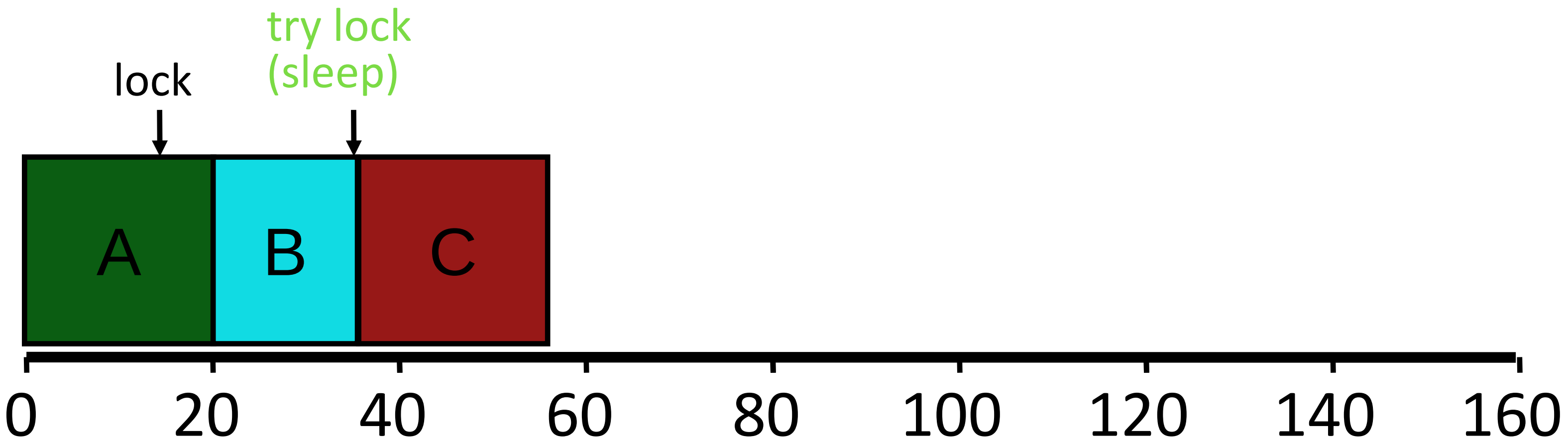
RUNNABLE: C, D, A

RUNNING:

WAITING: B



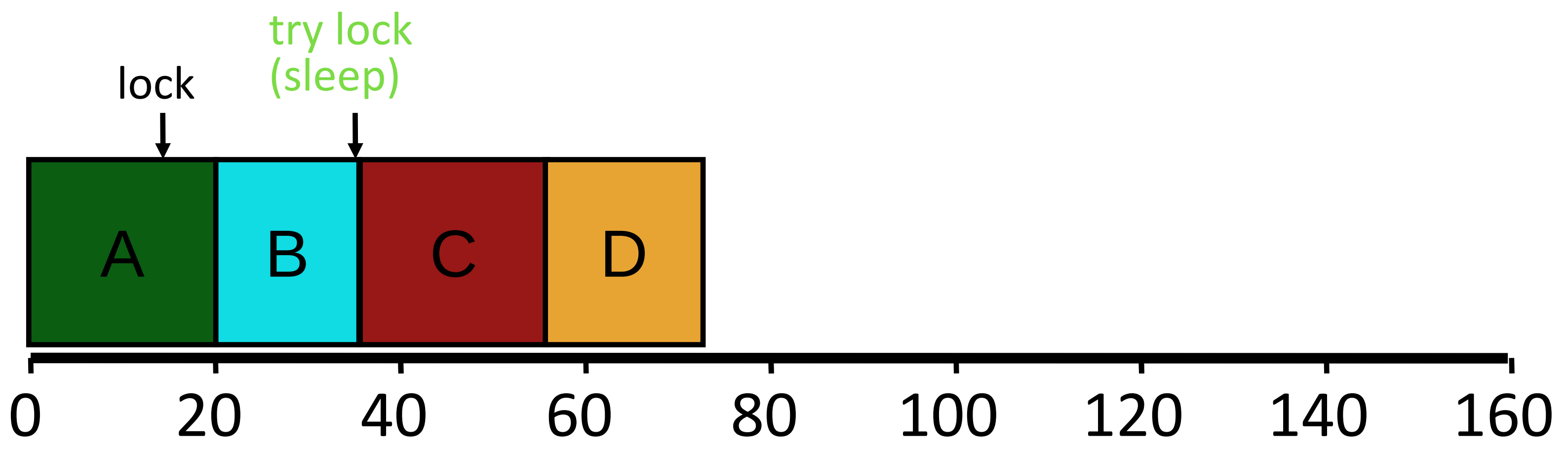
RUNNABLE:	D, A
RUNNING:	C
WAITING:	B



RUNNABLE: A, C

RUNNING: D

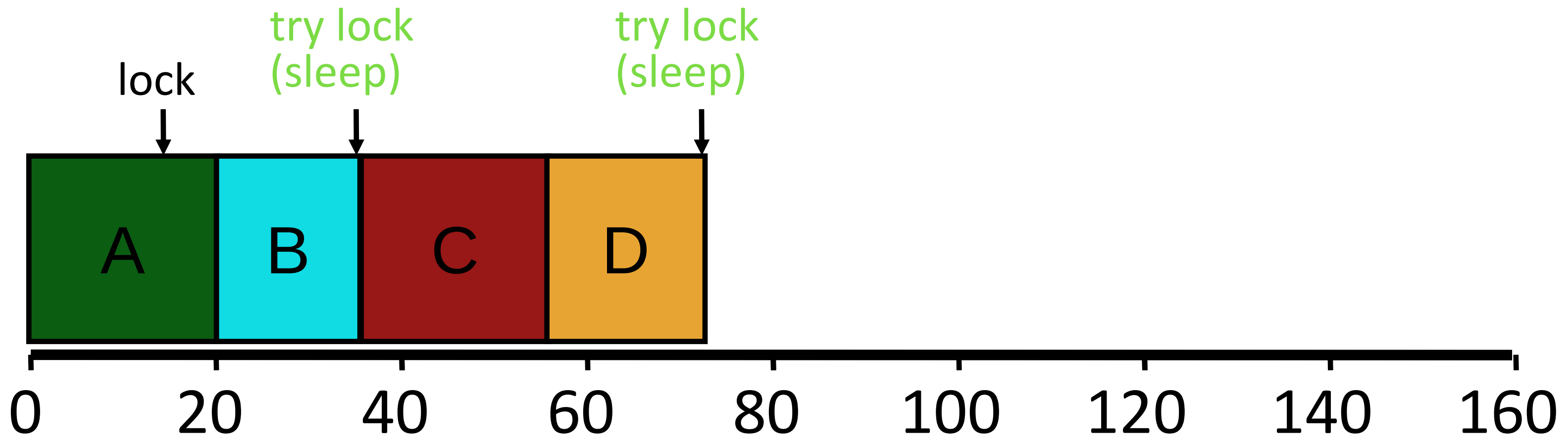
WAITING: B



RUNNABLE: A, C

RUNNING:

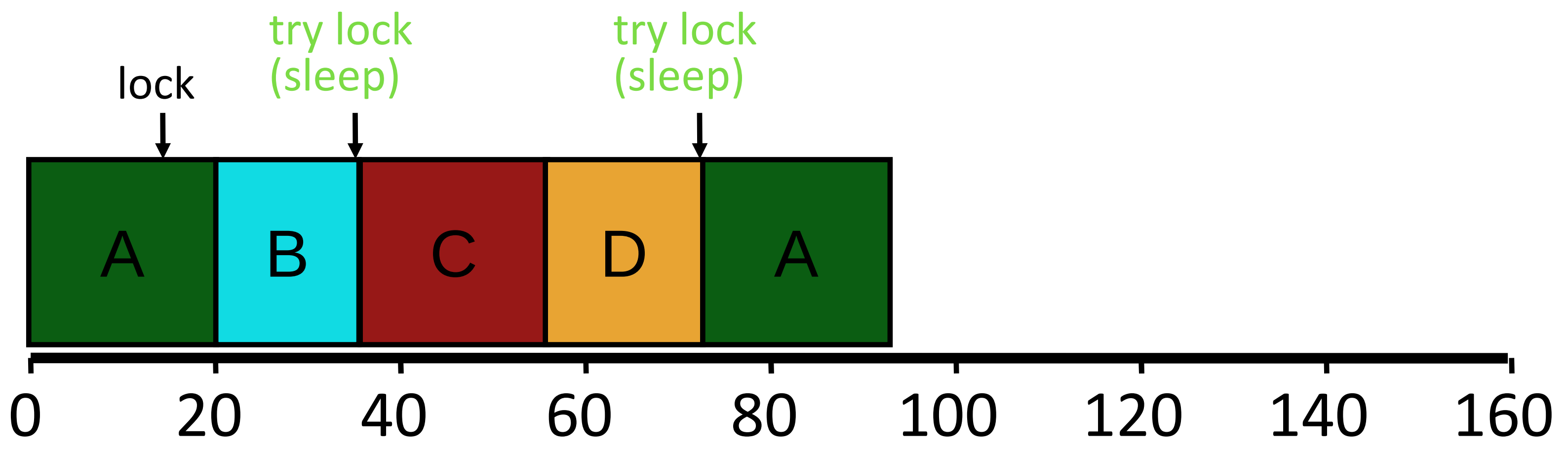
WAITING: B, D



RUNNABLE: C

RUNNING: A

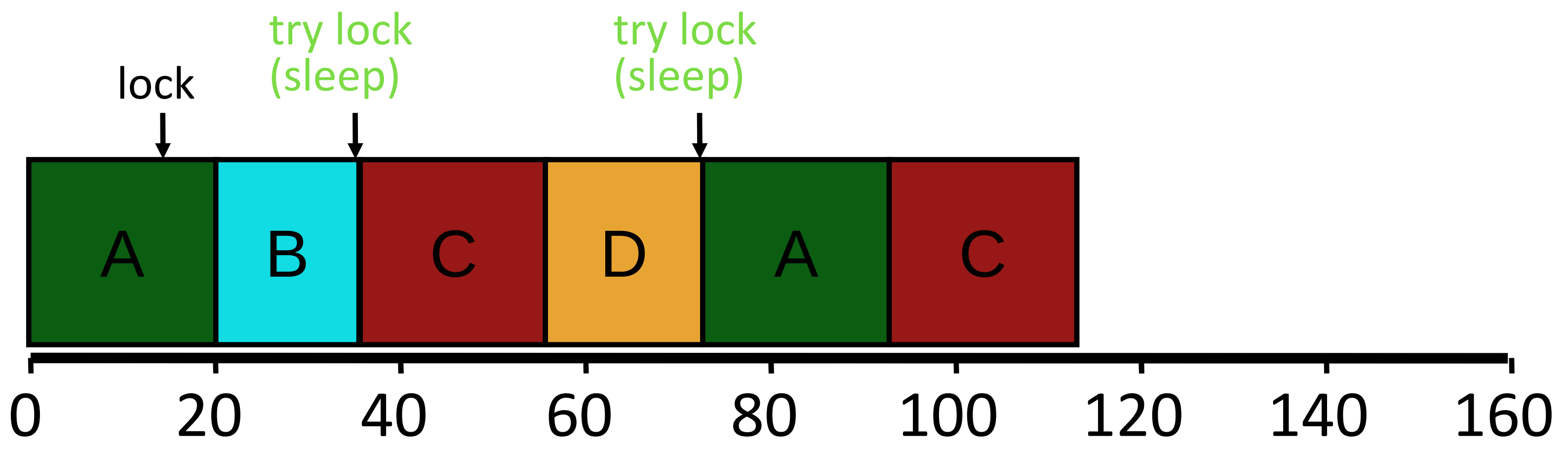
WAITING: B, D



RUNNABLE: A

RUNNING: C

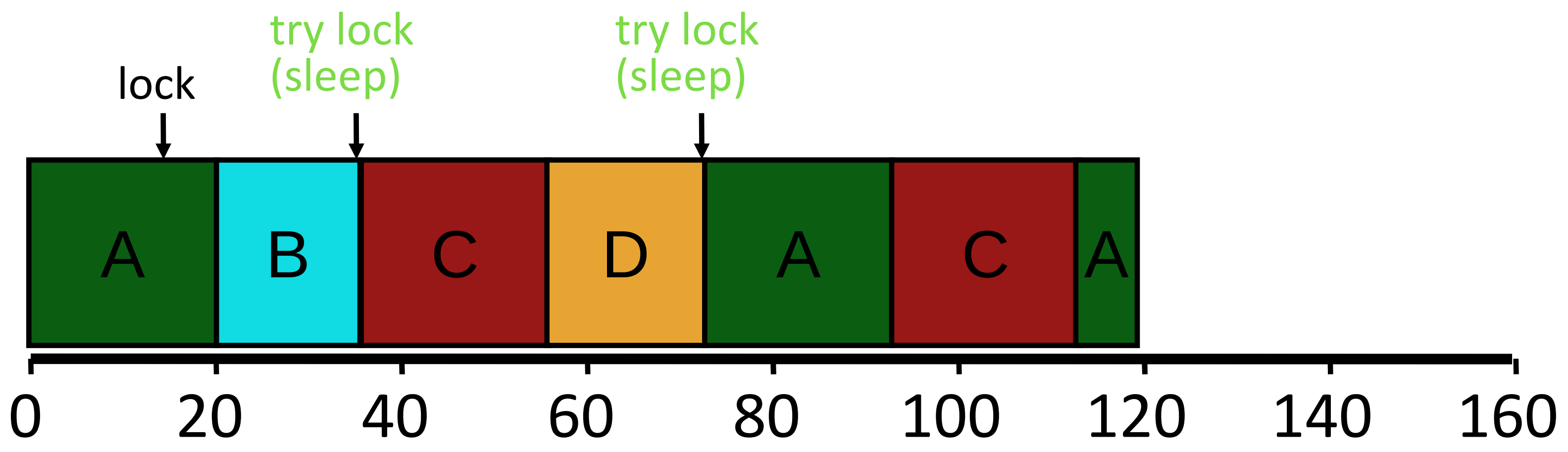
WAITING: B, D



RUNNABLE: C

RUNNING: A

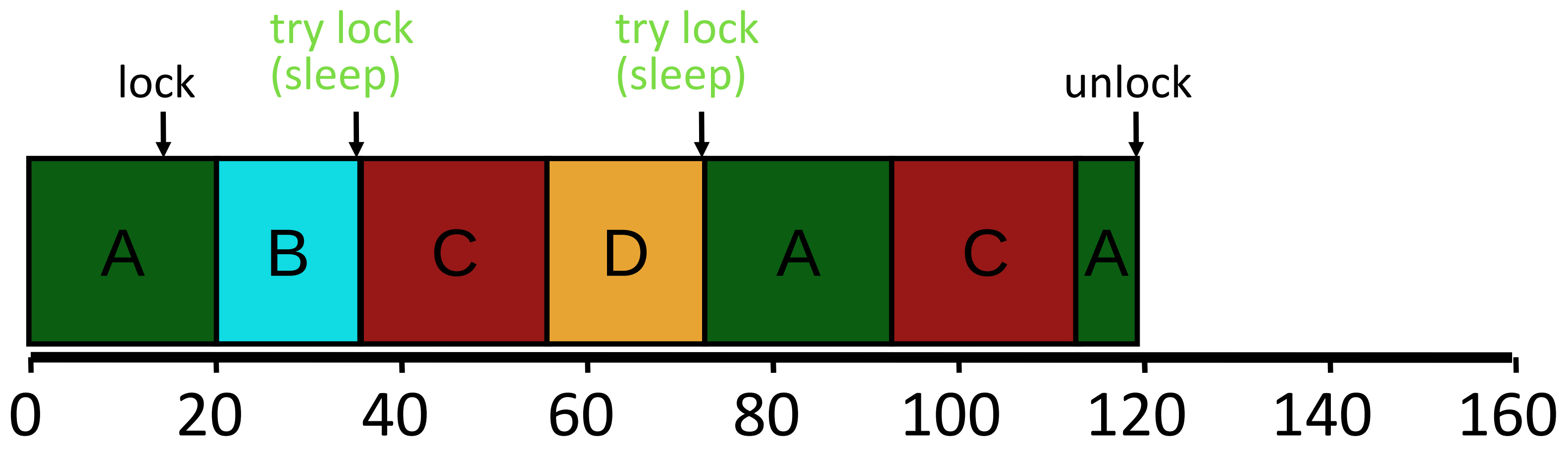
WAITING: B, D



RUNNABLE: B, C

RUNNING: A

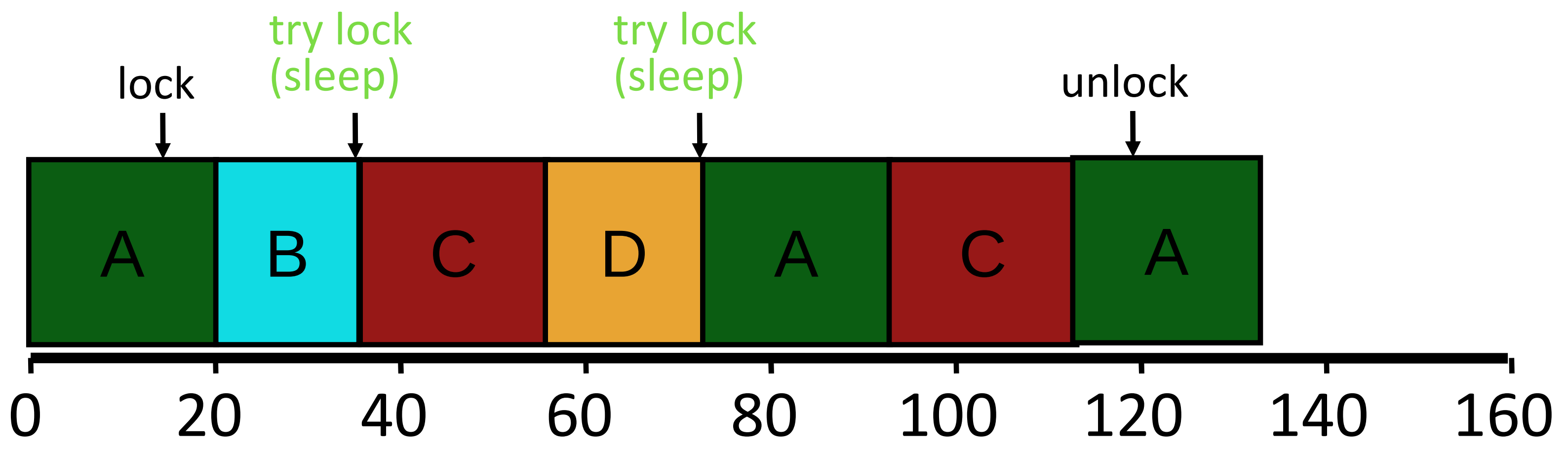
WAITING: D



RUNNABLE: B, C

RUNNING: A

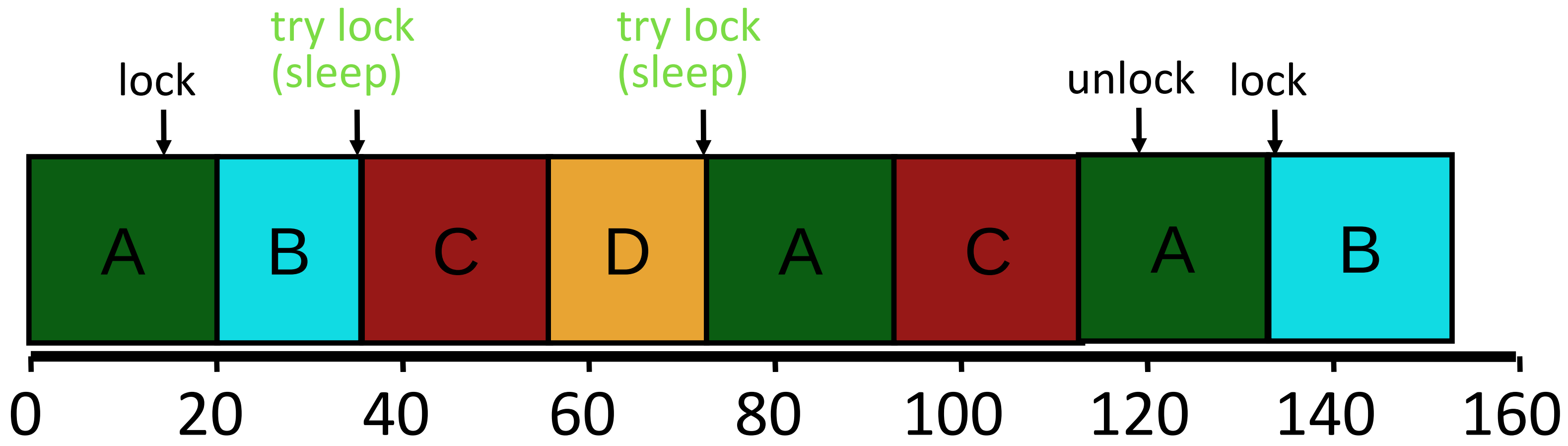
WAITING: D



RUNNABLE: C, A

RUNNING: B

WAITING: D



Lock Implementation: Block when Waiting

TAS is an atomic operation that returns the old value of guard and sets it to 1.

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

```
void acquire(LockT *l) {
    while (TAS(&l->guard, true));
    if (l->lock) {
        queue_add(l->q, tid);
       // add a thread into queue
        l->guard = false;
        park();     // sleep
    } else { // get lock
        l->lock = true;
        l->guard = false;
    }
}
```

(a) Why is guard used? protect queue
(b) Why okay to spin on guard?
(c) In release(), why not set lock=false when unpark? pass lock=true to acquire
(d) What is the race condition?

```
void release(LockT *l) { // scheduler
    while (TAS(&l->guard, true));
    if (qempty(l->q)) l->lock=false;
    else unpark(queue_remove(l->q));
        // runs a thread
    l->guard = false;
}
```

Race Condition

```
Thread 1      (in lock)                    Thread 2            (in unlock)
if (l->lock) {
        qadd(l->q, tid);
        l->guard = false;

                                           while (TAS(&l->guard, true));

                                           if (qempty(l->q)) // false!!

                                           else unpark(qremove(l->q));

                                           l->guard = false;

        park();    // block
```

Problem: Guard not held when call park(), put guard=false after park()?
Unlocking thread may unpark() before other park()

Block when Waiting: Final Correct Lock

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

setpark() in Solaris fixes race condition: if another thread calls unpark before park, the subsequent park returns immediately instead of sleeping

```
void acquire(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 release(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

When to Spin-Wait? When to Block?

- If know how long, t, before lock released, can determine optimal behavior
- How much CPU time is wasted when spin-waiting?

  t

- How much wasted when block?

  C

- What is the best action when t<C?

  spin-wait

- When t>C?

  block

- Problem:
  Requires knowledge of future; too much overhead to do any special prediction

Two-Phase Waiting

- **Theory: Bound worst-case performance; ratio of actual/optimal**
- **When does worst-possible performance occur?**

Spin for very long time t >> C
Ratio: t/C (unbounded)

- **Algorithm: Spin-wait for C, then block --> Factor of 2 of optimal**
- **Two cases:**
  - t < C: optimal spin-waits for t; we spin-wait t too
  - t > C: optimal blocks immediately (cost of C); we pay spin C then block (cost of 2 C); 2C / C -> 2-competitive algorithm
- **Example of competitive analysis**