CONCURRENCY: LOCKS

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

- Access slides and notes at ~arebello instead of ~shivaram

https://pages.cs.wisc.edu/~shivaram/cs537-sp23/

↓

https://pages.cs.wisc.edu/~arebello/cs537-sp23/

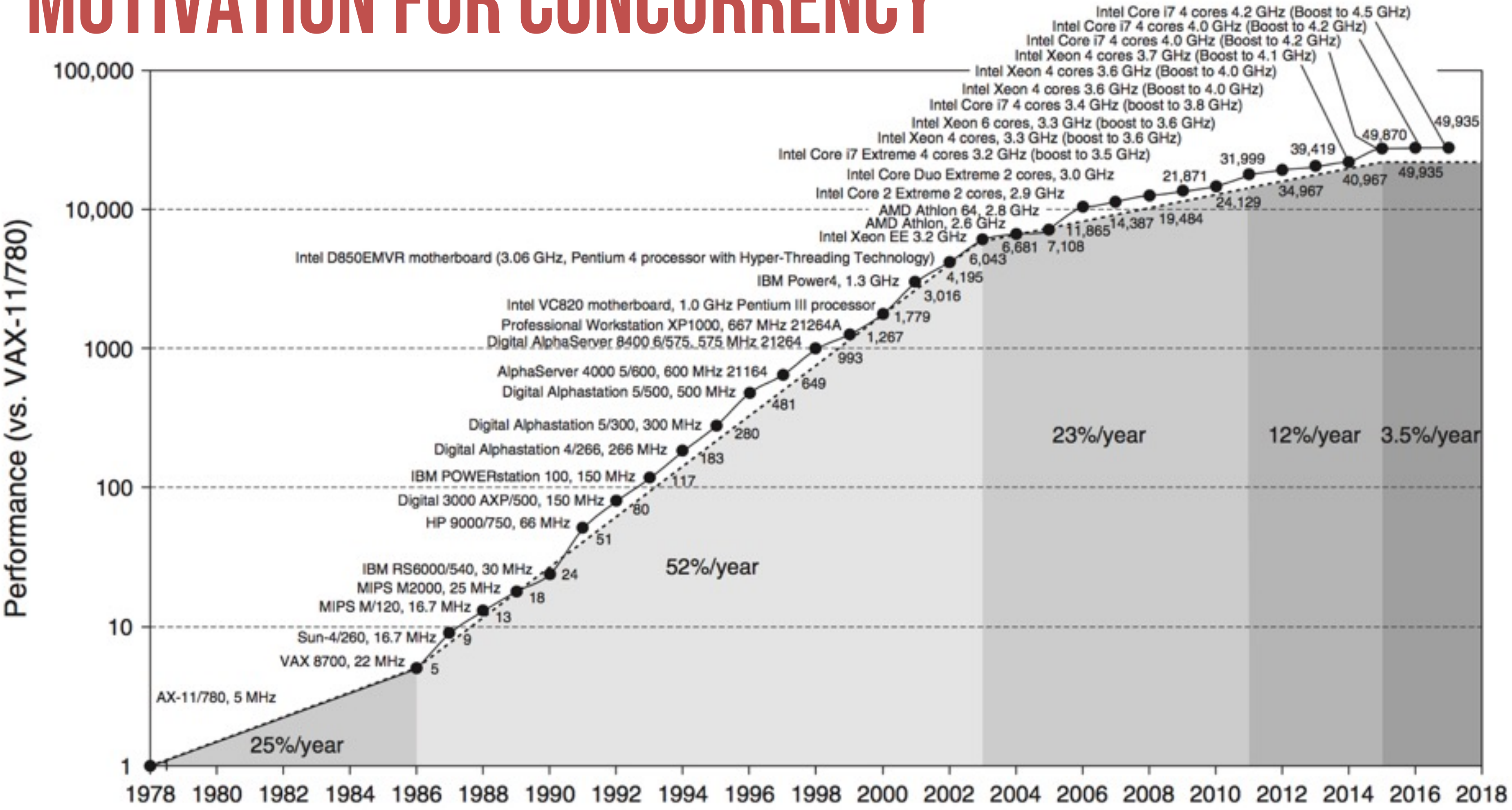
- Piazza and TAs for everything else

AGENDA / LEARNING OUTCOMES

- Concurrency
  - What are some of the challenges in concurrent execution?
  - How do we design locks to address this?

RECAP

MOTIVATION FOR CONCURRENCY



TIMELINE VIEW

x = x+1

10

Thread 1

mov 0x123, %eax

10

ctx switch

add %0x1, %eax 10+1

mov %eax, 0x123

11

x = x+2

Thread 2

mov 0x123, %eax 10

add %0x2, %eax 12

mov %eax, 0x123

12

WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group

That is, we want them to be atomic

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

More general: Need mutual exclusion for critical sections

if thread A is in critical section C, thread B isn't

(okay if other threads do unrelated work)

LOCK IMPLEMENTATION GOALS

API

lock() acquire()

unlock() release()

T1 T2 T3

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

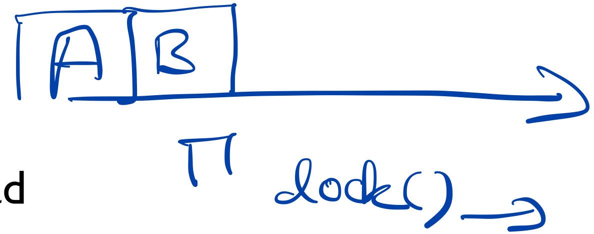
IMPLEMENTING SYNCHRONIZATION

Atomic operation: No other instructions can be interleaved

Approaches

- Disable interrupts
- Locks using loads/stores
- Using special hardware instructions

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?

Only works on uniprocessors

Process can keep control of CPU for arbitrary length

Cannot perform other necessary work

IMPLEMENTING LOCKS: W/ LOAD+STORE

Code uses a single **shared** lock variable

T1

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

T2 acquire()

```
void release(Boolean *lock) {
    *lock = false;
}
```

T1

Does this work? What situation can cause this to not work?

RACE CONDITION WITH LOAD AND STORE

```
*lock == 0 initially
```

acquire()

Thread 1	Thread 2
`while(*lock == 1)`	
lock is 0	`while(*lock == 1)` lock is 0
	`*lock = 1`
`*lock = 1`	

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

XCHG: ATOMIC EXCHANGE OR TEST-AND-SET (TAS)

A [100]

xchg(&A, 20)

How do we solve this ? Get help from the hardware!

```
// 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;
}
```

single atomic instruction

```
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```

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 = ??;
}
```

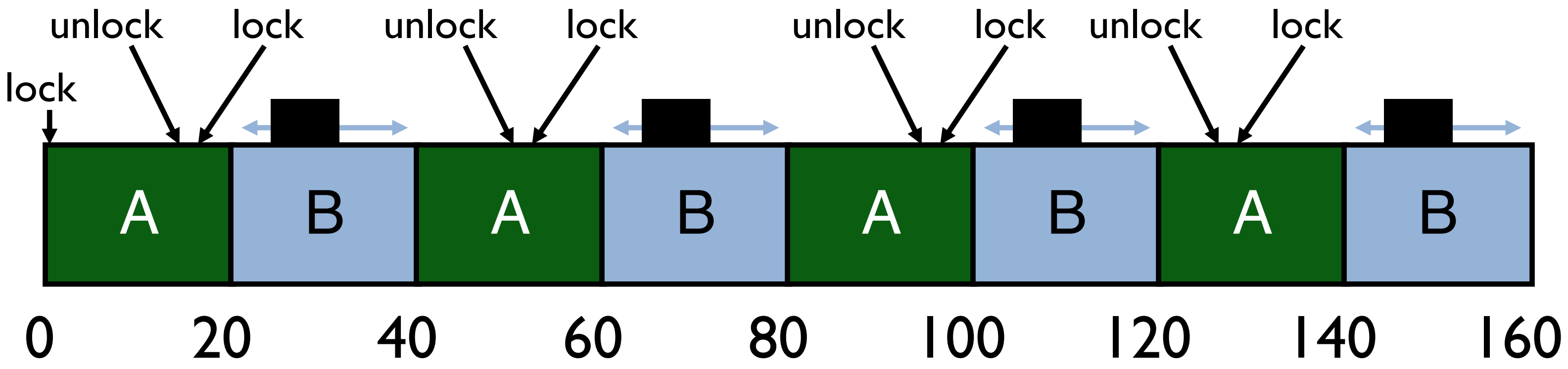
```
int xchg(int *addr, int newval)
```

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)
}
```

BASIC SPINLOCKS ARE UNFAIR



Scheduler is unaware of locks/unlocks!

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

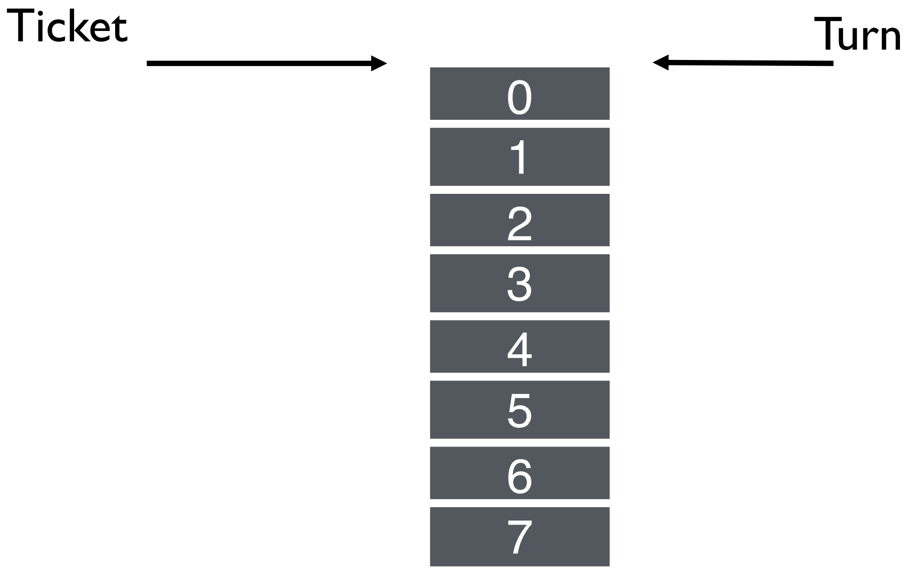
Ticket Lock Example

A lock():
B lock():
C lock():

A unlock():

A lock():
B unlock():

C unlock():
A unlock():



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);
    // spin
    while (lock->turn != myturn);
}

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

Spinlock Performance

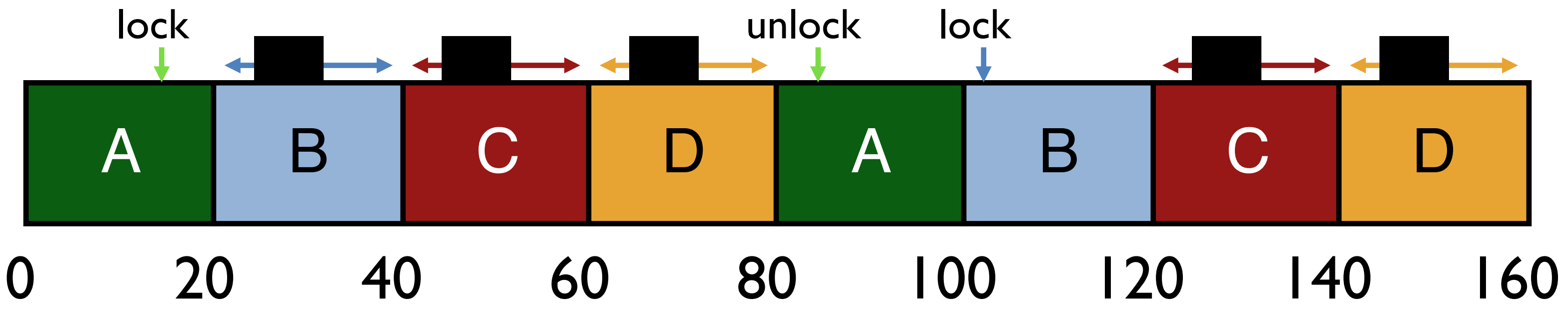
Fast when…

- 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, C, D** instead of **A**
even though **B, C, D** are waiting for **A**

TICKET LOCK WITH YIELD

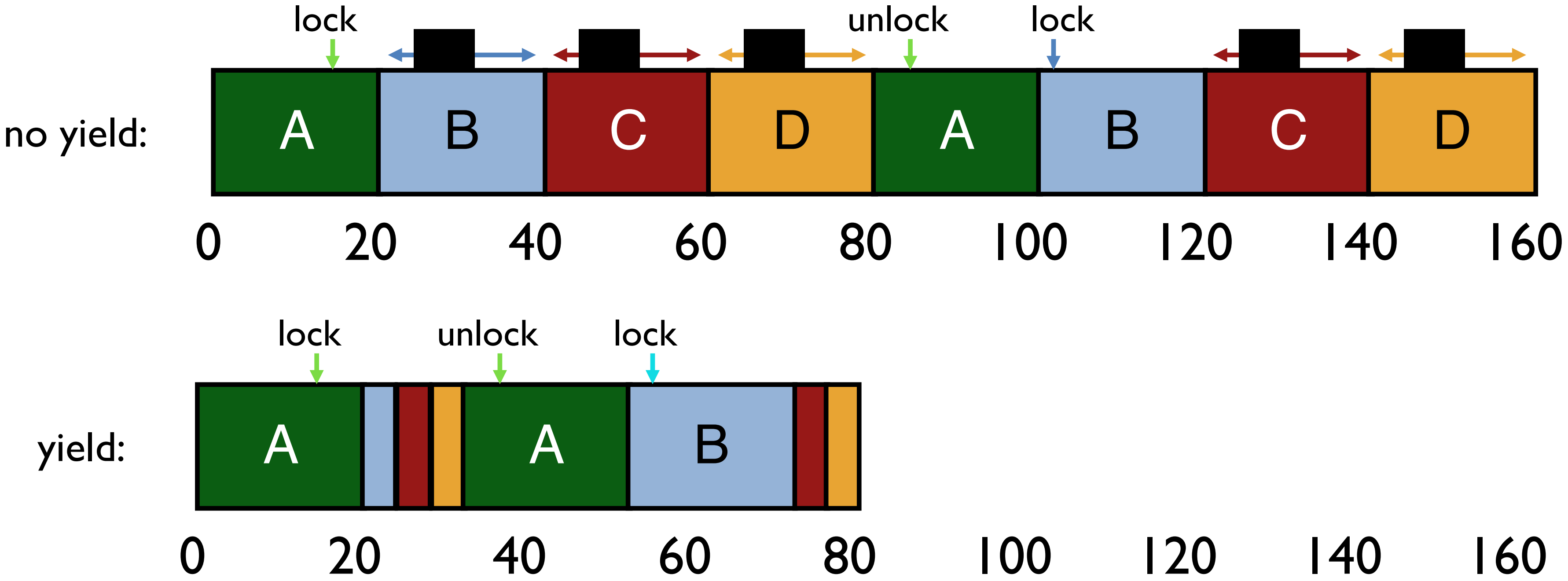
```
typedef struct __lock_t {
    int ticket;
    int turn;
}

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



QUIZ 16

https://tinyurl.com/cs537-sp23-quiz15

```
a = 1
int b = xchg(&a, 2)
int c = CAS(&b, 2, 3)
int d = CAS(&b, 1, 3)
```

Final values

Assuming round-robin scheduling, 10ms time slice. Processes A, B, C, D, E, F, G, H in the system

Timeline
A: lock() … compute … unlock()
B: lock() … compute … unlock()
C: lock()

SPINLOCK PERFORMANCE

Waste of CPU cycles?

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

Even with yield, spinning is slow with high thread contention

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

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

Remove waiting threads from scheduler runnable queue
(e.g., park() and unpark(threadID))

Scheduler runs any thread that is **runnable**

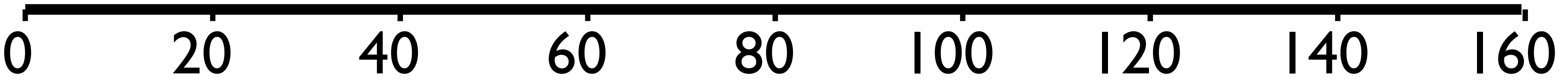
A B D contend for lock, C is not contending

A has 60 ms worth of work
20ms is the timeslice

RUNNABLE: A, B, C, D

RUNNING:

WAITING:



LOCK IMPLEMENTATION: BLOCK WHEN WAITING

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

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

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

LOCK IMPLEMENTATION: BLOCK WHEN WAITING

(a) Why is **guard** used?

(b) Why okay to **spin** on guard?

(c) In release(), why not set lock=false when unpark?

(d) Is there a race condition?

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

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

RACE CONDITION

Thread 1 (in lock)

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

    park();    // block
```

Thread 2 (in unlock)

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

BLOCK WHEN WAITING: FINAL CORRECT LOCK

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

setpark + unpark

park() ← ignore

setpark() fixes race condition

```
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 blocking?

What is the best action when t<C?

When t>C?

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

NEXT STEPS

Midterm on Thursday 3/2

No class on Thursday

Next Tuesday: Condition Variables