

# 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

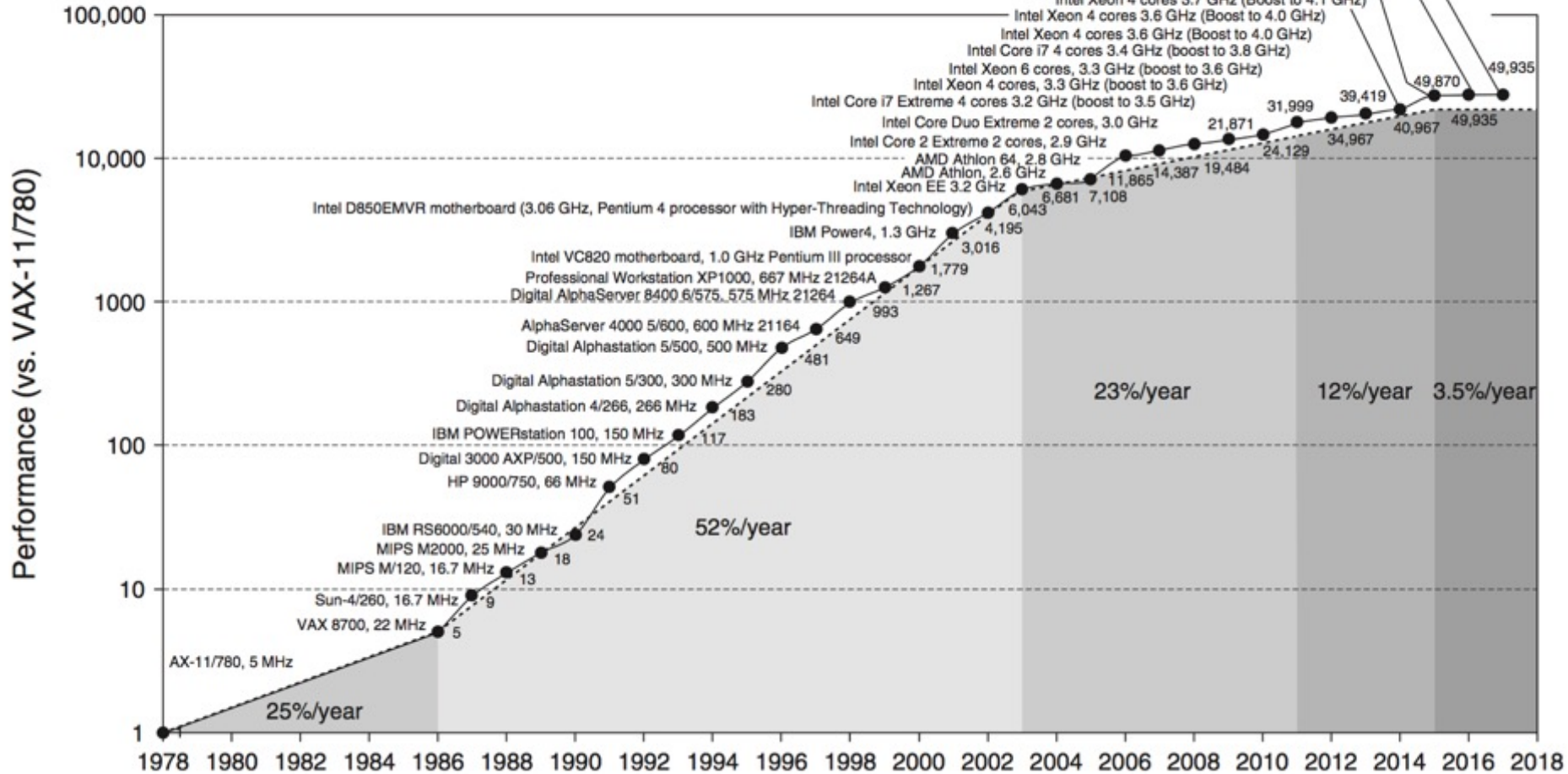
# AGENDA / LEARNING OUTCOMES

## Concurrency

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

**RECAP**

# MOTIVATION FOR CONCURRENCY



# TIMELINE VIEW

$x = x + 1$

**Thread 1**

mov 0x123, %eax

10

← CTX switch →

add %0x1, %eax 10 + 1

mov %eax, 0x123

11

10

$x = x + 2$

**Thread 2**

mov 0x123, %eax

add %0x2, %eax

mov %eax, 0x123

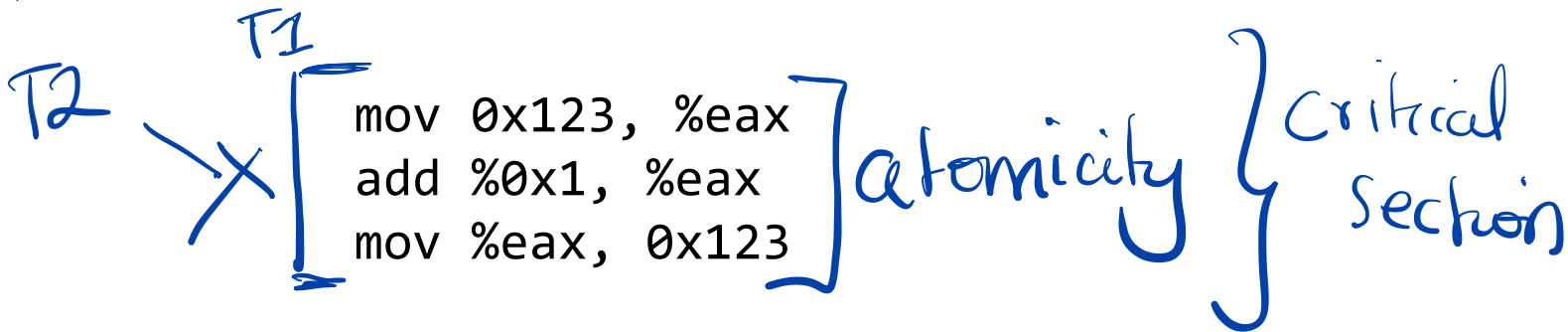
10  
12

12

# WHAT DO WE WANT?

Want 3 instructions to execute as an uninterruptable group


That is, we want them to be atomic



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

## 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

API

lock() acquire()  
unlock() release()

T1 T2 T3  
at least 1 gets the  
lock

All should eventually get  
the lock .

# IMPLEMENTING SYNCHRONIZATION

**Atomic operation:** No other instructions can be interleaved

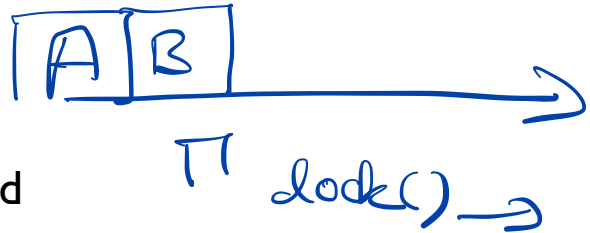
Approaches

- 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();  
}
```

→ Timer Interrupt doesn't run.  
Scheduler doesn't run.

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

\*lock = 1

while(\*lock == 1) lock is 0  
\*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

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

$xchg(\&A, 20)$   
100

20

```
// 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 = ??;
}
```

Handwritten notes and diagram:

Flag=0  
Flag=1  
Flag=0

T1 acquire()  
:  
release()

T2 acquire()

~~Unlocked~~ Unlocked = 0  
locked = 1

0 ———— Flag=1 ————> &lock->flag 1

int xchg(int \*addr, int newval)

if another holds the lock, keep trying.  
else take the lock

while(xchg(&lock->flag, 1) != 1);

→ ;

# OTHER ATOMIC HW INSTRUCTIONS

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

A 50

100

$50 \leftarrow \text{CAS}(\&A, 50, 100)$

$50 \leftarrow \text{CAS}(\&A, 60, 100)$  50

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

Lock it

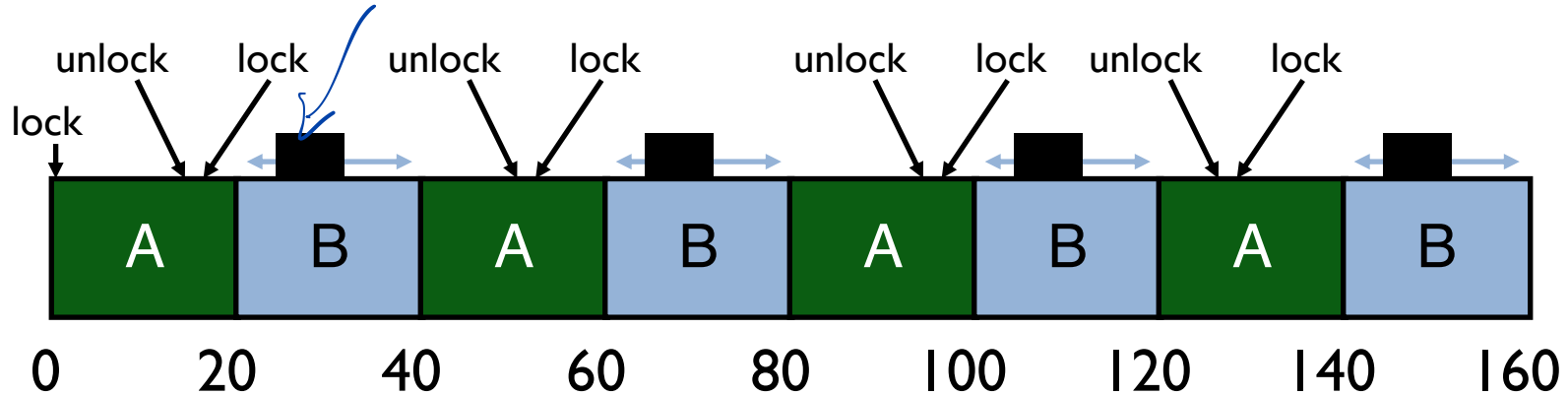
retry if already locked

we want to swap if unlocked

# BASIC SPINLOCKS ARE UNFAIR

*c1*      *c2*  
*A lock*      *unlock*  
*unlock*      *lock*

*acquire()*



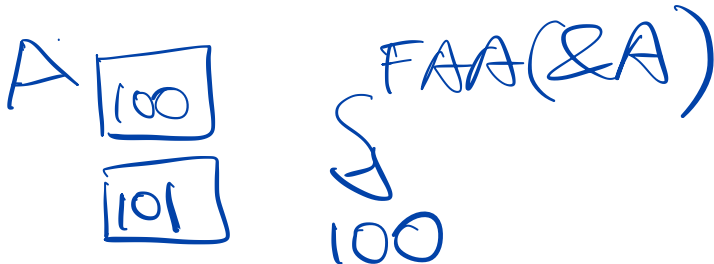
Scheduler is unaware of locks/unlocks!

# 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

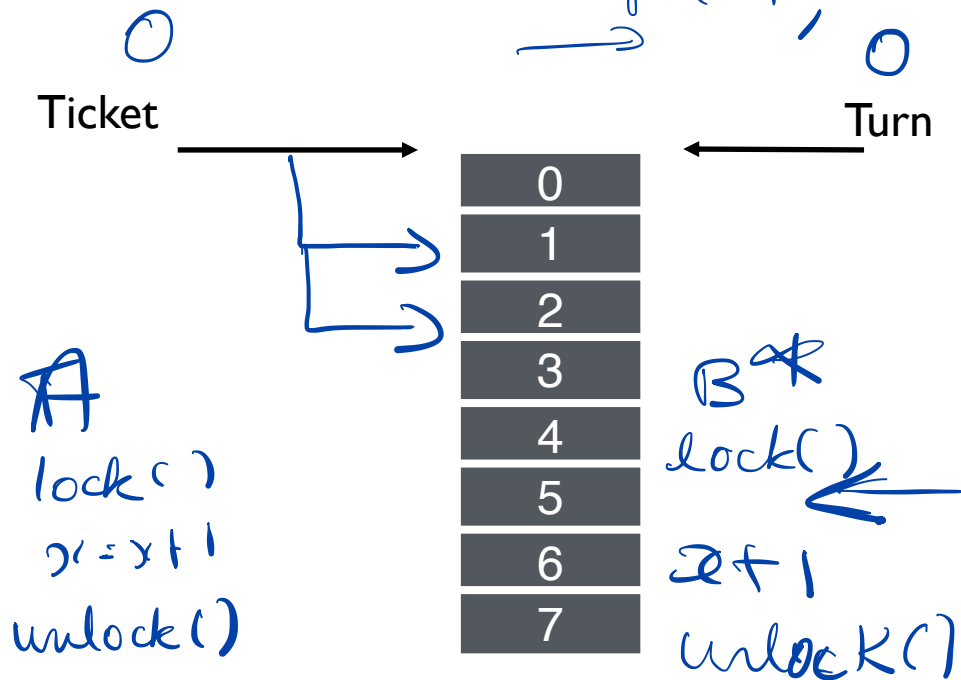
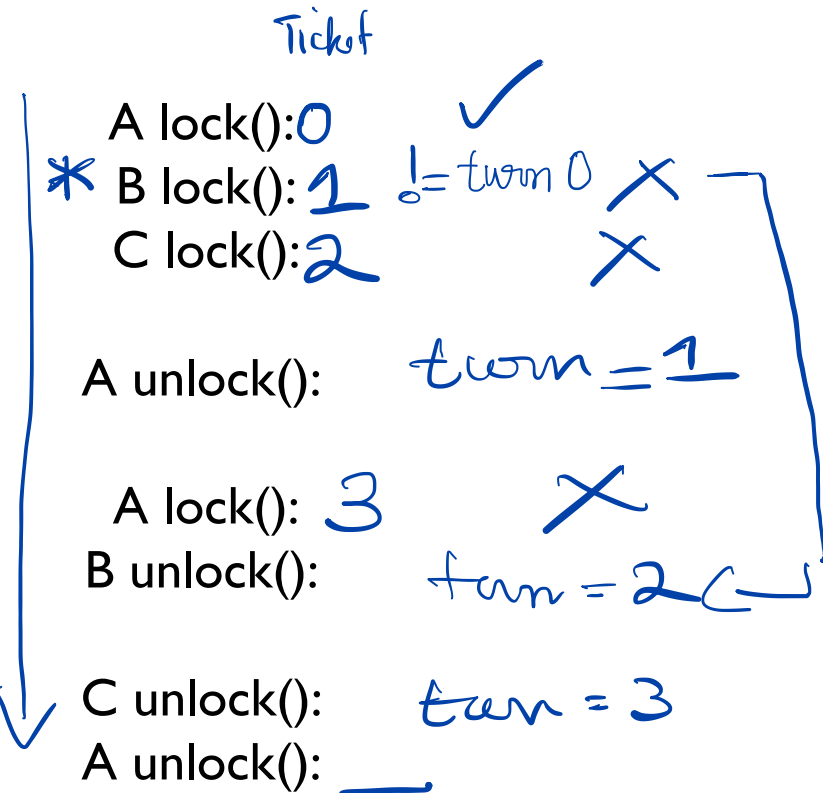
Release: Advance to next turn

RR

# TICKET LOCK EXAMPLE

```

    foo(int *ptr)
    →
    *ptr++;
    →
    0
  
```



# TICKET LOCK IMPLEMENTATION

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

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

*\_\_lock\_t L1, L2  
L1 acquire //  
x = x + 1  
L1*

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

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

*↑ No need for FAA  
Critical section held by  
lock->turn + 1 owner of  
lock*

# SPINLOCK PERFORMANCE

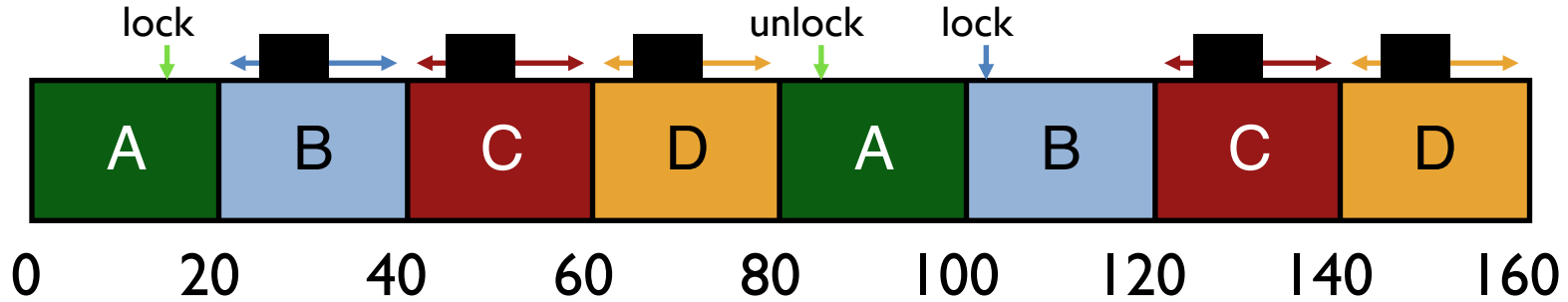
Fast when...

- 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;
}

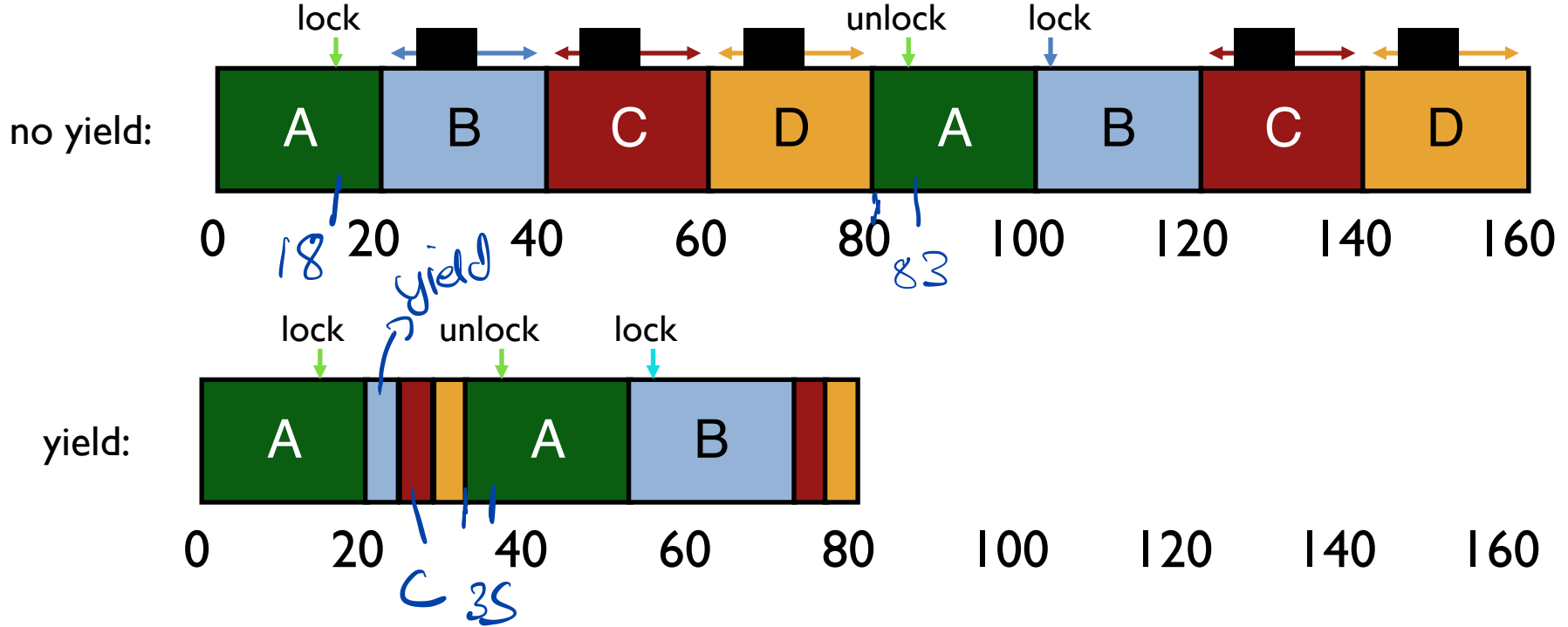
void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

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

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

*Instead of spinning*

# 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

$a=2$        $b=1$

$c=1$        $b=1$

$d=1$        $b=3$

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

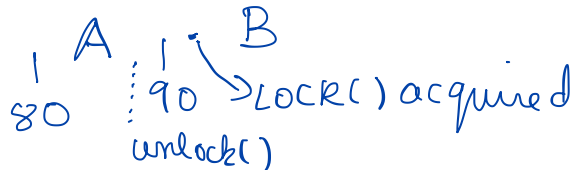
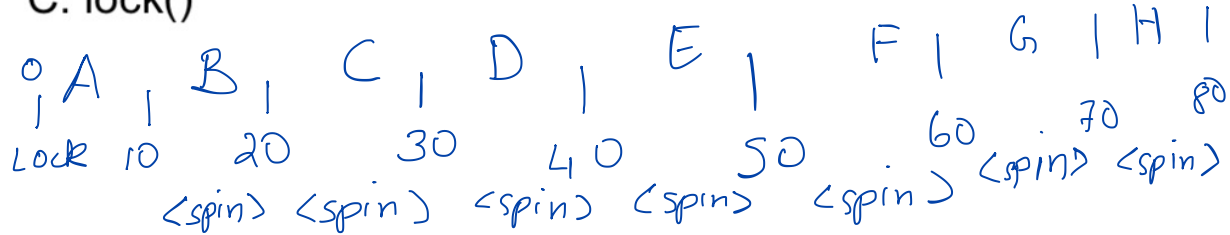
Timeline

A: lock() ... compute ... unlock()

B: lock() ... compute ... unlock()

C: lock()

thread exits after  
unlock



B acquires lock  
at 90ms

A	B	C	D
2	3	1	1

② B C D E F G H 160  
 90 100  
 160 B 170 . . . . . + 1230

C gets the lock at 240ms

230 B 1 240 C  
 unlock

③ yield() costs 1ms

A B C D E F G H  
 0 10 11 12 13 14 15 16 17

A B  
 17 27

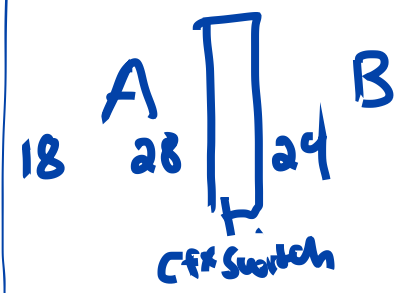
Barquies lock at 27ms

Context switch costs 1ms



context switch costs 1ms

B calls yield() which context switches 1ms + time to yield = ~1ms



Barquies lock at 29ms

# SPINLOCK PERFORMANCE

Waste of CPU cycles?

Without yield:  $O(\text{threads} * \text{time\_slice})$

With yield:  $O(\text{threads} * \text{context\_switch})$



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**

SOLARIS

SUN Microsystems

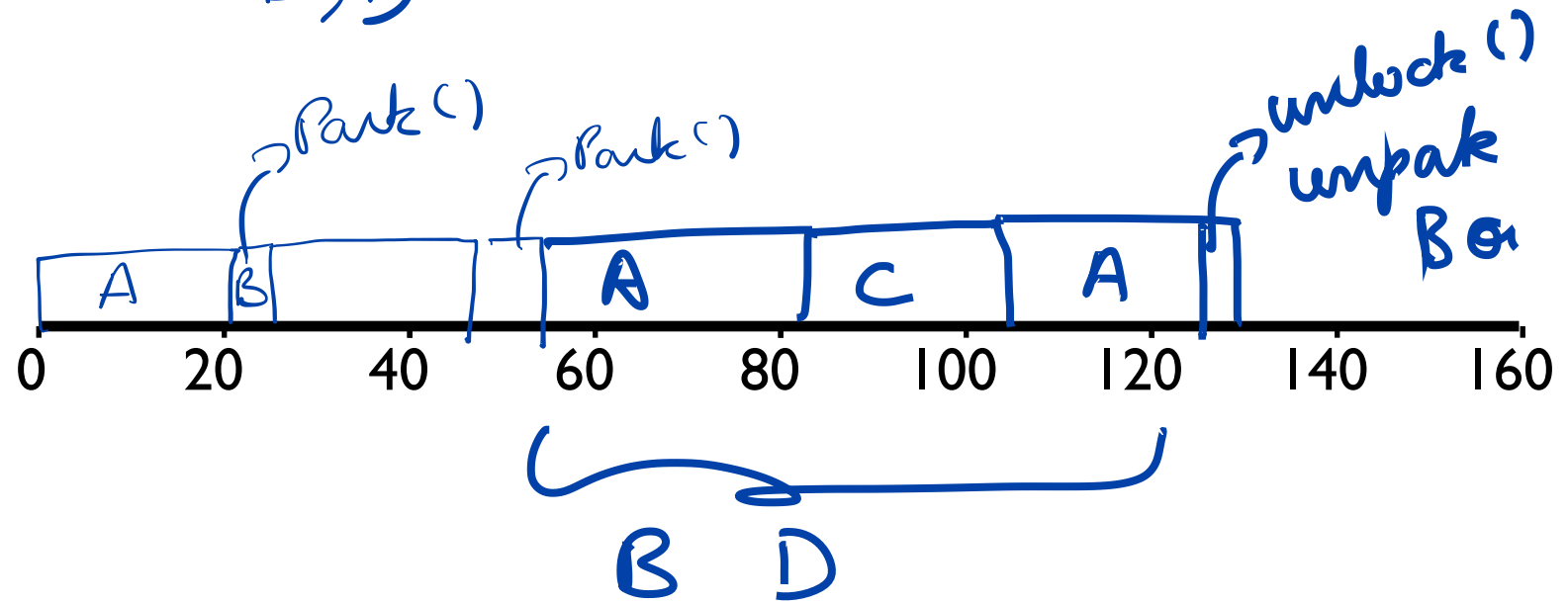
A B D contend for lock, C is not contending

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

RUNNABLE: A, ~~B~~, C, ~~D~~

RUNNING:

WAITING: B, D



# 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;
```

*T2 → acquire()*

*Q [T2]*

```
park(); // block
```

*↖ will never be unparked*

**Thread 2**

(in unlock)

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

*unpark(T2)*

*↑  
does not exist*

# 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  $t \gg C$

Ratio:  $t/C$  (unbounded)

Algorithm: Spin-wait for  $C$  then block  $\rightarrow$  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  $2C$ );

$2C / C \rightarrow 2$ -competitive algorithm

# NEXT STEPS

Midterm on Thursday 3/2

No class on Thursday

Next Tuesday: Condition Variables