

# CONCURRENCY: LOCKS

Shivaram Venkataraman

CS 537, Spring 2019

# ADMINISTRIVIA

- Project 2b is due **Wed Feb 27<sup>th</sup>, 11:59pm**
- Project 2a grades out by tonight

# AGENDA / LEARNING OUTCOMES

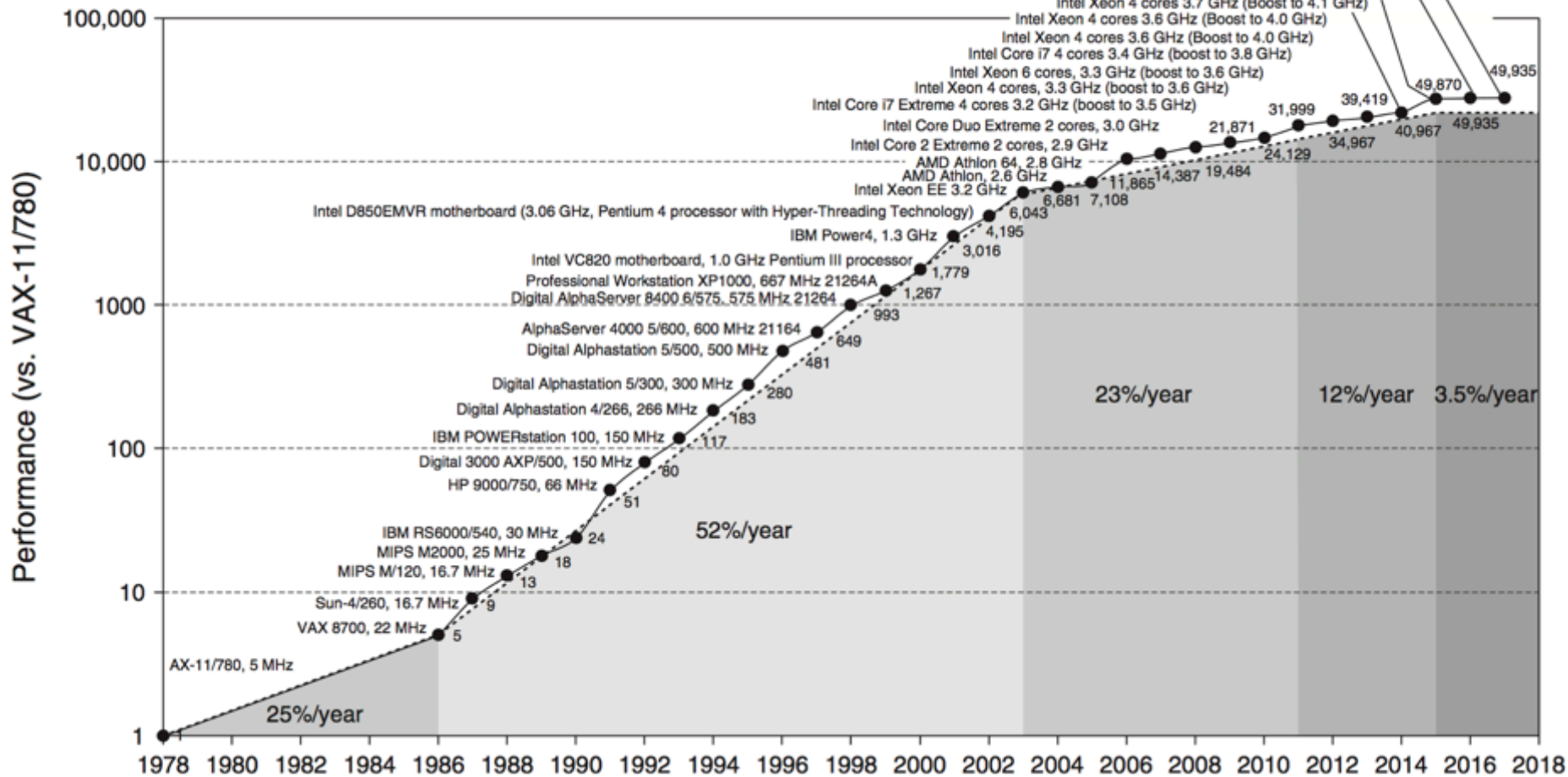
## Concurrency

What are some of the challenges in concurrent execution?

How do we design locks to address this?

**RECAP**

# MOTIVATION FOR CONCURRENCY



variable  
a = 0

# TIMELINE VIEW

## Thread 1

mov 0x123, %eax ✓

add %0x1, %eax ✓

mov %eax, 0x123 ✓

a = 1

## Thread 2

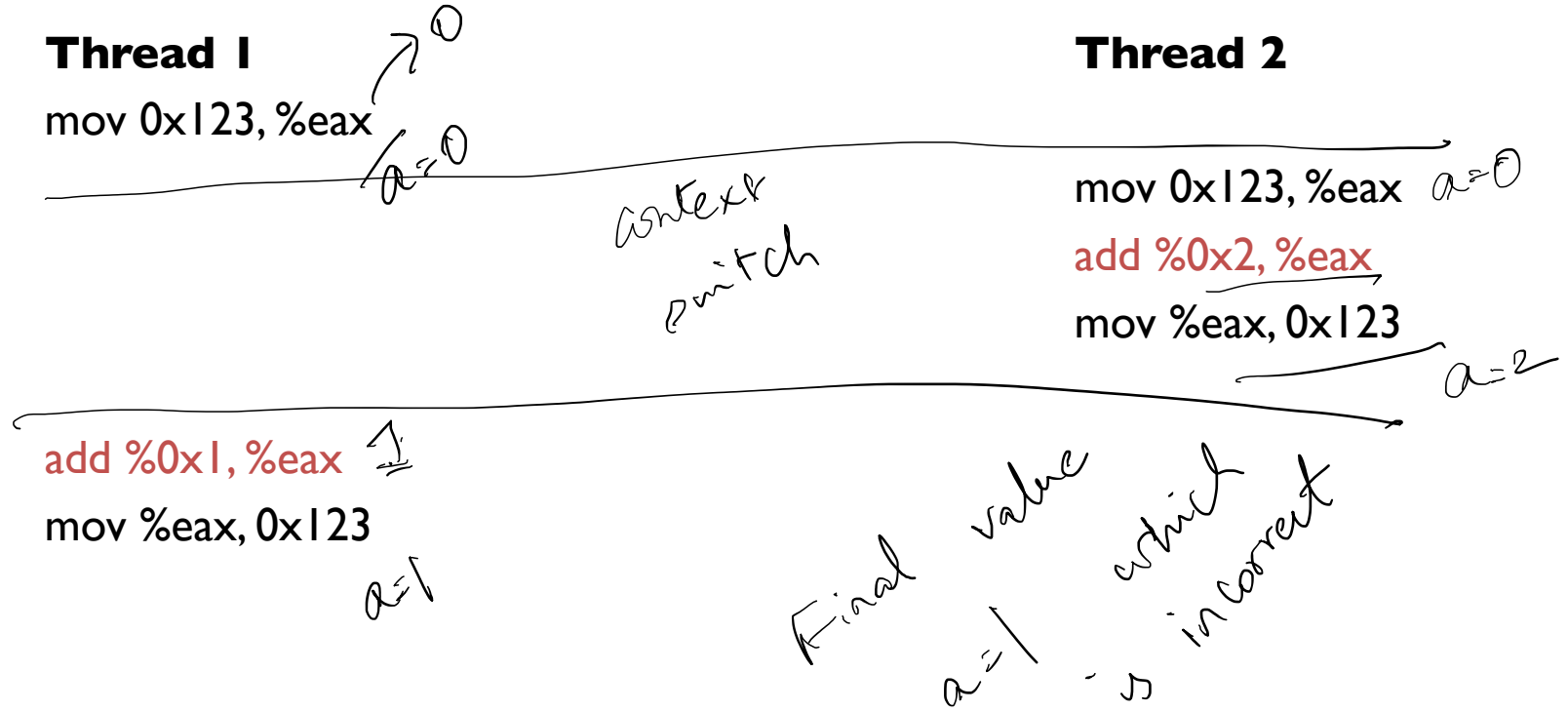
mov 0x123, %eax

add %0x2, %eax ✓

mov %eax, 0x123

a = 3

# TIMELINE VIEW



# NON-DETERMINISM

Concurrency leads to non-deterministic results

- Different results even with same inputs
- race conditions

Whether bug manifests depends on CPU schedule!

How to program: imagine scheduler is malicious?!



# 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
```

read  
increment  
write

"critical  
section"

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)

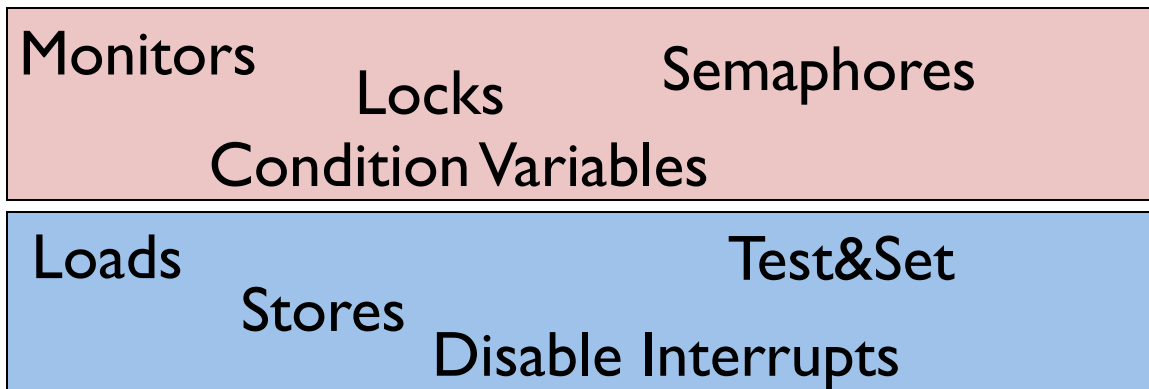
# SYNCHRONIZATION

Build higher-level synchronization primitives in OS

Operations that ensure correct ordering of instructions across threads

Use help from hardware

Motivation: Build them once and get them right



# CONCURRENCY SUMMARY

Concurrency is needed for high performance when using multiple cores

→ abstraction

(Threads) are multiple execution streams within a single process or address space (share PID and address space, own registers and stack)

Context switches within a critical section can lead to non-deterministic bugs

↓  
mutual  
exclusion

**LOCKS**

# LOCKS

Goal: Provide mutual exclusion (mutex)

↳ simple

Allocate and Initialize

– Pthread\_mutex\_t mylock = PTHREAD\_MUTEX\_INITIALIZER;

Acquire


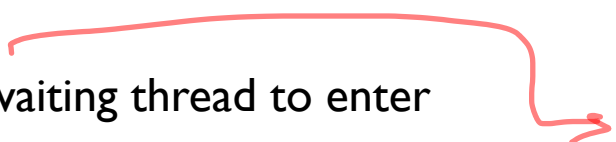
- Acquire exclusion access to lock;
- Wait if lock is not available (some other process in critical section)
- Spin or block (relinquish CPU) while waiting
- Pthread\_mutex\_lock(&mylock);

Release

- Release exclusive access to lock; let another process enter critical section
- Pthread\_mutex\_unlock(&mylock);

# 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

all threads are waiting for lock  
"dead lock"

waiting for lock is bounded

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

*Provides  
mutual  
exclusion*

```
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

*- Not Secure  
- Wasting CPU*



# IMPLEMENTING LOCKS: W/ LOAD+STORE

Code uses a **single shared lock variable**

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

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

if true lock is held  
false lock is free

lock var value  
true

T1

T2

T2 spin

T1 acquire

T1 release

false

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

T2 acquire

true

# LOCKS WITH VARIABLE DEMO

# RACE CONDITION WITH LOAD AND STORE

`*lock == 0 initially`

Thread 1

`while(*lock == 1)`

`*lock = 1`

Thread 2

`while(*lock == 1)`

`*lock = 1`

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

# XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

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;
}
```



→ "Test"

→ "set"


you want *addr* to have  
this value

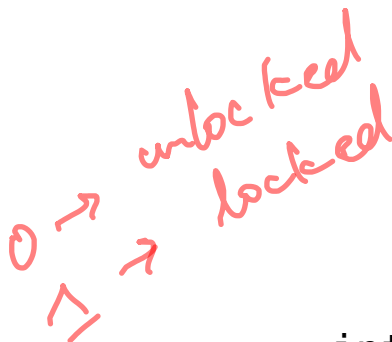
# LOCK IMPLEMENTATION WITH XCHG

```
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 = ??;   
}
```

  
0 → unlocked  
1 → locked

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

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

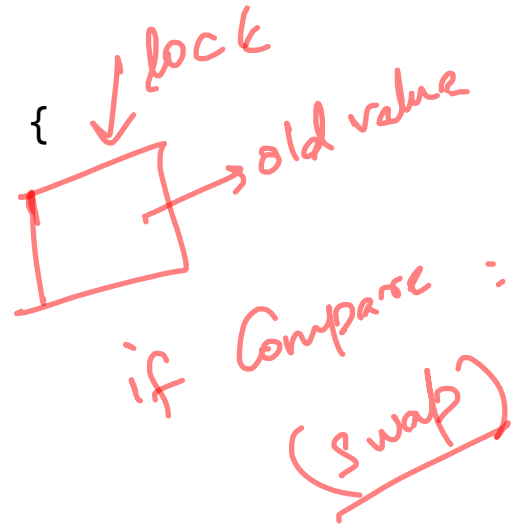
1

**DEMO XCHG**

# OTHER ATOMIC HW INSTRUCTIONS

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

20 unlock/CAS  
1 lock



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

if lock->flag == 0:  
 lock->flag = 1  
else:  
 return 0 ⇒ we got lock  
 1 ⇒ Not

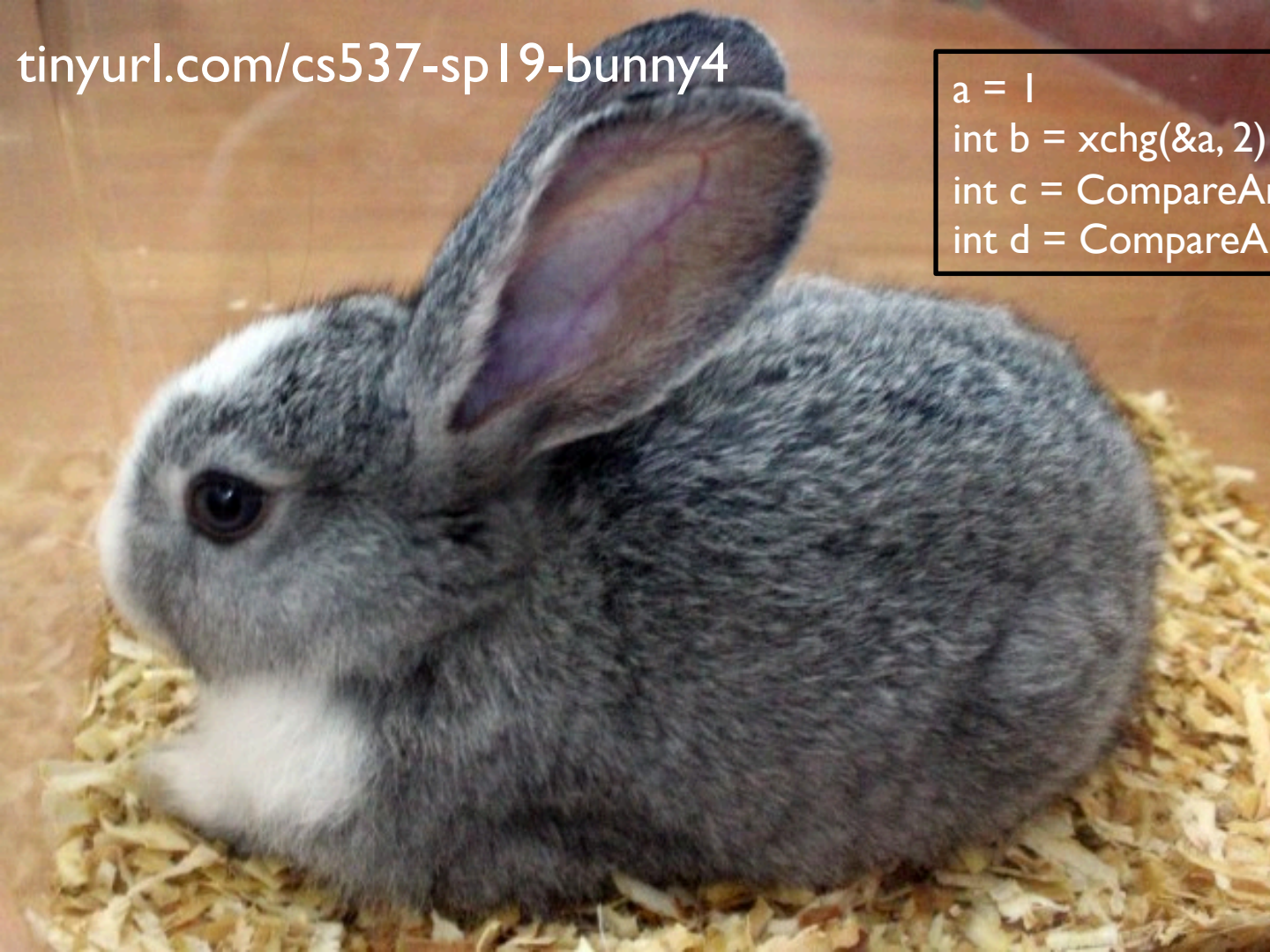
[tinyurl.com/cs537-sp19-bunny4](https://tinyurl.com/cs537-sp19-bunny4)

```
a = 1
```

```
int b = xchg(&a, 2)
```

```
int c = CompareAndSwap(&b, 2, 3)
```

```
int d = CompareAndSwap(&b, 1, 3)
```





# XCHG, CAS

*compute  
new value*

a = 1

int b = xchg(&a, 2)

int c = CompareAndSwap(&b, 2, 3)

int d = CompareAndSwap(&b, 1, 3)

A

B

C

D

1

2

2

2

1

1

3

1

1

1

# LOCK IMPLEMENTATION GOALS

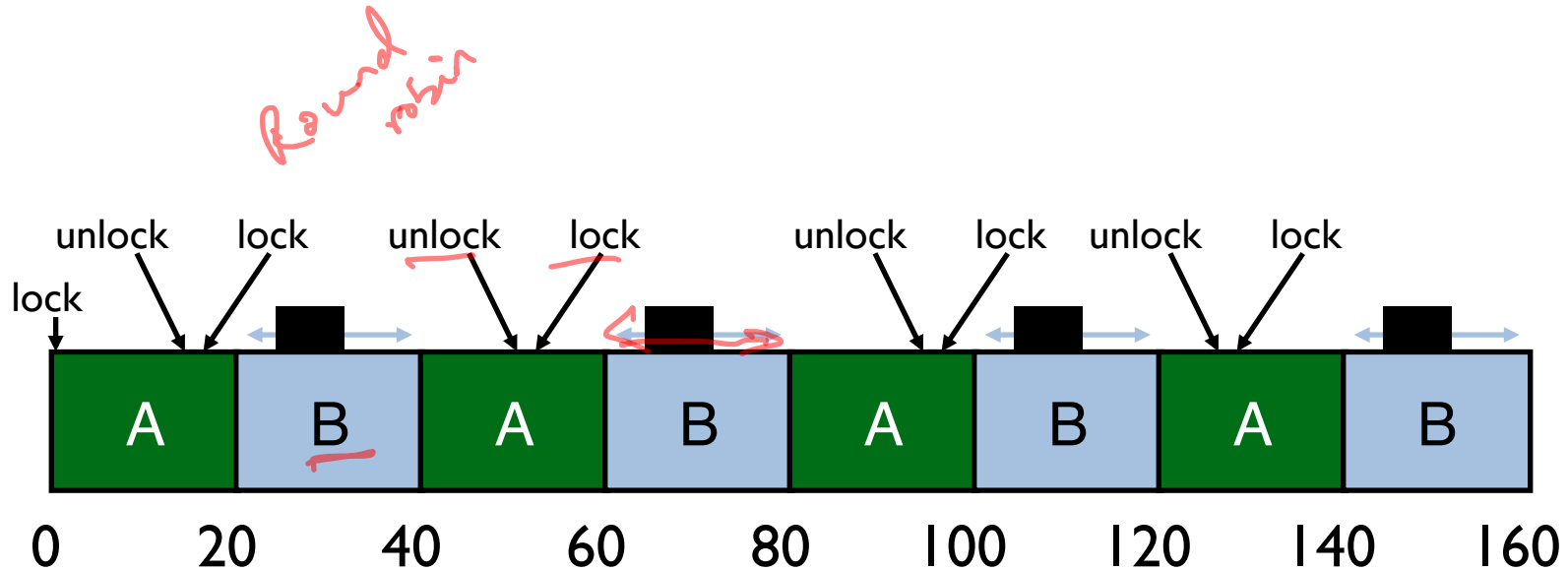
## Correctness

- ✓ – *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 unaware of locks/unlocks!

*Not fair*

# 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; → fetch  
    *ptr = old + 1; → add  
    return old; → old  
}
```

Acquire: Grab ticket; Spin while not thread's ticket != turn

Release: Advance to next turn

# TICKET LOCK EXAMPLE

Time

A lock(): *ticket = 0 A runs*

B lock(): *ticket = 1*

C lock(): *ticket = 2*

A unlock():

*turn ++*

A lock(): *ticket = 3*

*B runs*

B unlock():

*C runs*

C unlock():

A unlock():

*A runs*

Ticket

Turn = 0

0
1
2
3
4
5
6
7

# 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);  
}
```

→ Fetch And Add

# SPINLOCK PERFORMANCE

Fast when...

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when...

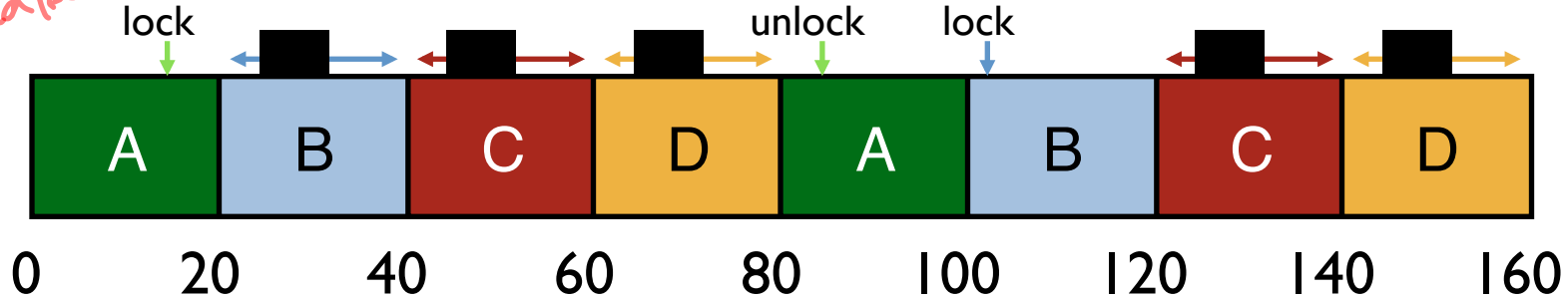
- one CPU
- locks held a long time
- disadvantage: spinning is wasteful in terms of CPU

# CPU SCHEDULER IS IGNORANT

*T1*  
A.lock  
B.lock  
deadlock!

*T2*  
B.lock  
A.lock

*locks  
as primitive*



CPU scheduler may run **B, C, D** instead of **A**  
even though **B, C, D** are waiting for **A**

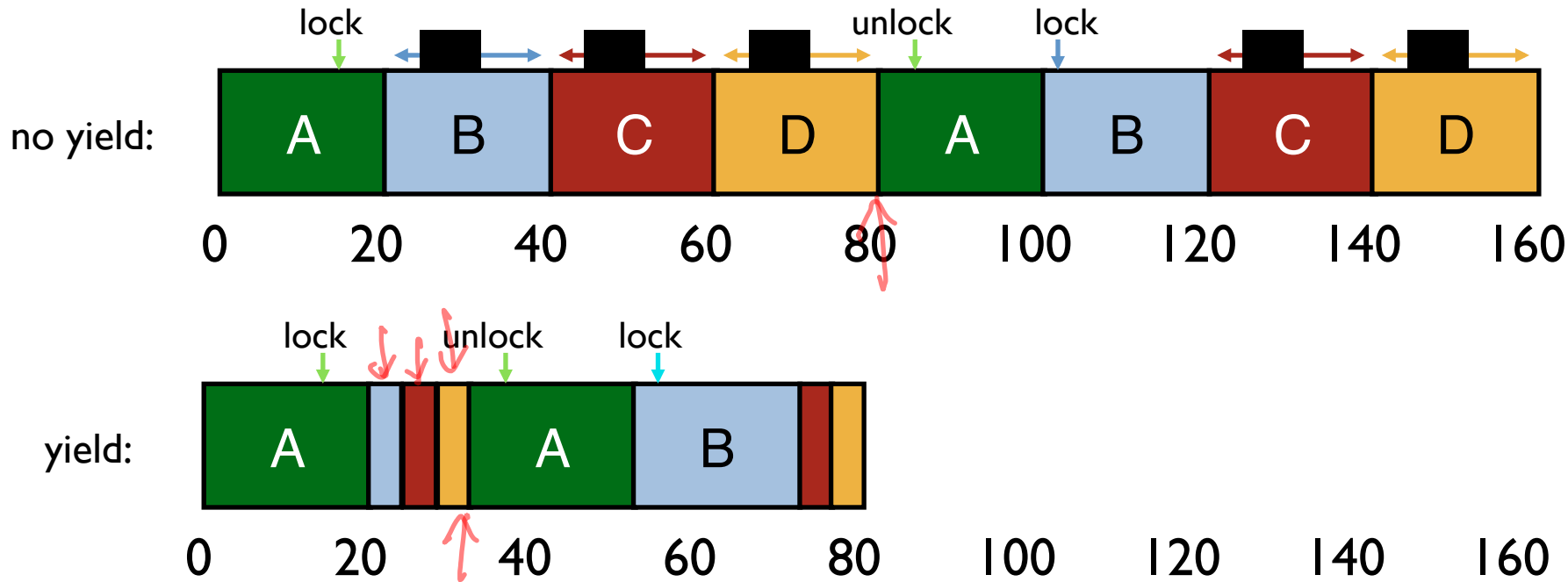


# TICKET LOCK WITH YIELD

```
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



<https://tinyurl.com/cs537-sp19-bunny5>

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

Timeline

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

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

C: lock() ...



# YIELD VS SPIN

Assuming round robin scheduling, 10ms time slice

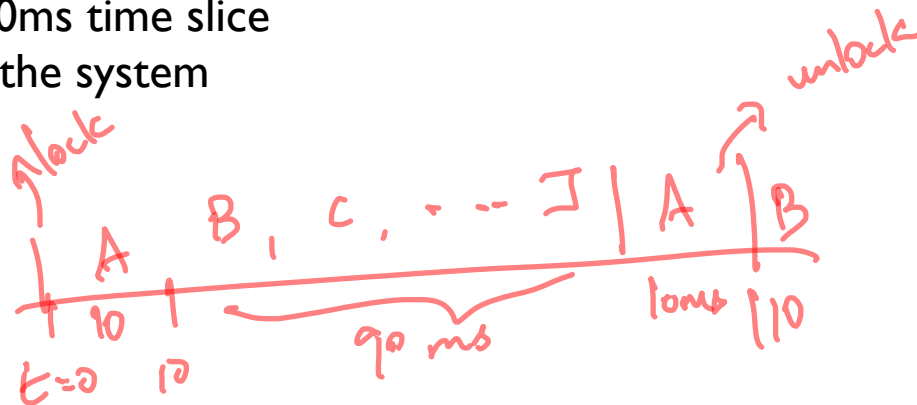
Processes A, B, C, D, E, F, G, H, I, J in the system

Timeline

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

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

C: lock()



If A's compute is 20ms long, starting at  $t = 0$ , when does B get lock with spin ?

110 ms

If B's compute is 30ms long, when does C get lock with spin ?

If context switch time = 1ms, when does B get lock with yield ?

# 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

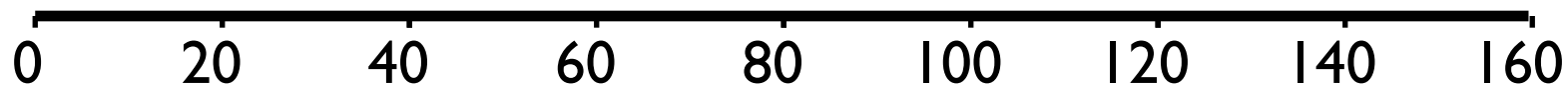
Remove waiting threads from scheduler ready queue  
(e.g., `park()` and `unpark(threadID)`)

Scheduler runs any thread that is **ready**

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()** 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 block?

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

Project 2b: Due tomorrow!

Next class: Condition Variables