CONCURRENCY: LOCKS

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

- Exam grades in handin/<LOGIN>/midterm1.pdf
  - In Canvas now
  - Average: 73/92 points (quintiles: 81, 77, 71, 65, 44)
  - Future exams: Slightly cumulative
- Project 4 Due Tuesday, Oct 22 5pm
  - Fill out form if want partner match
- Discussion Sections tomorrow
  - Midterm1 Answers
  - Project 4

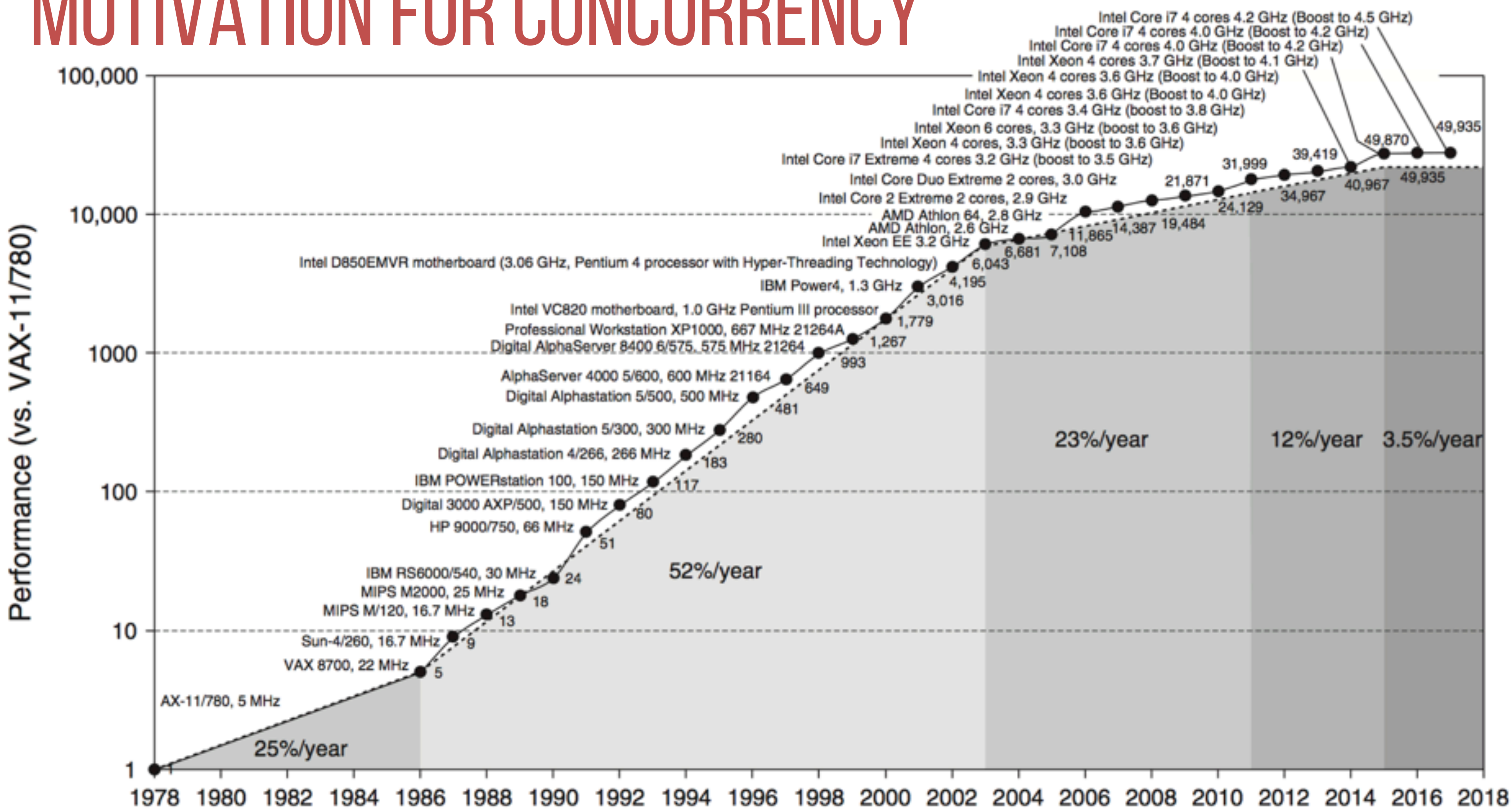
AGENDA / LEARNING OUTCOMES

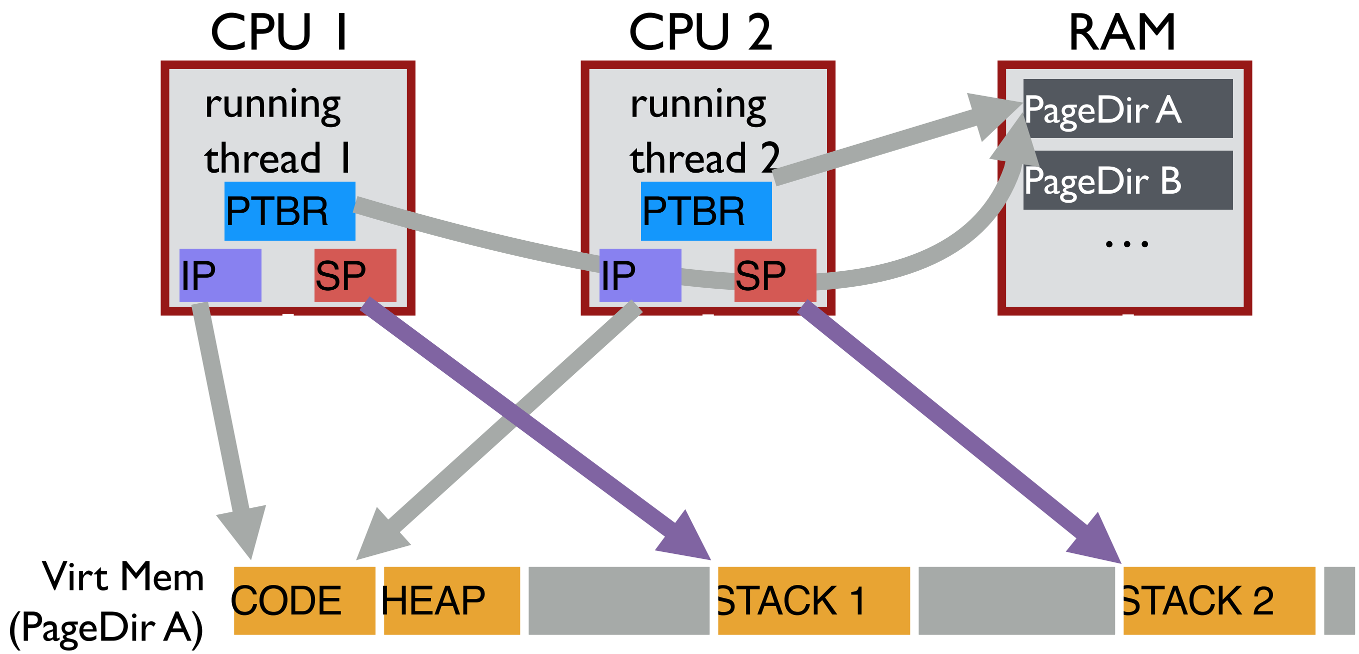
Concurrency

- Review **threads** and **mutual exclusion** for **critical sections**
- How can locks 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?

RECAP

MOTIVATION FOR CONCURRENCY





Do threads share stack pointer?

Threads executing different functions need different stacks
(But, stacks are in same address space, so trusted to be cooperative)

TIMELINE VIEW

Thread 1

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

Thread 2

mov 0x123, %eax
add %0x2, %eax
mov %eax, 0x123

Registers are virtualized: view each thread as having own copy of registers

Same concepts whether multiple cores (parallelism) or time-sharing single core (concurrency)

TIMELINE VIEW

Thread 1	**Thread 2**
mov 0x123, %eax	
	mov 0x123, %eax
	add %0x2, %eax
	mov %eax, 0x123
add %0x1, %eax	
mov %eax, 0x123	

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
```

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)

LOCKS

LOCKS

Goal: Provide mutual exclusion (mutex)

Allocate and Initialize

- Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
- Allocate on heap so all threads can share

Acquire

- Acquire exclusive 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);

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?

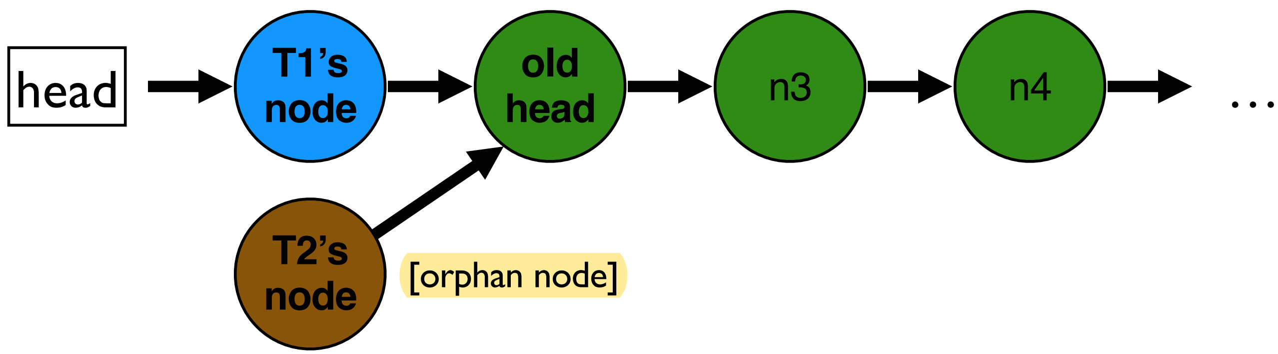
LINKED-LIST RACE

Thread 1 Thread 2

Both entries point to old head

Only one entry (which one?) 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;
}
```

Declaration: `pthread_mutex_t lock;`

One lock per list – Fine if add to OTHER lists concurrently

```
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);
}
```

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));
    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;
    }
return 0;
}

Pthread_mutex_unlock(&L->lock);
```

LOCKING LINKED LISTS : APPROACH #2

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

Pthread_mutex_lock(&L->lock);

Pthread_mutex_unlock(&L->lock);

Make as small as possible

```
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()?

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

If no List_Delete(), locks not needed

```
Pthread_mutex_unlock(&L->lock);
```

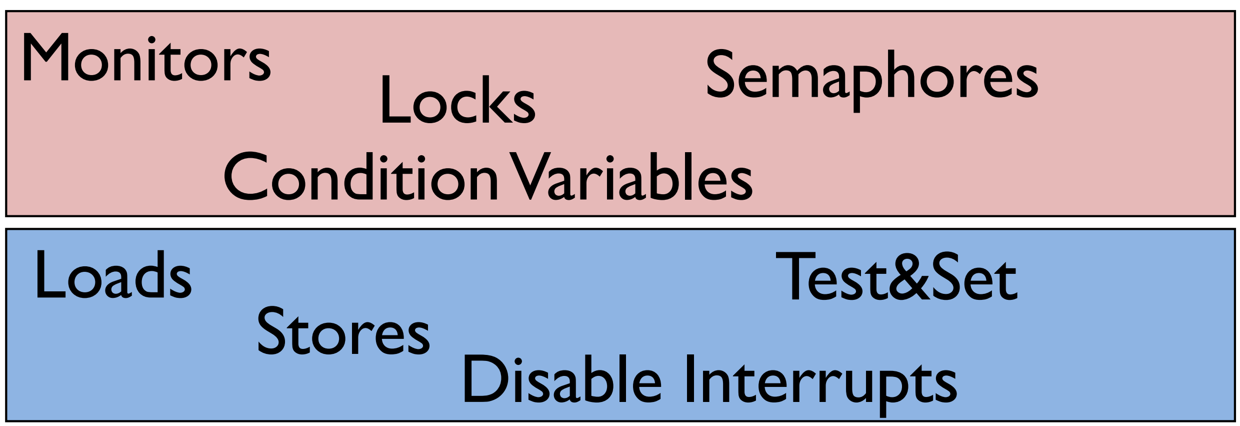
SYNCHRONIZATION

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



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* waiting (starvation-free)
  Must eventually allow each waiting thread to enter

Fairness: Each thread waits in some defined order

Performance:

- CPU is not used unnecessarily (e.g., no spin-waiting)
- Fast to acquire lock if no contention with other threads

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?

Only works on uniprocessors

Process can keep control of CPU for arbitrary length

Cannot perform other necessary work

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/ 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;

}
```

Example of spin-lock

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

LOCKS WITH LOAD/STORE DEMO

RACE CONDITION WITH LOAD AND STORE

`*lock == false initially`

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

Both threads grab lock!

Problem: Testing lock and setting lock are not atomic

THEORETICAL: PETERSON'S ALGORITHM

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

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

DIFFERENT CASES: ALL WORK

Only thread 0 wants lock initially

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

In critical section

```
                                                Lock[1] = true;
                                                turn = 0;

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


lock[0] = false;

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

DIFFERENT CASES: ALL WORK

Thread 0 and thread 1 both try to acquire lock at same time

```
Lock[0] = true;
```

```
Lock[1] = true;
```

```
turn = 1;
```

```
turn = 0;
```

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

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

Finish critical section

```
lock[0] = false;
```

```
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: INTUITION

Mutual exclusion: Enter critical section if and only if

- Other thread does not want to enter OR
- Other thread wants to enter, but your turn (only 1 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)

- Each process waits at most one critical section (because turn given to other)

Problem: doesn't work on modern hardware
(hw doesn't provide sequential consistency due to caching)

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**

XCHG: ATOMIC EXCHANGE OR TEST-AND-SET

```
// ATOMIC: 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)
```

LOCK IMPLEMENTATION WITH XCHG

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

DEMO XCHG

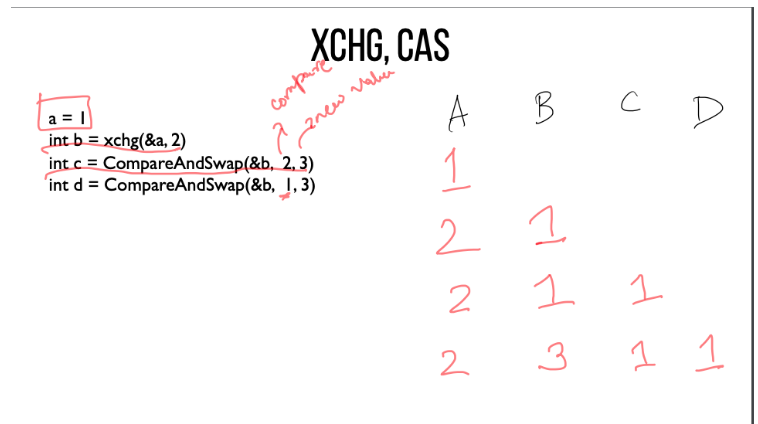
OTHER ATOMIC HW INSTRUCTIONS

```
int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}
```

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

CHAT: WHAT ARE THE VALUES?

```
int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}
```




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

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

CHAT: WHAT ARE THE VALUES?

```
int xchg(int *addr, int newval) {
  int old = *addr;
  *addr = newval;
  return old;
}
```

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

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

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

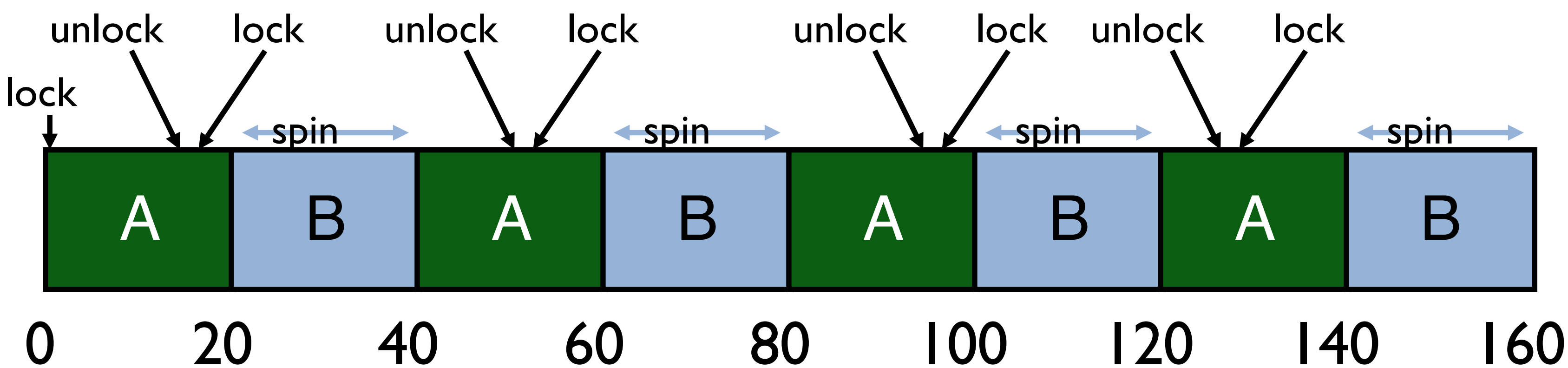
Correctness

- *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 unaware of locks/unlocks!
B may be unlucky and never acquire lock

BOUNDED WAITING + FAIRNESS: TICKET LOCKS

Idea: reserve each thread's turn to use a lock.

Each thread spin-waits until their turn

Use new atomic hw primitive, fetch-and-add

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

Acquire: Grab ticket; Spin while thread's ticket != turn

Release: Advance to next turn

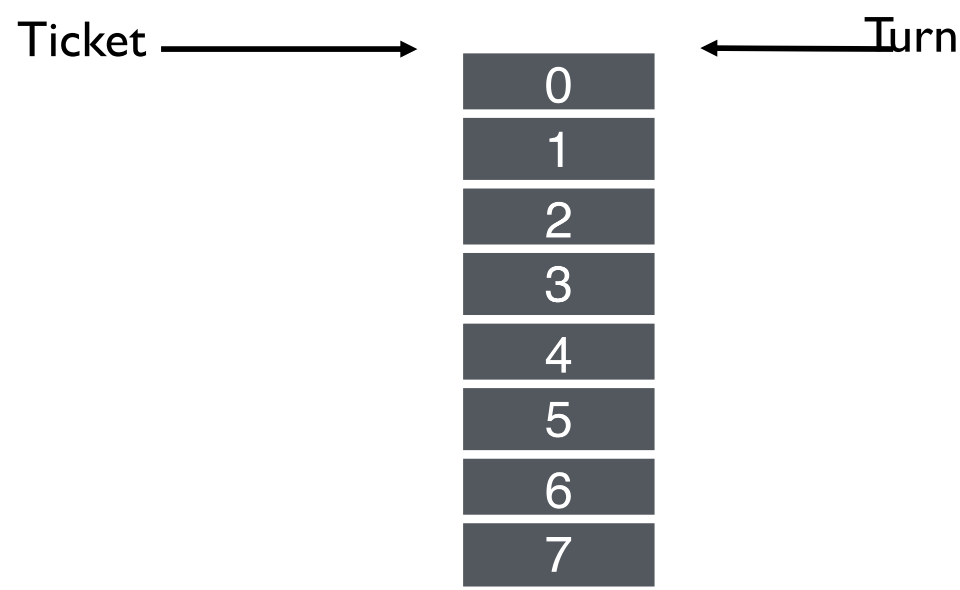
TICKET LOCK EXAMPLE

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

A unlock():

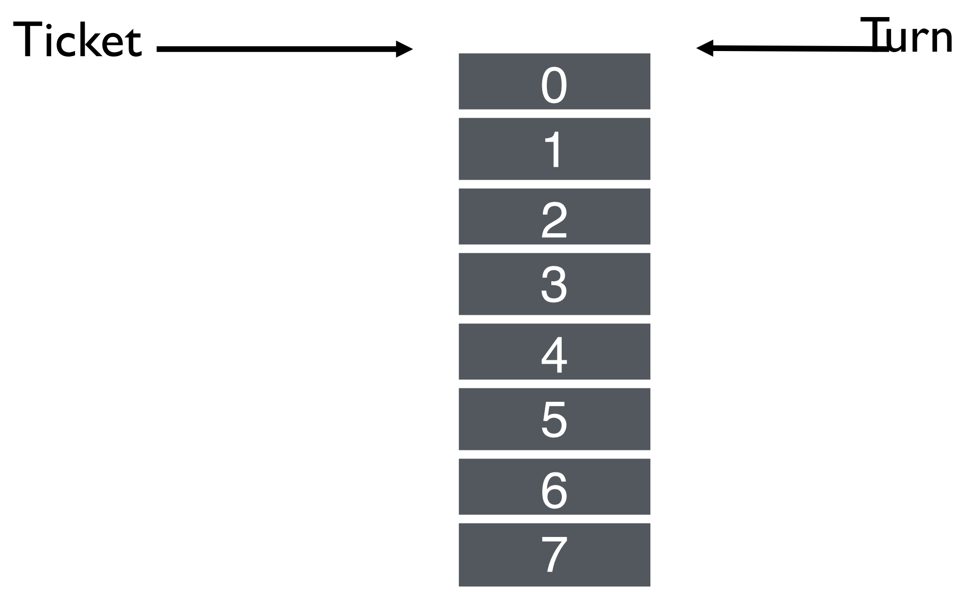
A lock():
B unlock():

C unlock():
A unlock():



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

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

TICKET LOCK IMPLEMENTATION

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

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

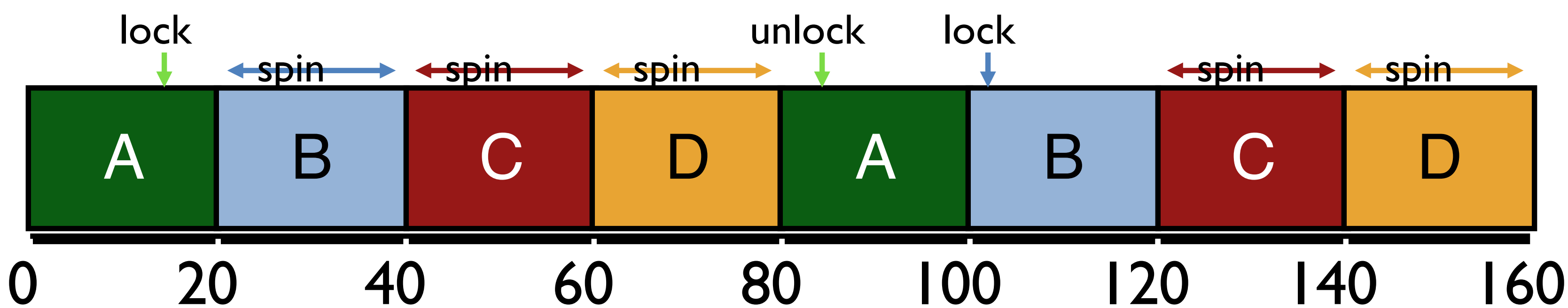
```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    // spin
    while (lock->turn != myturn);
}

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

SPINLOCK PERFORMANCE

Fast when…
- many CPUs
- locks held a short time
- advantage: avoid overhead of 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**

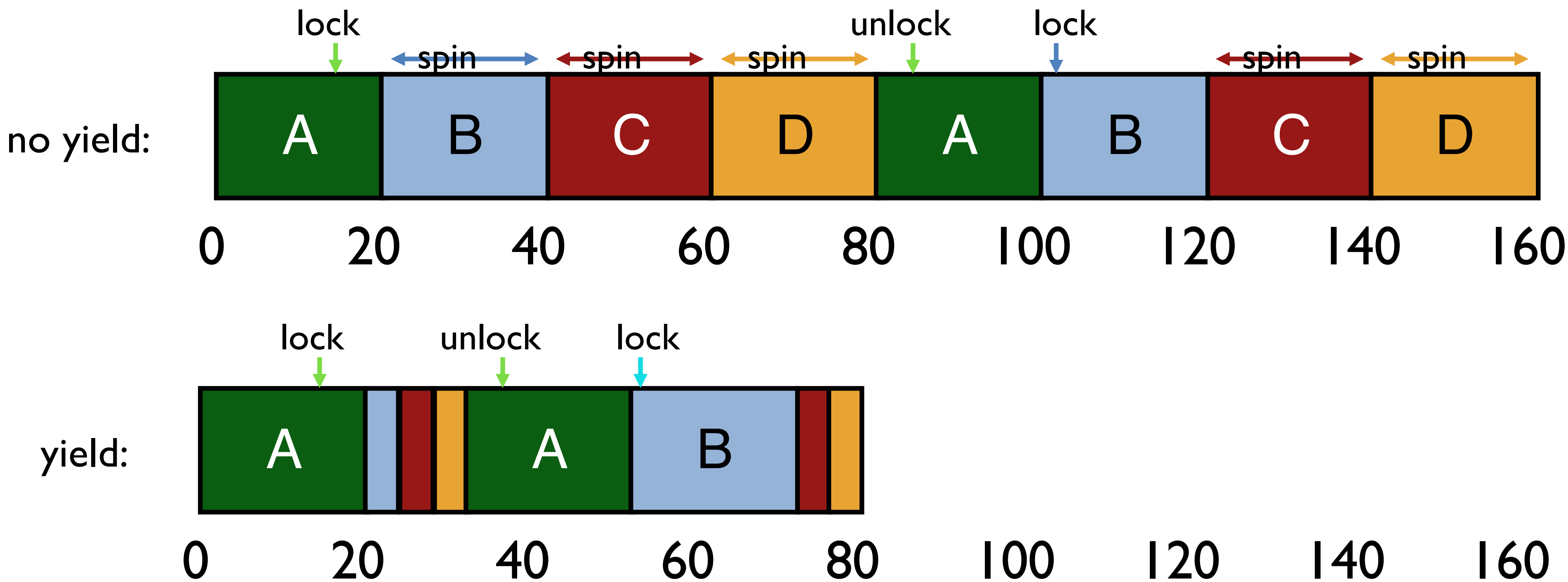
TICKET LOCK WITH YIELD

```
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);
}
```

Remember: yield() voluntarily relinquishes CPU for remainder of timeslice, but process remains READY

YIELD INSTEAD OF SPIN



CHAT: YIELD VS SPIN

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

Timeline – all processes doing same lock/unlock pattern
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 ?

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

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

NEXT 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**