CS 423
Operating System Design: Concurrency: Locks
02/28

Ram Alagappan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

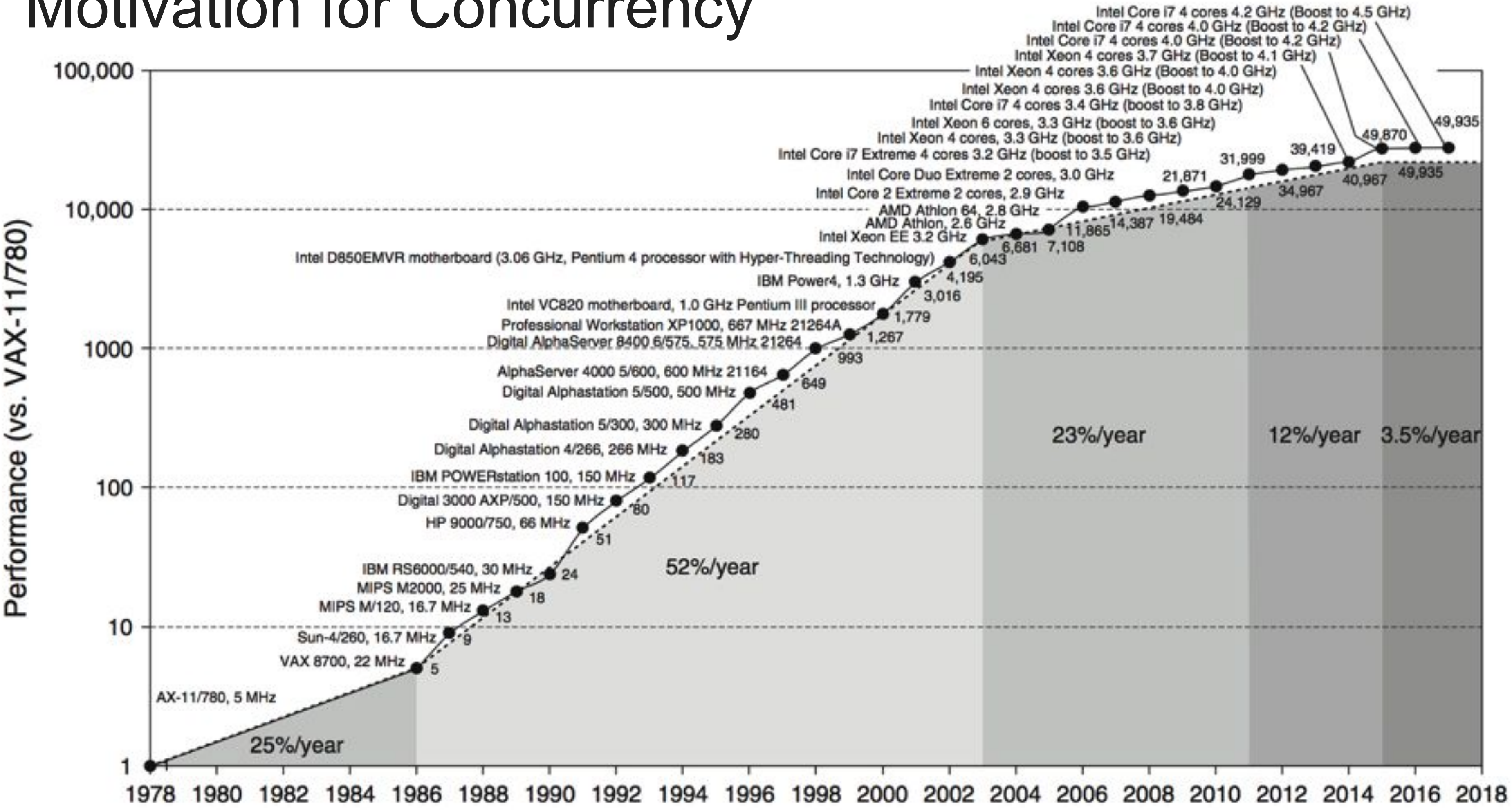
AGENDA / LEARNING OUTCOMES

Concurrency:

- Finish discussion on threads
- Locks:
  - How to use them?
  - How to implement them?

RECAP

Motivation for Concurrency



Common Programming Models

Multi-threaded programs tend to be structured as:

- **Producer/consumer**
  Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  Task is divided into series of subtasks, each of which is handled in series by a different thread
- **Defer work with background thread**
  One thread performs non-critical work in the background (when CPU idle)

END RECAP

OS Support: Approach 1

User-level threads: Many-to-one thread mapping

- Implemented by user-level runtime libraries

  Create, schedule, synchronize threads at user-level
- OS is not aware of user-level threads

  OS thinks each process contains only a single thread of control

Advantages

- Does not require OS support; Portable
- Can tune policies to suit application
- Lower overhead thread operations since no system call

Disadvantages?

- Cannot leverage multiprocessors
- Entire process blocks when one thread blocks

OS Support: Approach 2

Kernel-level threads: One-to-one thread mapping

- OS provides each user-level thread with a kernel thread
- Each kernel thread scheduled independently
- Thread operations (creation, scheduling, synchronization) performed by OS

Advantages

- Each kernel-level thread can run in parallel on a multiprocessor
- When one thread blocks, other threads from process can be scheduled

Disadvantages

- Higher overhead for thread operations

THREAD SCHEDULE

```c
volatile int balance = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
      balance++;
    }
    pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

THREAD SCHEDULE #1

`balance = balance + 1; balance at 0x9cd4`

Registers are virtualized by OS;
Each thread thinks it has own

State:
0x9cd4: 100
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: ? %rip: 0x195	%eax: ? %rip: 0x195

T1 ➡

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

THREAD SCHEDULE #1

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: ?	%eax: ?
%rip: 0x195	%rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

TI

THREAD SCHEDULE #1

State:
0x9cd4: 101
%eax: ?
%rip = 0x195

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

T2

```
0x195   mov 0x9cd4, %eax
0x19a   add $0x1, %eax
0x19d   mov %eax, 0x9cd4A
```

THREAD SCHEDULE #1

State:
0x9cd4: 102
%eax: 102
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T2

Desired Result!

THREAD SCHEDULE #2

State:
0x9cd4: 100
%eax: 101
%rip = 0x19d

process control blocks:

	Thread 1	Thread 2
%eax:	?	?
%rip:	0x195	0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1 → 0x19d

Thread Context Switch before T1 executes 0x19d

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x19d	%eax: ? %rip: 0x195

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T2 →

THREAD SCHEDULE #2

State:
0x9cd4: 101
%eax: 101
%rip = 0x1a2

process control blocks:

Thread 1	Thread 2
%eax: 101 %rip: 0x1a2	%eax: 101 %rip: 0x1a2

```
0x195  mov 0x9cd4, %eax
0x19a  add $0x1, %eax
0x19d  mov %eax, 0x9cd4A
```

T1

WRONG Result! Final value of balance is 101

TIMELINE VIEW

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

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

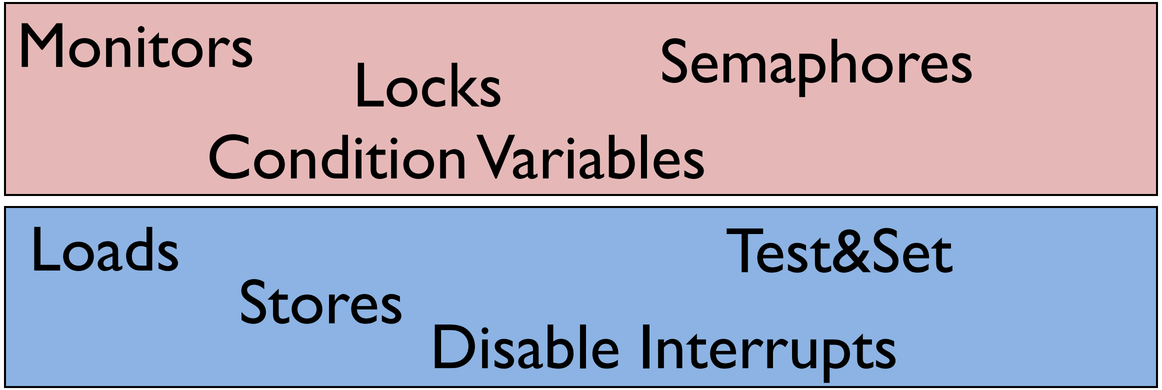
Synchronization

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



LOCKS

Locks

Goal: Provide mutual exclusion (mutex)

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);

Thread-Safe Queue

```
tryget() {
    item = NULL;
    lock.acquire();
    if (front < tail) {
        item = buf[front % MAX];
        front++;
    }
    lock.release();
    return item;
}
```

```
tryput(item) {
    lock.acquire();
    if ((tail – front) < size) {
        buf[tail % MAX] = item;
        tail++;
    }
    lock.release();
}
```

Initially: front = tail = 0; lock = FREE; MAX is buffer capacity

If tryget return NULL, can we be sure that there are no elements in the queue at that point?

What if we do tryget in a loop?

Lock Implementation Goals

Correctness

- *Mutual exclusion (safety)*
  Only one thread in critical section at a time
- *Progress* (liveness)
  If several simultaneous requests, must allow one to proceed

Fairness: does each thread have a fair shot at acquiring? Does anybody starve?

Performance: CPU is not used unnecessarily (spinning)

Implementing Locks

Approaches

- Disable interrupts
- Load and stores of words
- Using atomic hardware instructions (e.g., test and set)

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?

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

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

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

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

Using only Loads and Stores

Peterson's algorithm - uses only atomic load and store instructions to implement locks

Cumbersome to reason about correctness

But no lock is implemented this way today…

Most locks assume some support from hardware (or even OS)

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: return old value, Set: set addr with passed in value, HW does them atomically

LOCK Implementation with XCHG

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

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

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)
}
```

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion (safety)*
  Only one thread in critical section at a time
- *Progress* (liveness)
  If several simultaneous requests, must allow one to proceed

Fairness: does each thread have a fair shot at acquiring? Does anybody starve?

Performance: CPU is not used unnecessarily (spinning)

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)
}
```

Chat for a minute…

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

Final values?

LOCK IMPLEMENTATION GOALS

Correctness

- *Mutual exclusion*

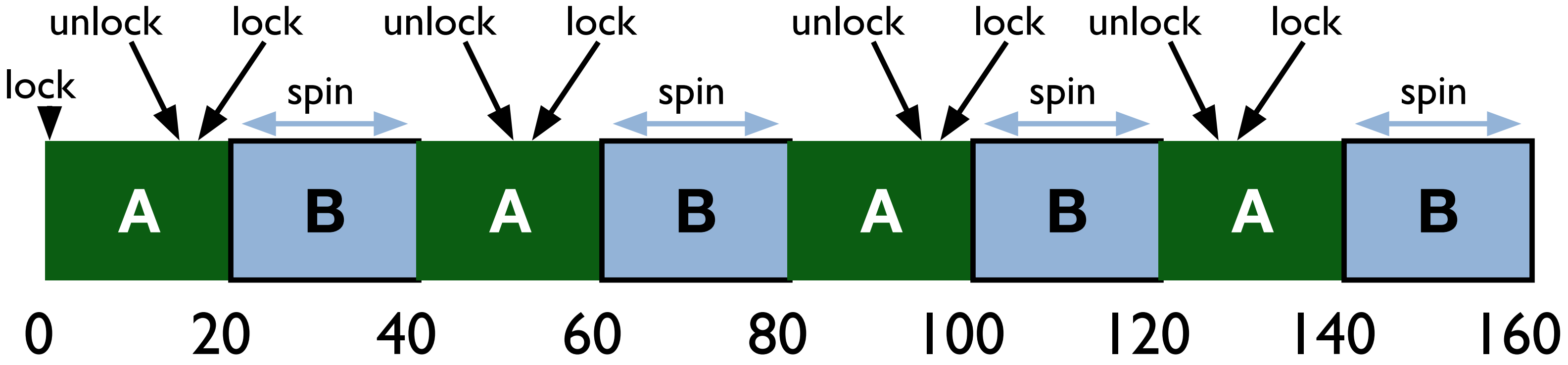
  Only one thread in critical section at a time
- *Progress* (deadlock-free)

  If several simultaneous requests, must allow one to proceed

Fairness: does each thread have a fair shot at acquiring? Does anybody starve?

Performance: CPU is not used unnecessarily (spinning)

BASIC SPINLOCKS ARE UNFAIR



Scheduler is unaware of locks/unlocks!
B is unlucky - never is able to acquire 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 IMPLEMENTATION

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

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

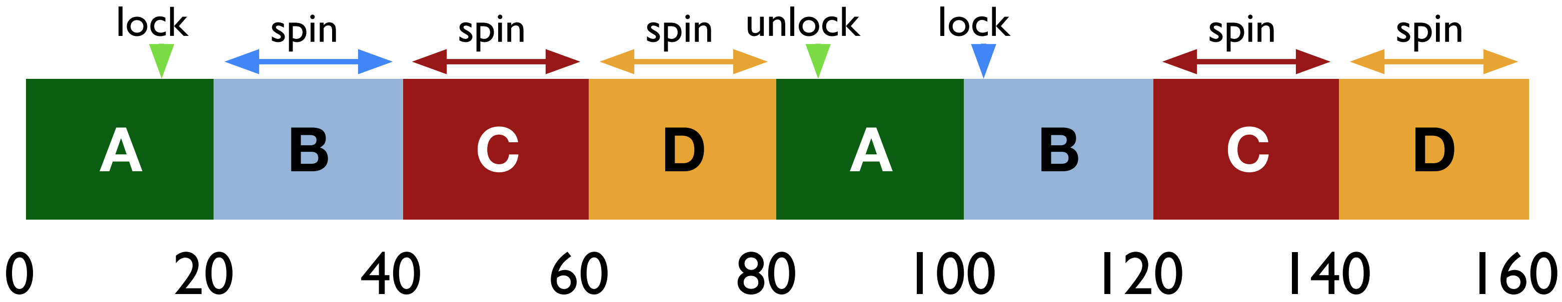
```
void acquire(lock_t *lock) {
    int myturn = FAA(&lock->ticket);
    // spin
    while (lock->turn != myturn);
}

void release(lock_t *lock) {
    FAA(&lock->turn);
    // can you do (&lock->turn)++?
}
```

SPINLOCK PERFORMANCE

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**

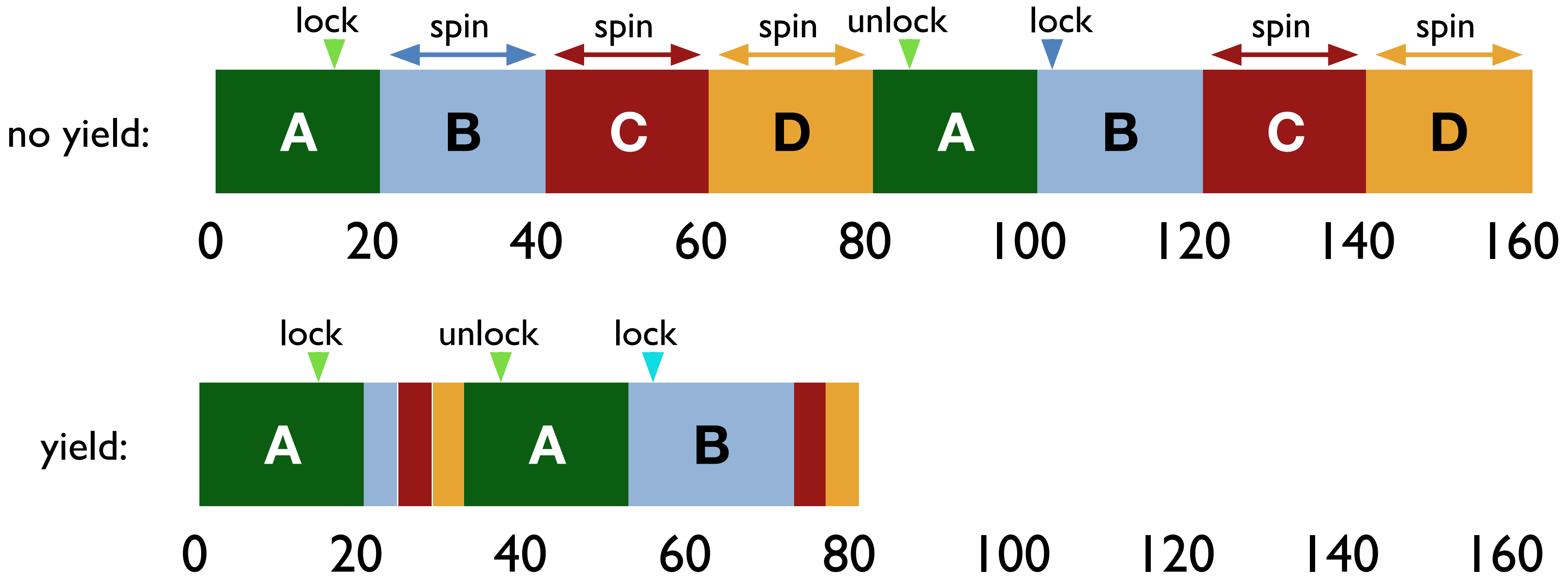
TICKET LOCK WITH YIELD

```
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() voluntarily relinquishes the CPU for remainder of time slice, but process remains READY

Yield Instead of Spin



YIELD VS SPIN

Waste of CPU cycles?

Without yield: O(threads * **time_slice**)

With yield: O(threads * **context_switch**)

While yield is better than spinning, with high contention, it can also be bad

Problem: the thread is still in READY state (and so can be scheduled on CPU)

Next improvement: Block and put thread on waiting queue

Lock Implementation: Block when Waiting

Remove waiting threads from scheduler runnable queue

Scheduler runs any thread that is **runnable**

Support in Solaris OS: park(), unpark()

Support in Linux: futex (fast userspace mutex)