EECS-343 Operating Systems
Lecture 11:
Implementing Locks

Steve Tarzia

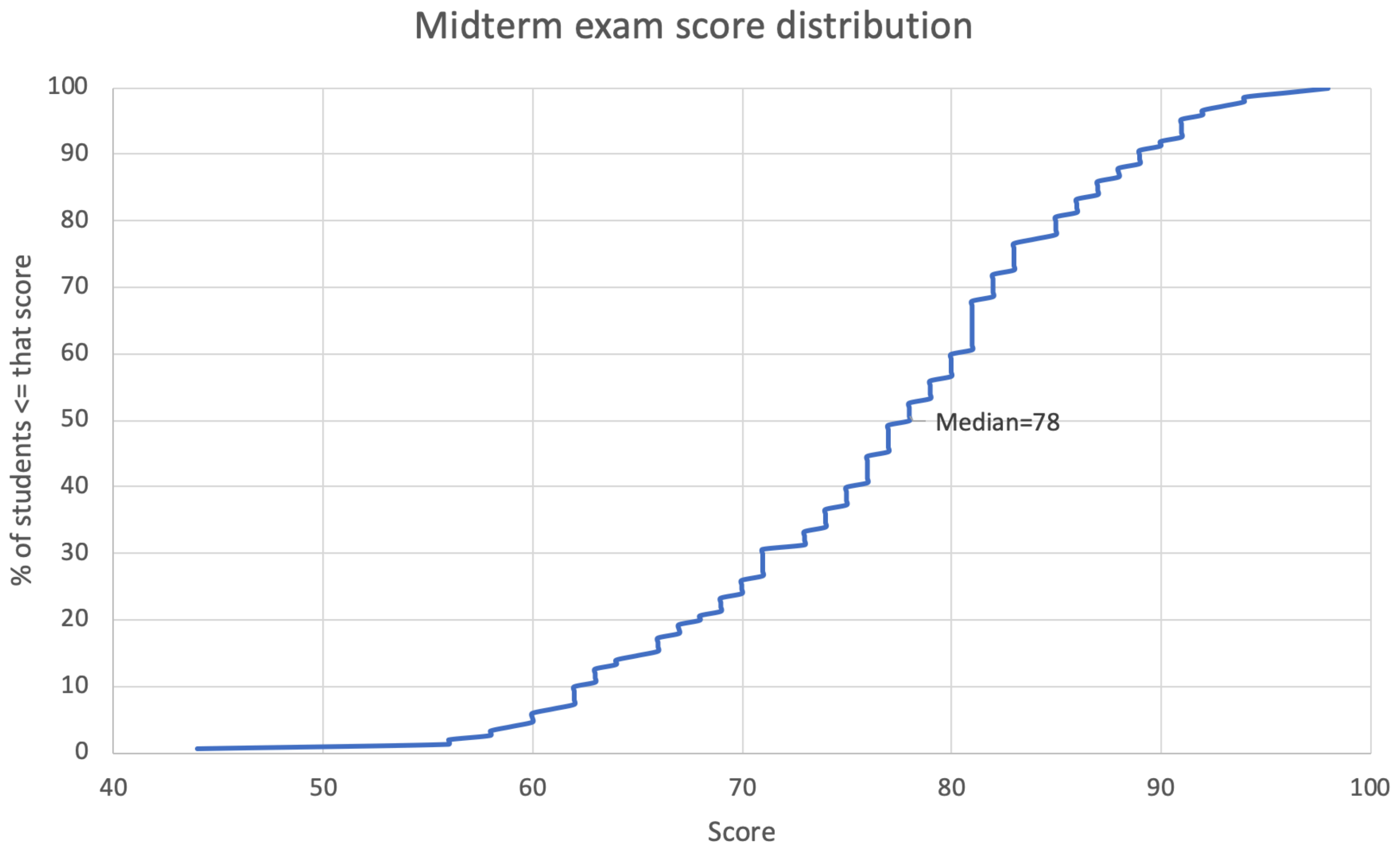
Spring 2019

Northwestern

Diagrams in the slides are by Arpaci-Dusseau

Announcements

- Drop deadline is tomorrow. Stop by my office if you're thinking about dropping.



Last Lecture: Threads

- Processes can have multiple ***threads*** sharing the virtual address space
- ***Critical sections*** are block of code that must be run ***atomically***
- If unprotected, critical sections lead to ***race conditions*** that make code ***indeterminant*** – we get different results depending on timing.
- ***Locks*** are the simplest ***mutual exclusion primitive***, with two main functions:
  - **Acquire/lock** – get exclusive access to a shared resource.
  - **Release/unlock** – release the shared resource.
- Concurrency occurs naturally in multi-CPU systems
- Concurrency is created by the process scheduler in single-CPU systems

Simple approach for single-CPU: *disable interrupts*

```
void lock() {
  disable_interrupts();
}

void unlock() {
  enable_interrupts();
}
```

- Disabling interrupts prevents preemption during a critical section.
- This simple approach is used in some kernel code, **but**:
  - Does not help with concurrency on multiple CPUs.
  - Masking/unmasking interrupts is slow.
  - Important HW interrupts might be lost.
  - Would give too much power to user code
    - A process could acquire a lock and run forever
    - With interrupts disabled, kernel has no ability to preempt user processes.
- So, this is **not** a very useful lock strategy.

Using a lock flag

```
int locked = 0;

void lock() {
  while (locked);
  locked = 1;
}

void unlock() {
  locked = 0;
}
```

- Lock will keep checking the flag until it's unset, then set it to exclude any other threads.
- But this implementation does not work because two threads may *simultaneously* see locked==0, exit the while loop, and both grab the lock.
- In other words, the ***test*** and the ***set*** are not atomic.

CPU hardware support for concurrency

- Atomic ***test-and-set*** instruction
  - Called "atomic exchange" – "`lock; xchg`" on x86
- Operates on a particular memory location
- Simultaneously sets a new value and returns the old value

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr; // fetch old value at ptr
    *ptr = new;     // store 'new' into ptr
    return old;     // return the old value
}
```

- It's *atomic*, so the three steps cannot be interrupted halfway through.

Our first useful **spinlock**, with test-and-set

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

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

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

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

xv6's spinlock.[ch]

```c
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

```c
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

```c
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  xchg(&lk->locked, 0);

  popcli();
}
```

Compare and Swap

- Another, more powerful, atomic instruction
- Atomically compares a memory location to a register, returns the original result, and sets a new value if the comparison was true.

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

- It's a generalization of test-and-set
  - TestAndSet(ptr, new) → CompareAndSwap(ptr, *ptr, new)
- "`lock; cmpxchg`" in x86 assembly

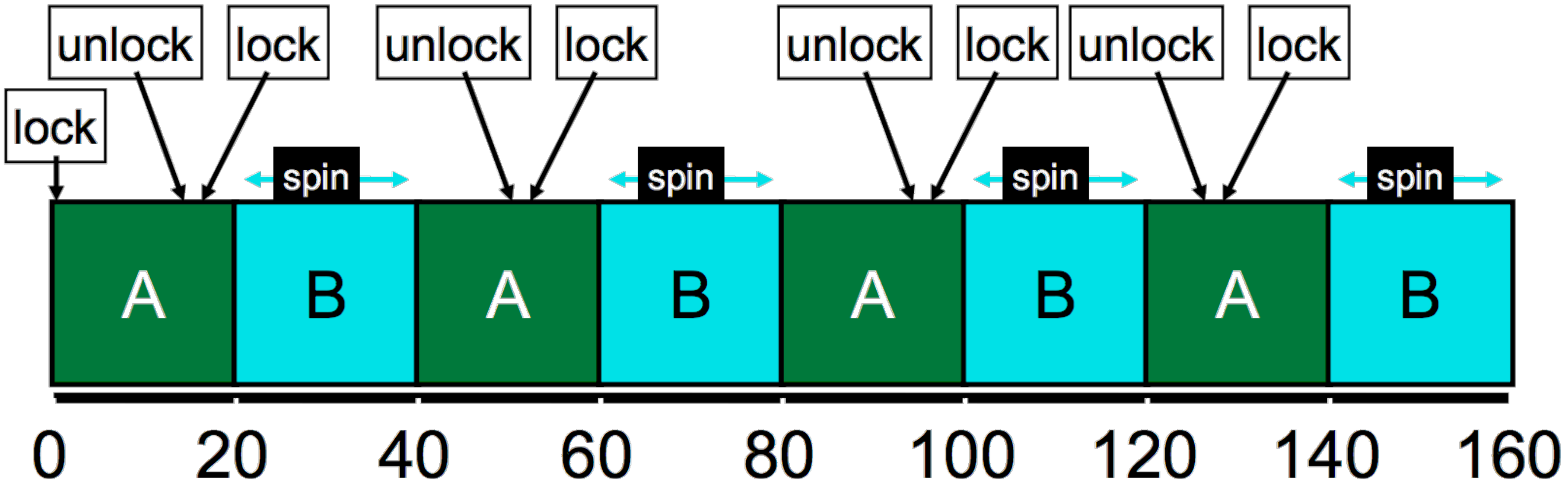
How to evaluate a lock implementation?

- ***Correctness*** – must provide mutual exclusion
- ***Fairness*** – threads acquire lock in the order they request it
- ***Progress*** – if several threads request the lock, one must acquire it
  - (avoid *deadlock*)
- ***Bounded wait*** – no thread should wait forever (or starve).
- ***Performance*** – minimize latency/overhead introduced by the lock

Simple spinlock problems

- Lacks ***Fairness*** and ***bounded wait*** – starvation can occur.
  - Next thread to acquire the lock is whichever the scheduler chooses
  - Even if scheduler is "fair" and schedules the waiting process periodically, there is no guarantee that the lock will be available when scheduled.
- ***Performance*** – (on uniprocessor)
  - CPU "spins," repeatedly checking a variable that will not change.
  - Timeslice must expire before another thread is given a chance to unlock
  - If *N* threads want the lock, then *N* timeslices can be wasted spinning.
  - Notice that spinlocks may be efficient on a multiprocessor, because a thread on another core may release the lock being waited for.
- Nevertheless, the spinlock is ***correct***, simple, and safe for user code.

Spinlock starvation illustrated



- Problem is that scheduler has no knowledge of locks, and locking threads have no control over scheduler
- **B** makes no progress *and* wastes a timeslice every time it is scheduled!

Fetch and add

- Return old value and increment it
- This is yet another atomic instruction for concurrency
- Can be used to atomically reserve a "ticket number"
- "`lock; xadd`" in x86 assembly

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

Ticket lock

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

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

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

void unlock(lock_t *lock) {
    FetchAndAdd(&lock->turn);
}
```

- Each thread uses fetch-and-add to atomically ***reserve*** its turn number.
- In lock(), spin while checking whether it's your turn.
    - Unique turn numbers prevent race
- To unlock, just increment "turn"
- ***Prevents starvation*** because threads acquire the lock in FIFO order.
- Atomic instruction is not really needed in unlock.
    - Can avoid overflow with:

```
lock->turn = (lock->turn + 1) % MAX_INT
```

GCC has built-in functions for atomic operations

- `type __sync_fetch_and_add (type *ptr, type value)`
- `bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval)`
- `type __sync_lock_test_and_set (type *ptr, type value)`
- `void __sync_lock_release (type *ptr)`
- … and more
  - See https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html

- These will be compiled to the appropriate atomic instructions on the particular target CPU architecture.

Load-linked & Store-conditional

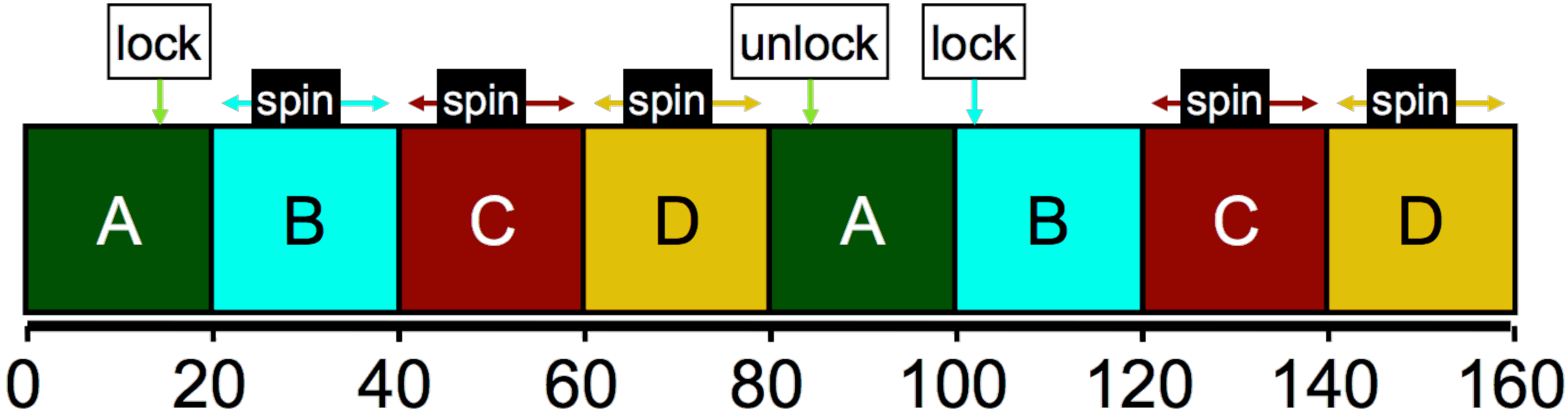
- A special pair of load/store instructions for concurrency
- ***Load-linked*** reads the value at an address
- ***Store-conditional*** writes a new value to an address
  - However, it aborts if there has already been a write to that address since the last load-linked.
- Can be used to implement a lock
  - But more importantly, can be used directly for ***lock-free*** concurrent code.
  - Instead of locking before working on shared memory, just use load-linked.
    - The store-conditional later on will tell you whether you need to retry.
- Supported on some RISC/ARM CPUs, but not on x86.

Intermission

"Do you know you've had your caps lock on for the last ten miles?"

Ticket lock avoids starvation, but it's still not ideal

- Imagine 4 processes competing on one CPU for a lock:



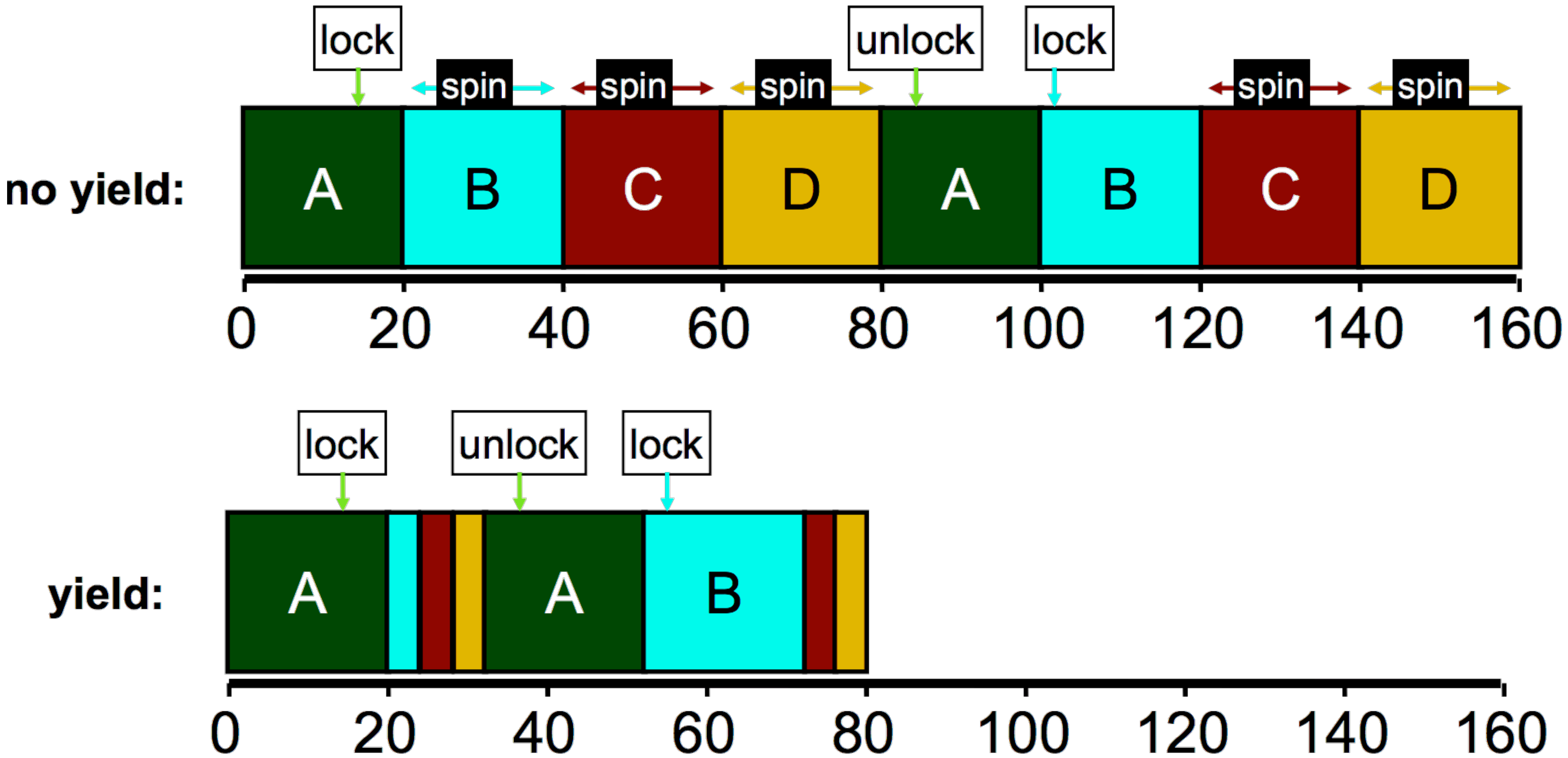
- B,C,D are wasting time by ***busy waiting***.
- Scheduler is trying to be fair to B,C,D by letting them run, but scheduler is ignorant of locks, and does not know they are just waiting.
- It would be better to skip B,C,D and let A finish the critical section.

Yielding is a simple solution

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

…

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

void release(lock_t *lock) {
    lock->turn += 1;
}
```

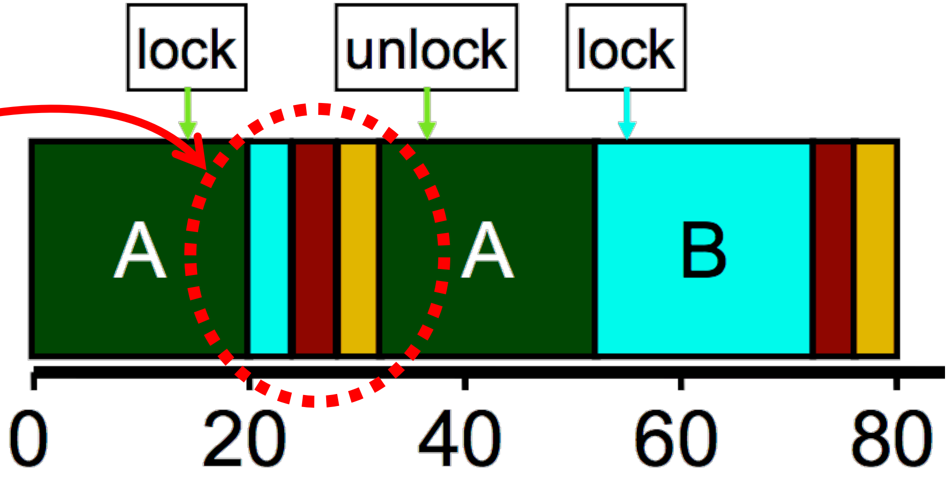
- Give the user process *just a little* control over the scheduler.
- Create a ***yield*** syscall that un-schedules the current thread (before the timeslice expires).
- In **acquire**, the thread will check the lock variable once.
- If the lock is not available, the thread is better off letting another thread run because it's waiting for someone else to unlock.

Yielding eliminates busy-waiting



One remaining problem

- Performance is better with ***yield***, but we are still doing a lot of unnecessary context switches
- Remember that context switches are costly because they flush caches & TLB.



- Solution is to make the scheduler aware of who is holding which locks
- Then scheduler can avoid scheduling a thread until the lock it's waiting for is free.
- Thread "A" should be scheduled again at time 20, because the other three processes are all waiting for a lock that has not yet been released.

Blocking locks and wait queues

- A better solution requires some cooperation between the user thread's locks and the OS scheduler.
- Solaris provides park/unpark syscalls to influence the scheduler:
  - ***Park*** blocks the current thread.
    - Yields, but also puts the thread in a special blocked state so it cannot run.
  - ***Unpark*** unblocks ***another*** thread, identified by thread_id.
- A lock based on park/unpark can be implemented as follows:
  - If lock **acquire** fails, add the thread to the queue of parked threads and park.
  - **release** dequeues the next waiting thread (if any) and unparks it, so it can run.
  - Queue resides in user memory and unlocking thread effectively decides which thread is scheduled next.
- See the book for details.

Linux Futex ("fast userspace mutex") syscalls

- Similar to park/unpark, but the queue is in the kernel.
- `futex_wait(address, expected)` – put the thread to sleep if the value at address equals "expected." Used in lock/acquire function.
- `futex_wake(address)` – wake one thread (in FIFO order) that previously called futex_wait. Used in unlock/release function.
- Behind the scenes, the kernel will create a queue for each address associated with a futex. (Queue will be protected by locks.)

Two classes of locks

Spinlocks

- Just use an atomic CPU instruction like test-and-set or fetch-and-add.
- "Spinning" is trying to acquire the lock repeatedly in a loop.

- Simple
- Wastes CPU time

Blocking locks

- Still require atomic instructions.
- But somehow tell the scheduler to run a different thread if lock acquire failed.

- Frees up the CPU
- But context switches are costly

Both blocking locks *and* spinlocks are useful!

- Spinlocks on multiprocessors do not require a context switch.
- If locks are held a short time, and threads are running on multiple CPUs, then spinlocks are the most efficient choice.
  - In this scenario, a thread will only spin a few times before a thread scheduled on another CPU releases the lock.
  - The short "hold time" suggests that the lock holder is probably running. **Why?**
  - **Is this still true on a uniprocessor?**
- On the other hand, a spinning thread on an uniprocessor will have to be preempted before another thread is given an opportunity to release.

- On a ***uniprocessor***, blocking locks are ***always*** better
- On a ***multiprocessor***, spinlocks are better for ***short*** critical sections.
  - For long-held locks, spinning would waste a lot of CPU time.

A ***two-phase*** lock (in Linux/glibc's NPTL lib)

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
  if (atomic_bit_test_set (mutex, 31) == 0) ★
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) { ★
          atomic_decrement (mutex);
          return;
      }
      /* We have to wait now. First make sure the futex value
         we are monitoring is truly negative (i.e. locked). */
      v = *mutex;
      if (v >= 0)
        continue; ★
      futex_wait (mutex, v);
  }
}

void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 if and only if
    there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
```

- The top bit (31^{st}) is set if the lock is acquired.
- Lower 0-30 bits count the number of waiting threads.
- If there is no contention, lock and unlock are very fast (just one atomic op).
- Otherwise use the futex.

★Check the lock at least three times before blocking with futex.

Recap

- Hardware support for atomicity:
  - Disable interrupts
  - ***Test and set***
  - ***Compare and swap***
  - ***Fetch and add***
  - ***Load-linked*** & ***Store-conditional***
- Various lock implementations
  - Spinlock
  - Ticket lock
  - Yielding lock
  - Queuing locks
    - ***Park/unpark*** on Solaris
    - ***Futex*** on Linux

- Sophisticated locks can be more ***fair*** and avoid starvation, but they can add unnecessary context-switch overhead on multiprocessors.
- ***Two-phase locks*** try to combine the best of both approaches.
- OS scheduler and concurrent user code must coordinate for best performance.