IE 411: Operating Systems

Spinlocks

Recap: Locking

- Locking has two operations:
  - lock(): obtain the right to enter the critical section
  - unlock(): give up the right to be in the critical section
- Note: terminology can vary: acquire/release
- Building a lock needs some help from the hardware

Atomic Read-Modify-Write (RMW) primitives

- Any modern processor provides some form of atomic RMW instructions

Atomic Read-Modify-Write (RMW) primitives

- Any modern processor provides some form of atomic RMW instructions
- Examples:
  - Test & set

Atomic Read-Modify-Write (RMW) primitives

- Any modern processor provides some form of atomic RMW instructions
- Examples:
  - Test & set
  - Exchange (or swap)

Atomic Read-Modify-Write (RMW) primitives

- Any modern processor provides some form of atomic RMW instructions
- Examples:
  - Test & set
  - Exchange (or swap)
  - Fetch and increment

Atomic RMW primitives

- `test-and-set(loc)`:
  - sets contents of memory location loc to 1 and returns its old value as a single atomic operation

Atomic RMW primitives

- `test-and-set(loc)`:
  - sets contents of memory location loc to 1 and returns its old value as a single atomic operation
- `xchg(loc, r)`:
  - atomically exchanges a value in a register r for a value in memory location loc

Atomic RMW primitives

- `test-and-set(loc)`:
  - sets contents of memory location loc to 1 and returns its old value as a single atomic operation
- `xchg(loc, r)`:
  - atomically exchanges a value in a register r for a value in memory location loc
- `fetch-and-add(loc)`
  - returns the value stored at `loc` and atomically increments it

Atomic RMW primitives

- `test-and-set(loc)`:
  - sets contents of memory location loc to 1 and returns its old value as a single atomic operation
- `xchg(loc, r)`:
  - atomically exchanges a value in a register r for a value in memory location loc
- `fetch-and-add(loc)`
  - returns the value stored at `loc` and atomically increments it

Atomic RMW primitives

- `test-and-set(loc)`:
  - sets contents of memory location loc to 1 and returns its old value as a single atomic operation
- `xchg(loc, r)`:
  - atomically exchanges a value in a register r for a value in memory location loc
- `fetch-and-add(loc)`
  - returns the value stored at `loc` and atomically increments it
- Note that `xchg` is more general than `test-and-set`

A first lock: Spinlock

- Assume a shared variable `flag`
- The value of `flag` defines the state of the lock
  - 0 indicates available (unlocked, free)
  - 1 indicates acquired (locked, held)

A first lock: Spinlock

- Assume a shared variable `flag`
- The value of `flag` defines the state of the lock
  - 0 indicates available (unlocked, free)
  - 1 indicates acquired (locked, held)
- To acquire the lock
  - Set a register to 1 and atomically swap the contents of register and `flag`

A first lock: Spinlock

- Assume a shared variable `flag`
- The value of `flag` defines the state of the lock
  - 0 indicates available (unlocked, free)
  - 1 indicates acquired (locked, held)
- To acquire the lock
  - Set a register to 1 and atomically swap the contents of register and `flag`
  - What will be the outcome?

A first lock: Spinlock

- New value in register determines success in getting lock
  - 0 if you succeeded in setting the lock (you were first)
  - 1 if the lock was already held by someone else, so try again

A first lock: Spinlock

- New value in register determines success in getting lock
  - 0 if you succeeded in setting the lock (you were first)
  - 1 if the lock was already held by someone else, so try again
- note: a thread busy-waits, or spins, for lock to be released

A first lock: Spinlock

- How do we implement lock release?
  - simply write a 0 to `flag`

Spinlock: XCHG implementation

- pseudo-C code for the `xchg` instruction:

```
int xchg(int *addr, int newval) {
    // start of atomic segment
    int old = *addr;
    *addr = newval;
    // end of atomic segment
    return old;
}
```

  - return what was pointed to by `addr`
  - at the same time, store `newval` into `addr`

Spinlock: XCHG implementation

- pseudo-C code for the `xchg` instruction:

```
int xchg(int *addr, int newval) {
    // start of atomic segment
    int old = *addr;
    *addr = newval;
    // end of atomic segment
    return old;
}
```

  - return what was pointed to by `addr`
  - at the same time, store `newval` into `addr`
- note: the above code is for illustrative purposes only
  - compiling it will not guarantee atomicity

Spinlock: XCHG implementation

```
typedef struct {
    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 (xchg(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

Evaluating our spinlock

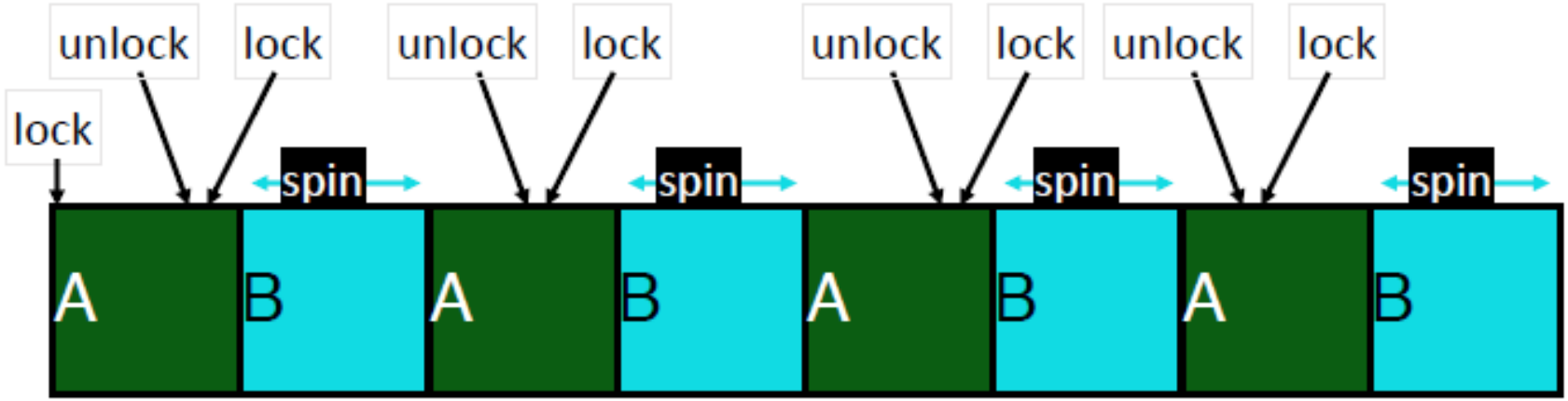
- Desirable properties
  - Mutual exclusion: only one thread can acquire lock at a time
  - Progress: whenever lock is available some thread will get it
  - Bounded wait: a waiting thread will eventually get the lock

Evaluating our spinlock

- Desirable properties
  - Mutual exclusion: only one thread can acquire lock at a time
  - Progress: whenever lock is available some thread will get it
  - Bounded wait: a waiting thread will eventually get the lock
- Which property is NOT satisfied by our lock impl?

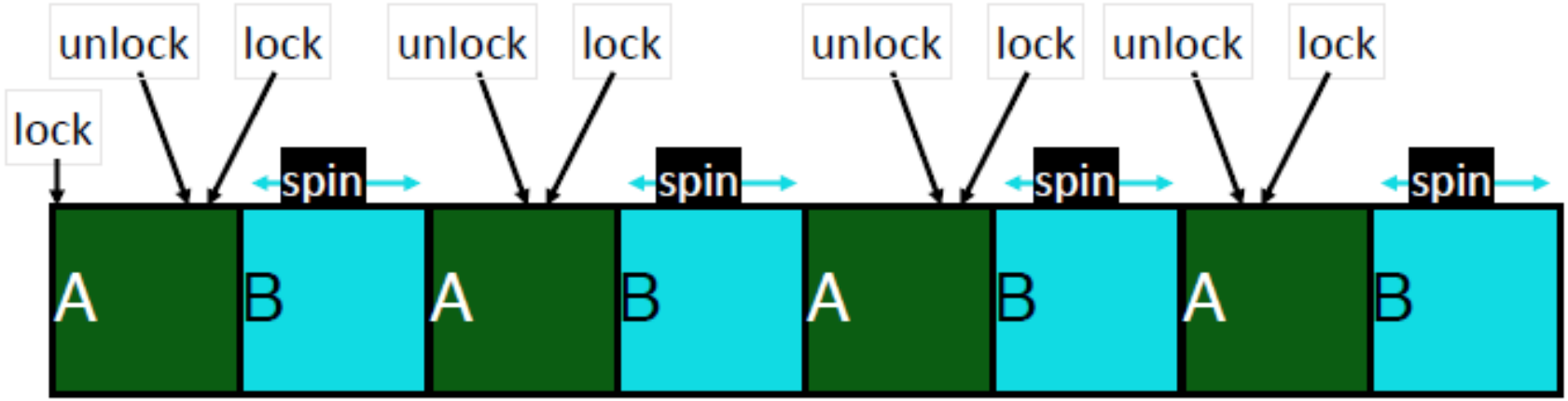
Evaluating our spinlock

- Our lock impl does not ensure bounded waiting
  - Thread B may wait indefinitely while thread A repeatedly acquires and releases the lock



Evaluating our spinlock

- Our lock impl does not ensure bounded waiting
  - Thread B may wait indefinitely while thread A repeatedly acquires and releases the lock



- note: scheduler is ignorant of locks so it may run B instead of A even though B is waiting for a lock that is held by A

Ticket spinlock: waiting for turn

- Ticket lock employs the same concept as e.g., banks do to serve their customers in the order of arrival
- On arrival customers draw a ticket from a ticket dispenser which hands out tickets with increasing numbers

Ticket spinlock: waiting for turn

- Ticket lock employs the same concept as e.g., banks do to serve their customers in the order of arrival
- On arrival customers draw a ticket from a ticket dispenser which hands out tickets with increasing numbers
- A screen displays the ticket number served next
- The customer holding the ticket with the number currently displayed on the screen is served next

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served
- To acquire the lock
  - grab a ticket using `fetch-and-add`
  - spin until thread's ticket value is called

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served
- To acquire the lock
  - grab a ticket using `fetch-and-add`
  - spin until thread's ticket value is called
- Why atomic increment is necessary?

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served
- To acquire the lock
  - grab a ticket using `fetch-and-add`
  - spin until thread's ticket value is called
- Why atomic increment is necessary?
  - two threads should never get the same ticket

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served
- To release lock, thread that holds the lock increments `turn` by 1
  - `turn = turn + 1`

Ticket spinlock: implementation

- Need two variables to hold the lock state
  - `ticket = 0;` // ticket number handed out to the next arriving thread
  - `turn = 0;` // ticket number currently served
- To release lock, thread that holds the lock increments `turn` by 1
  - `turn = turn + 1`
  - just a normal add operation (non-atomic)

recall: fetch-and-add (atomic increment)

- pseudo-C code for the `fetch-and-add` instruction:

```
int fetch-and-add(int *addr) {
    // start of atomic segment
    int old = *addr;
    *addr = old + 1;
    // end of atomic segment
    return old;
}
```

  - increment the value pointed to by `addr`
  - at the same time, return the previous value

recall: fetch-and-add (atomic increment)

- pseudo-C code for the `fetch-and-add` instruction:

```
int fetch-and-add(int *addr) {
    // start of atomic segment
    int old = *addr;
    *addr = old + 1;
    // end of atomic segment
    return old;
}
```

  - increment the value pointed to by `addr`
  - at the same time, return the previous value
- note: the above code is for illustrative purposes only
  - compiling it will not guarantee atomicity

Ticket spinlock: fetch-and-add implementation

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

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

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

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

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0
T1 calls unlock	3		1

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0
T1 calls unlock	3		1
T2 enters CS	3		1

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0
T1 calls unlock	3		1
T2 enters CS	3		1
T2 calls unlock	3		2

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0
T1 calls unlock	3		1
T2 enters CS	3		1
T2 calls unlock	3		2
T3 enters CS	3		2

Ticket spinlock example

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

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

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

	ticket	myturn	turn
Initial	0		0
T1 calls lock	1	0	0
T1 enters CS	1		0
T2 calls lock	2	1	0
T3 calls lock	3	2	0
T1 calls unlock	3		1
T2 enters CS	3		1
T2 calls unlock	3		2
T3 enters CS	3		2
T3 calls unlock	3		3

Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU

Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU
- running: executing on the CPU
  - is the thread that currently controls the CPU

Thread states

- ready: waiting to be assigned to CPU
  - could run, but another thread has the CPU
- running: executing on the CPU
  - is the thread that currently controls the CPU
- waiting: waiting for an event, e.g. I/O
  - cannot make progress until event happens

Spinlock performance

- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again

Spinlock performance

- When context switching occurs inside a CS, “other” threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later

Spinlock performance

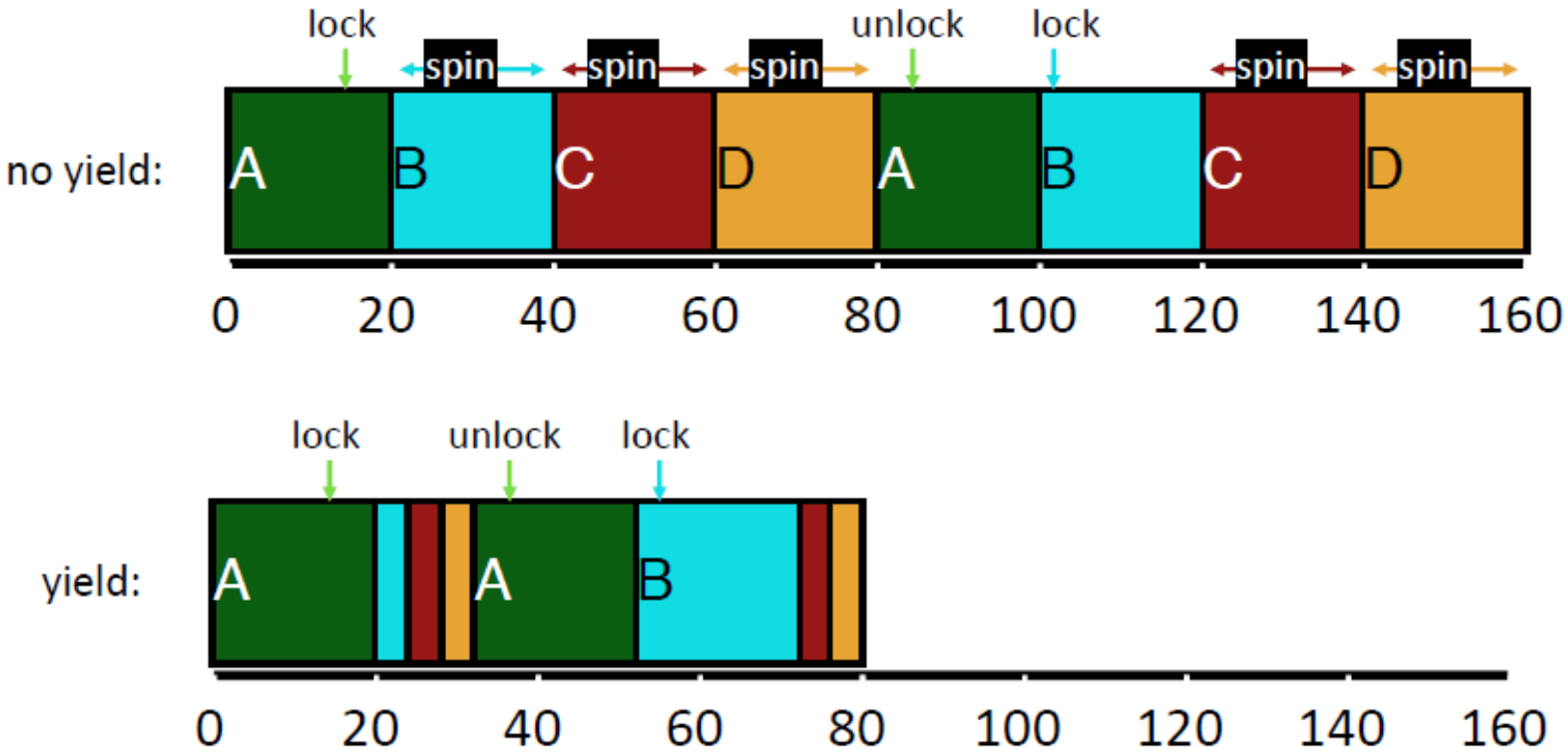
- When context switching occurs inside a CS, "other" threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  - `yield` system call moves calling thread from running to ready state

Spinlock performance

- When context switching occurs inside a CS, "other" threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  - `yield` system call moves calling thread from running to ready state

```
void lock(lock_t *lock) {
    while (xchg(&lock->flag, 1) == 1)
        yield(); // give up the CPU
}
```

Why is yield() useful?



Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)

Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)
  - better than spinlock which wastes 99 time slices spinning

Context switching

- How about 100 threads on one CPU?
- If one thread acquires the lock and is preempted before releasing it, the other 99 will call lock(), find the lock held, and yield the CPU (99 context switches)
  - better than spinlock which wastes 99 time slices spinning
- Even with yield, high context switch cost