CS 423
Operating System Design: Locks
Mar 05

Ram Kesavan

Slide ack: Prof. Shivaram Venkataraman (Wisconsin)

Announcements

MP2 out; Due (3/23) 11:59 pm CT
MP1 20% students 1:1s with TA on-going
    Grades finalized by next week

Mid-term online survey: next week
Midterms: **Mar 26th** (Piazza announcement)
    Last Name starts with A-L: DCL 1310
    Last Name starts with M-Z: Everitt Lab 2310

Mar 12 lecture: handled by TAs; I'm traveling

AGENDA / LEARNING OUTCOMES

Last Lecture:

- Motivation for Concurrency
- Need for synchronization primitives
- Critical sections & other models

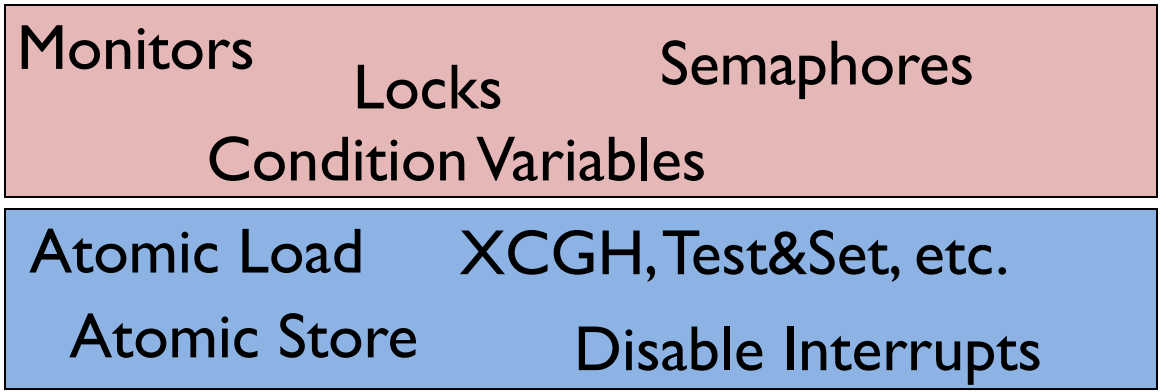
Today:

- Locks: how to use and how to implement
- HW atomic instructions to implement locks
- What do do when waiting for a lock
- If we have time, condition variables

Synchronization

Generic synchronization: for all models from previous slides

Goal: Learn how synchronization primitives are built (lib + OS)
With a little help from our friend…the hardware



LOCKS

Locks

Goal: Provide mutual exclusion (mutex)

Allocate and Initialize

- pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

Acquire

- Acquire exclusive access to lock
- Wait if lock is not available; some other thread has it
- Spin or block (relinquish CPU) while waiting
- pthread_mutex_lock(&mylock);

Release

- Release exclusive access to lock; let another thread at it
- pthread_mutex_unlock(&mylock);

LOCK IMPLEMENTATION GOALS

Goals:

- *Mutual exclusion*
  Only one thread in critical section at a time
- *Progress* (deadlock-free)
  If many simultaneous acquirers, must allow one to proceed
- *Bounded Waits* (starvation-free)
  Must eventually allow each waiting thread to enter

Fairness: Each acquirer waits for same amount of time (approx fifo)

Performance: CPU is not wasted unnecessarily

IMPLEMENT LOCKS BY DISABLING INTERRUPTS

Turn off interrupts for critical sections
- Prevent timer interrupt, so no context switch possible
- Don't make any blocking system calls

Code between interrupts executes atomically

Can this work?

```
acquire(&lock);

// critical
//   section
//     code
//       here

release(&lock);
```

```
void acquire(lockT *l) {
    disableInterrupts();
}

void release(lockT *l) {
    enableInterrupts();
}
```

IMPLEMENT LOCKS BY DISABLING INTERRUPTS

Turn off interrupts for critical sections
- Prevent timer interrupt, so no context switch possible
- Don't make any blocking system calls

Code between interrupts executes atomically

```
void acquire(lockT *l) {
    disableInterrupts();
}
```

```
void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages?

- Only works on uniprocessors
- Process may hold onto CPU for too long
- Turning off interrupts for too long is really bad
- Cannot perform any blocking work (network/file IO)

IMPLEMENT LOCKS w/ Load+Store

Use a single **shared** lock variable

```
// shared variable
boolean lock = false;
void acquire(boolean *lock) {
    while (*lock); // spin…
    *lock = true;
}
void release(boolean *lock) {
    *lock = false;
}
```

Does this work?

IMPLEMENT LOCKS w/ Load+Store

Use a single **shared** lock variable

```
// shared variable
boolean lock = false;
void acquire(boolean *lock) {
    while (*lock); // spin…
    *lock = true;
}
void release(boolean *lock) {
    *lock = false;
}
```

Problems

Correctness

1. Say T1's acquire() is interrupted just after it gets past the while check
2. T2's acquire() runs through and sets lock to true, and is interrupted before release()
3. T1 resumes, also sets lock to true and enters critical section

Performance:

spin-waiting is wasted CPU cycles

General Principle: Spinning on a lock is typically wasteful

Implementing Synchronization

Disabling Interrupts

- Used by OS in very special circumstances
  - Eg. within an interrupt handler

Locking using loads/stores (Peterson's algorithm)

- Correct but it suffers from wasteful spin-waiting

But locks today are built using support from hardware (and OS)

XCHG: Atomic Exchange

Atomic instructions that allow a test+set atomically: XCHG, CAS

```
// pseudo-code for xchg (load + store in a single instruction!)
// return what was pointed to by addr
// also, store newval into addr
int xchg(int *addr, int newval) {
        int old = *addr;
        *addr = newval;
        return old;
}
```

CAS

Compare: check value, if success then Swap: with new value

LOCK Implementation with XCHG

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

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

// Use the XCHG instruction
void acquire(lock_t *lock) {
    ????; ← while (xchg(&lock->flag, 1) == 1);
    // spin-wait (do nothing)
}

void release(lock_t *lock) {
    lock->flag = ??; ← lock->flag = 0;
}
```

int xchg(int *addr, int newval)

Using Atomic Instructions

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Test – return old value

Set – set the passed in value

HW does them atomically!

Other Atomic HW Instructions

```
int CompareAndSwap(int *addr, int expected, int new) {
    int actual = *addr;
    if (actual == expected)
        *addr = new;
    return actual;       // OR return true if comparison succeeded
}
```

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

LOCK IMPLEMENTATION GOALS

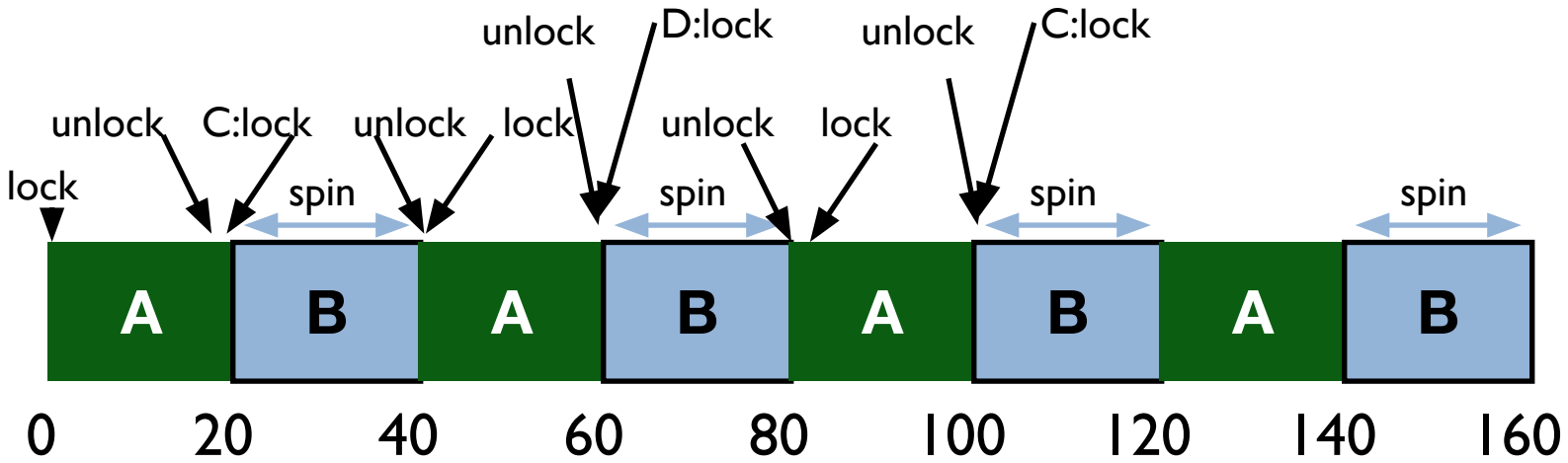
Correctness

- *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: Does CPU get wasted?

BASIC SPINLOCKS ARE UNFAIR



Scheduler is totally unaware of locks/unlocks.

B is unlucky - not guaranteed 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);
    while (lock->turn != myturn);
        // spin
}

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

SPINLOCK PERFORMANCE

Not great, because…

- locks may be held for a long time

    Large critical sections

- Spinning is wasted CPU cycles

POP QUIZ

https://forms.gle/sWBjTzB42QgCBRht6

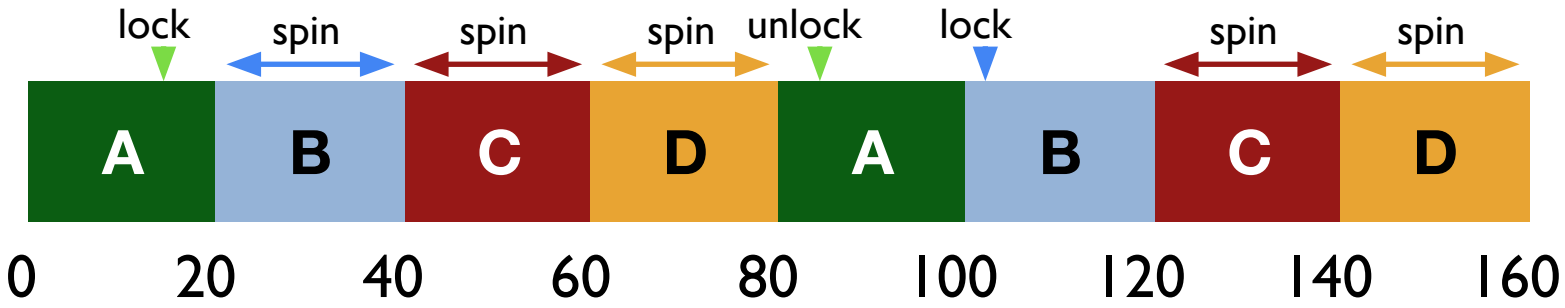
HOLDING A LOCK: USER vs KERNEL

Question 1:
Can a thread running in user mode acquire a mutex, and then get context-switched or interrupted while in its critical section?
If so, what happens?

Question 2:
Ditto for a thread running in kernel/privileged mode?

Question 3:
What if a kernel thread acquires a mutex, is interrupted, and that interrupt service routine tries to acquire that same mutex?

CPU SCHEDULER IS IGNORANT



What if thread A is de-scheduled while in its critical section?

CPU scheduler may run **B, C, D** instead of **A**

even though **B, C, D** are waiting for **A**

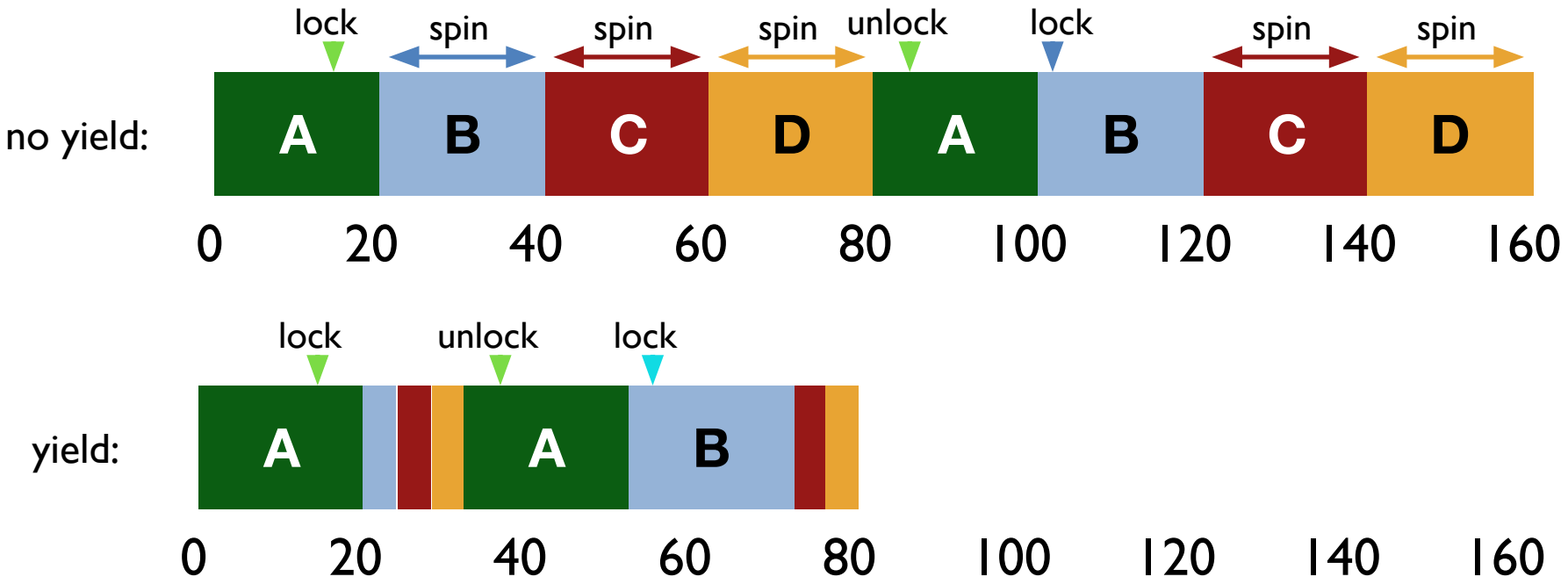
TICKET LOCK WITH YIELD

```
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) {
    lock->turn++;
}
```

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

Yield Instead of Spin



Lock Implementation: Block when Waiting

Remove waiting threads from scheduler runnable queue

Scheduler won't run any waiting thread

- CPU cycles saved

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

- park(): Solaris puts thread to SLEEPING state
- unpark(): Solaris transitions thread to READY state

Support in Linux: futex (fast userspace mutex)

Block when Waiting

```
typedef struct {
   bool lock = false;
   bool guard = false;
   queue_t q;
} LockT;
```

```
acquire(&lock);

// critical section

release(&lock);
```

```
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
            qadd(l->q, tid); // use gettid()
            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)); // wakeup
    l->guard = false;
}
```

Block when Waiting

(a) Why is **guard** used? What does it protect?

(b) Why okay to **spin** on guard?

(c) In release(), why not set lock=false when unpark?

```
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)); // wakeup
    l->guard = false;
}
```

Block when Waiting

(d) What if the order of guard=false and park() is flipped?

(e) Is there a case where a thread may indefinitely sleep?

```
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)); // wakeup
    l->guard = false;
}
```

RACE CONDITION

T1 (inside acquire)	T2 (inside release)
`qadd(l->q, T1);`	
`l->guard = false;` INTERRUPT	
	`XCHG succeeds;` `l->q is not empty // has T1`
	`unpark(T1);` `lost as T1 is RUNNING or READY`
`park(T1);` `T1 put to SLEEPING; sleeps forever`	

FINAL VERSION

```
typedef struct {
    bool lock = false;
    bool guard = false;
    queue_t q;
} LockT;
```

setpark() fixes race condition:
park() won't block if unpark()
occurred after setpark()

```
void acquire(LockT *l) {
    while (XCHG(&l->guard, true));
    if (l->lock) {
            qadd(l->q, tid);
            setpark();     ←– NEW CODE
            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)); // wakeup
    l->guard = false;
}
```

Spin-Waiting vs Blocking

Optimal approach depends on the circumstance

Uniprocessor

- Waiting thread should always relinquish processor
  - So that process (with lock) is scheduled
- Associate queue of waiters with each lock

Multiprocessor

- Thread holding lock may be running, may release lock soon
- Spin vs block: depends on how long (t) before lock is released
  - soon (small t) -> Spin
  - not so soon (large t)-> Block
  - “soon” is relative to context-switch cost, C

When to Spin-Wait vs Block?

How much CPU time is wasted when spin-waiting?
How much wasted when blocking?
What is the best action when t < C?
When t > C?
Problem: Requires knowledge of future

Hybrid approach: Two-phase Locks (not the distributed systems version!)

- Spin for some time (< C)
- Block after that

Synchronization

Generic synchronization: for all models from previous slides

Goal: Learn how synchronization primitives are built (lib + OS)
With a little help from our friend…the hardware