

# Synchronization II

**Hakim Weatherspoon**  
**CS 3410, Spring 2012**  
Computer Science  
Cornell University

P&H Chapter 2.11

# Administrivia

---

## Pizza party: PA3 Games Night

- Friday, April 27<sup>th</sup>, 5:00-7:00pm
- Location: Upson B17

## Prelim3 Review

- Today, Tuesday, April 24<sup>th</sup>, 5:30-7:30pm
- Location: Hollister 110

## Prelim 3

- Thursday, April 26<sup>th</sup>, 7:30pm
- Location: Olin 155

## PA4: Final project out next week

- Demos: May 14-16
- ***Will not be able to use slip days***

# Goals for Today

---

## Synchronization

- Threads and processes
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
  - Cache coherency causes problems
  - Locks + barriers
- Language level synchronization

# Synchronization

---

Two processors sharing an area of memory

- P1 writes, then P2 reads
- Data race if P1 and P2 don't *synchronize*
  - Result depends of order of accesses

Hardware support required

- Atomic read/write memory operation
- No other access to the location allowed between the read and write

Could be a single instruction

- E.g., atomic swap of register  $\leftrightarrow$  memory (e.g. AT&S, BTS; x86)
- Or an atomic pair of instructions (e.g. LL and SC; MIPS)

# Synchronization in MIPS

---

Load linked:            LL rt, offset(rs)

Store conditional: SC rt, offset(rs)

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic swap (to test/set lock variable)

```
try:  MOVE $t0, $s4      ; copy exchange value
      LL   $t1, 0($s1)   ; load linked
      SC   $t0, 0($s1)   ; store conditional
      BEQZ $t0, try      ; branch store fails
      MOVE $s4, $t1     ; put load value in $s4
```

# Programming with Threads

---

Need it to exploit multiple processing units

...to provide interactive applications

...to parallelize for multicore

...to write servers that handle many clients

Problem: hard even for experienced programmers

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Needed: synchronization of threads

# Programming with Threads

---

Concurrency poses challenges for:

## Correctness

- Threads accessing shared memory should not interfere with each other

## Liveness

- Threads should not get stuck, should make forward progress

## Efficiency

- Program should make good use of available computing resources (e.g., processors).

## Fairness

- Resources apportioned fairly between threads

# Two threads, one counter

---

Example: Web servers use concurrency

Multiple threads handle client requests in parallel.

Some shared state, e.g. hit counts:

- each thread increments a shared counter to track number of hits

```
...  
hits = hits + 1;  
...
```

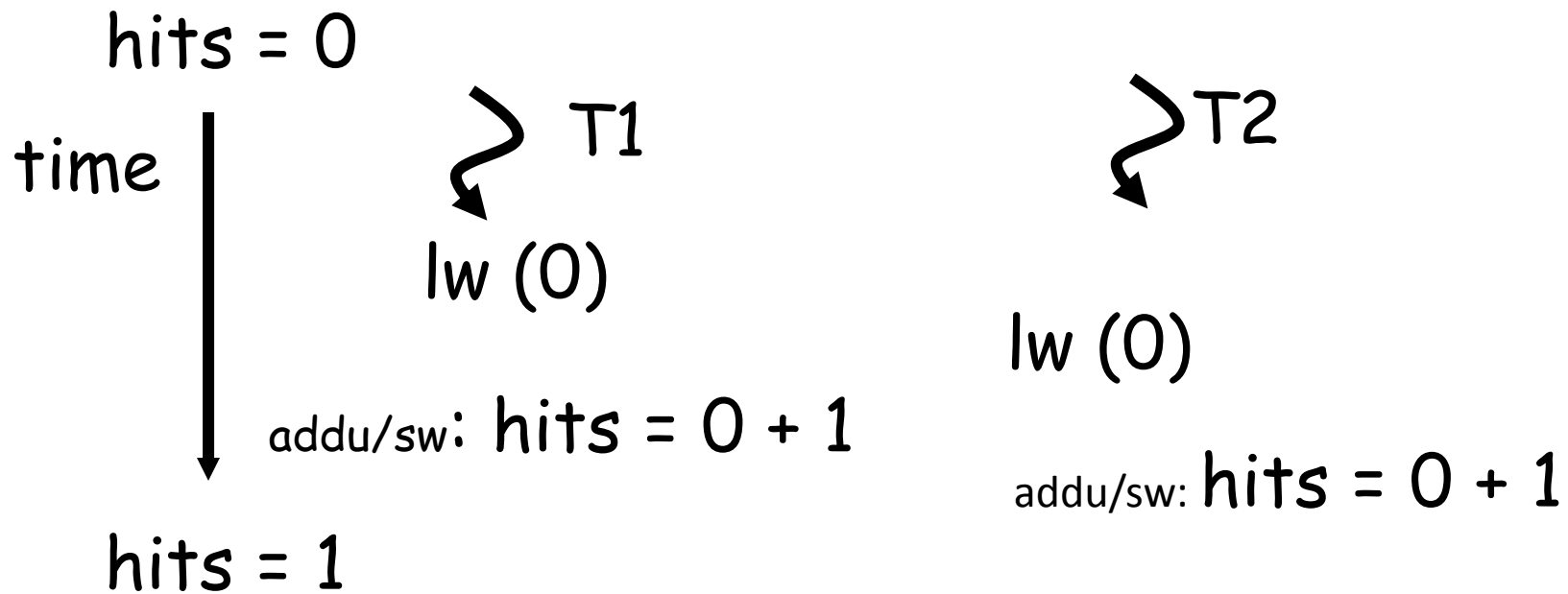
```
...  
LW R0, hitsloc  
ADDI R0, r0, 1  
SW R0, hitsloc
```

What happens when two threads execute concurrently?



# Shared counters

Possible result: lost update!



Timing-dependent failure  $\Rightarrow$  race condition

- hard to reproduce  $\Rightarrow$  Difficult to debug

# Race conditions

---

Def: timing-dependent error involving access to shared state

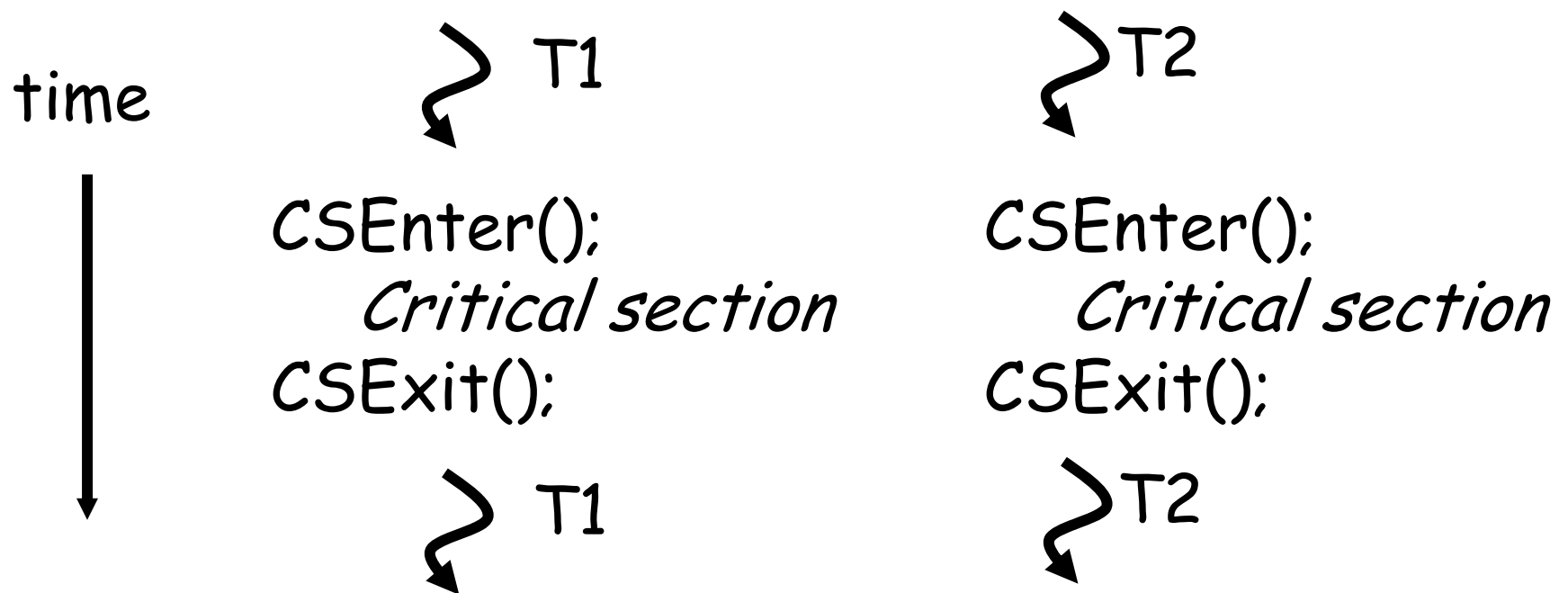
- Whether it happens depends on how threads scheduled: who wins “races” to instruction that updates state vs. instruction that accesses state
- Races are intermittent, may occur rarely
  - Timing dependent = small changes can hide bug
- A program is correct *only* if *all possible* schedules are safe
  - Number of possible schedule permutations is huge
  - Need to imagine an adversary who switches contexts at the worst possible time

# Critical sections

---

To eliminate races: use *critical sections* that only one thread can be in

- Contending threads must wait to enter



# Mutexes

---

Critical sections typically associated with mutual exclusion locks (*mutexes*)

Only one thread can hold a given mutex at a time

Acquire (lock) mutex on entry to critical section

- Or block if another thread already holds it

Release (unlock) mutex on exit

- Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(&m);  
pthread_mutex_lock(&m);      pthread_mutex_lock(&m);  
    hits = hits+1;           hits = hits+1;  
pthread_mutex_unlock(&m);    pthread_mutex_unlock(&m);
```

↪ T1

↪ T2

# Mutexes

---

Q: How to implement critical section in code?

A: Lots of approaches....

Mutual Exclusion Lock (mutex)

lock(m): wait till it becomes free, then lock it

unlock(m): unlock it

```
safe_increment() {  
    pthread_mutex_lock(&m);  
    hits = hits + 1;  
    pthread_mutex_unlock(&m)  
}
```

---

# Hardware Support for Synchronization

# Synchronization in MIPS

---

Load linked:           LL rt, offset(rs)

Store conditional: SC rt, offset(rs)

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic swap (to test/set lock variable)

```
try:  MOVE $t0, $s4      ; copy exchange value
      LL   $t1, 0($s1)   ; load linked
      SC   $t0, 0($s1)   ; store conditional
      BEQZ $t0, try      ; branch store fails
      MOVE $s4, $t1      ; put load value in $s4
```

# Mutex from LL and SC

---

Linked load / Store Conditional

```
mutex_lock(int *m) {  
    while(test_and_test(m)){}  
}
```

```
int test_and_set(int *m) {  
    old = *m;  
    *m = 1;  
    return old;  
}
```



# Mutex from LL and SC

---

Linked load / Store Conditional

```
mutex_lock(int *m) {  
    while(test_and_test(m)){  
    }  
}
```

```
int test_and_set(int *m) {  
    LI $t0, 1  
    LL $t1, 0($a0)  
    SC $t0, 0($a0)  
    MOVE $v0, $t1  
}
```

# Mutex from LL and SC

---

Linked load / Store Conditional

```
mutex_lock(int *m) {  
    test_and_set:  
        LI $t0, 1  
        LL $t1, 0($a0)  
        BNEZ $t1, test_and_set  
        SC $t0, 0($a0)  
        BEQZ $t0, test_and_set  
}
```

```
mutex_unlock(int *m) {  
    *m = 0;  
}
```

# Mutex from LL and SC

---

Linked load / Store Conditional

```
mutex_lock(int *m) {
    test_and_set:
        LI $t0, 1
        LL $t1, 0($a0)
        BNEZ $t1, test_and_set
        SC $t0, 0($a0)
        BEQZ $t0, test_and_set
}

mutex_unlock(int *m) {
    SW $zero, 0($a0)
}
```

# Alternative Atomic Instructions

---

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)

# Alternative Atomic Instructions

---

## Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional  
(MIPS, ARM, PowerPC, DEC Alpha, ...)

# Synchronization

---

## Synchronization techniques

### clever code

- must work despite adversarial scheduler/interrupts
- used by: hackers
- also: noobs

### disable interrupts

- used by: exception handler, scheduler, device drivers, ...

### disable preemption

- dangerous for user code, but okay for some kernel code

### mutual exclusion locks (mutex)

- general purpose, except for some interrupt-related cases

---

Using synchronization primitives to build  
concurrency-safe datastructures

# Broken invariants

Access to shared data must be synchronized

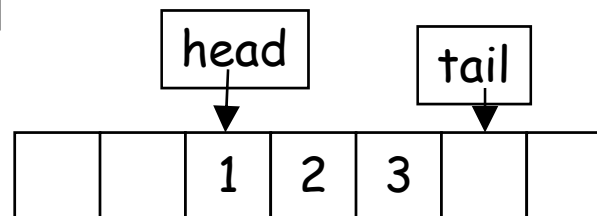
- goal: enforce datastructure invariants

```
// invariant:
```

```
// data is in A[h ... t-1]
```

```
char A[100];
```

```
int h = 0, t = 0;
```



```
// producer: add to list tail // consumer: take from list head
```

```
void put(char c) {
```

```
    A[t] = c;
```

```
    t++;
```

```
}
```

```
char get() {
```

```
    while (h == t) { };
```

```
    char c = A[h];
```

```
    h++;
```

```
    return c;
```

```
}
```



# Protecting an invariant

```
// invariant: (protected by m)
// data is in A[h ... t-1]
pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;

// producer: add to list tail // consumer: take from list head
void put(char c) {                char get() {
    pthread_mutex_lock(m);        pthread_mutex_lock(m);
    A[t] = c;                    while(h == t) {}
    t++;                          char c = A[h];
    pthread_mutex_unlock(m);      h++;
}                                  pthread_mutex_unlock(m);
                                  return c;
}
```

Rule of thumb: all updates } that can affect  
invariant become critical sections

# Guidelines for successful mutexing

---

Insufficient locking can cause races

- Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

```
P1: lock(m1);    P2: lock(m2);  
    lock(m2);    lock(m1);
```

- know why you are using mutexes!
- acquire locks in a consistent order to avoid cycles
- use lock/unlock like braces (match them lexically)
  - lock(&m); ...; unlock(&m)
  - watch out for return, goto, and function calls!
  - watch out for exception/error conditions!

---

# Cache Coherency causes yet more trouble

# Remember: Cache Coherence

---

Recall: Cache coherence defined...

Informal: Reads return most recently written value

Formal: For concurrent processes  $P_1$  and  $P_2$

- $P$  writes  $X$  before  $P$  reads  $X$  (with no intervening writes)  
⇒ read returns written value
- $P_1$  writes  $X$  before  $P_2$  reads  $X$   
⇒ read returns written value
- $P_1$  writes  $X$  and  $P_2$  writes  $X$   
⇒ all processors see writes in the same order  
– all see the same final value for  $X$

# Relaxed consistency implications

---

Ideal case: sequential consistency

- Globally: writes appear in interleaved order
- Locally: other core's writes show up in program order

In practice: not so much...

- write-back caches → sequential consistency is tricky
- writes appear in semi-random order
- locks alone don't help

\* MIPS has sequential consistency; Intel does not

# Acquire/release

---

## Memory Barriers and Release Consistency

- Less strict than sequential consistency; easier to build

### One protocol:

- Acquire: lock, and force subsequent accesses after
- Release: unlock, and force previous accesses before

P1: ...

```
    acquire(m);
```

```
    A[t] = c;
```

```
    t++;
```

```
    release(m);
```

P2: ...

```
    acquire(m);
```

```
    A[t] = c;
```

```
    t++;
```

```
    unlock(m);
```

Moral: can't rely on sequential consistency  
(so use synchronization libraries)

---

Are Locks + Barriers enough?

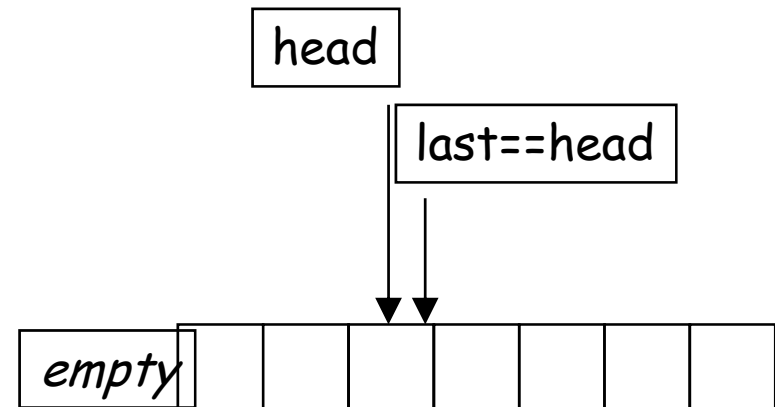
# Beyond mutexes

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    while(empty) {}  
    acquire(L);  
    char c = A[h];  
    h++;  
    release(L);  
    return c;  
}
```





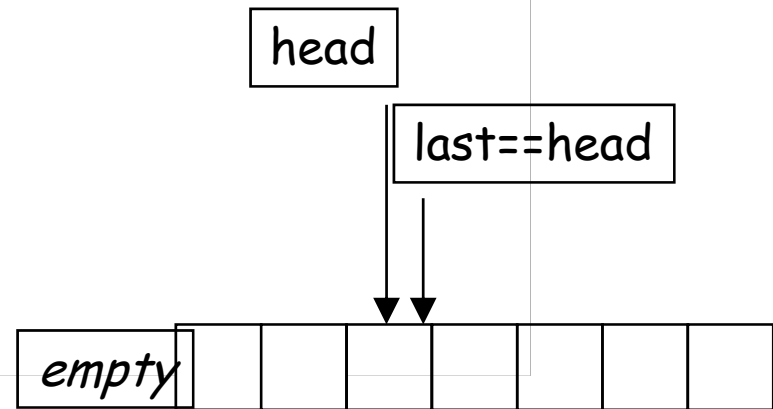
# Beyond mutexes

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    while (h == t) { };  
    acquire(L);  
    char c = A[h];  
    h++;  
    release(L);  
    return c;  
}
```



Dilemma: Have to check while holding lock,

# Beyond mutexes

---

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {  
    acquire(L);  
    while (h == t) { };  
    char c = A[h];  
    h++;  
    release(L);  
    return c;  
}
```

Dilemma: Have to check while holding lock,  
but cannot wait while hold lock

# Beyond mutexes

---

Writers must check for full buffer

& Readers must check if for empty buffer

- ideal: don't busy wait... go to sleep instead

```
char get() {
    do {
        acquire(L);
        empty = (h == t);
        if (!empty) {
            c = A[h];
            h++;
        }
        release(L);
    } while (empty);
    return c;
}
```

---

# Language-level Synchronization

# Condition variables

---

Use [Hoare] a condition variable to wait for a condition to become true (without holding lock!)

`wait(m, c) :`

- atomically release `m` and sleep, waiting for condition `c`
- wake up holding `m` sometime after `c` was signaled

`signal(c) :` wake up one thread waiting on `c`

`broadcast(c) :` wake up all threads waiting on `c`

POSIX (e.g., Linux): `pthread_cond_wait`,  
`pthread_cond_signal`, `pthread_cond_broadcast`

# Using a condition variable

`wait(m, c)` : release m, sleep until c, wake up holding m

`signal(c)` : wake up one thread waiting on c

```
cond_t *not_full = ...;
cond_t *not_empty = ...;
mutex_t *m = ...;
```

```
void put(char c) {
    lock(m);
    while ((t-h) % n == 1)
        wait(m, not_full);
    A[t] = c;
    t = (t+1) % n;
    unlock(m);
    signal(not_empty);
}
```

```
char get() {
    lock(m);
    while (t == h)
        wait(m, not_empty);
    char c = A[h];
    h = (h+1) % n;
    unlock(m);
    signal(not_full);
    return c;
}
```

# Monitors

---

A Monitor is a concurrency-safe datastructure, with...

- one mutex
- some condition variables
- some operations

All operations on monitor acquire/release mutex

- one thread in the monitor at a time

Ring buffer was a monitor

Java, C#, etc., have built-in support for monitors

# Java concurrency

---

## Java objects can be monitors

- “synchronized” keyword locks/releases the mutex
- Has one (!) builtin condition variable
  - `o.wait()` = `wait(o, o)`
  - `o.notify()` = `signal(o)`
  - `o.notifyAll()` = `broadcast(o)`
- Java `wait()` can be called even when mutex is not held. Mutex not held when awoken by `signal()`. Useful?



# More synchronization mechanisms

---

Lots of synchronization variations...

(can implement with mutex and condition vars.)

## Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

## Semaphores

- N threads can hold lock at the same time

Message-passing, sockets, queues, ring buffers, ...

- transfer data and synchronize

# Summary

---

Hardware Primitives: test-and-set, LL/SC, barrier, ...  
... used to build ...

Synchronization primitives: mutex, semaphore, ...  
... used to build ...

Language Constructs: monitors, signals, ...