

# Synchronization II

**Hakim Weatherspoon**

**CS 3410, Spring 2013**

Computer Science

Cornell University

P&H Chapter 2.11

# 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

## Next Goal

---

Understanding challenges of taking advantage of parallel processors and resources?

i.e. Challenges in parallel programming!

# 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:

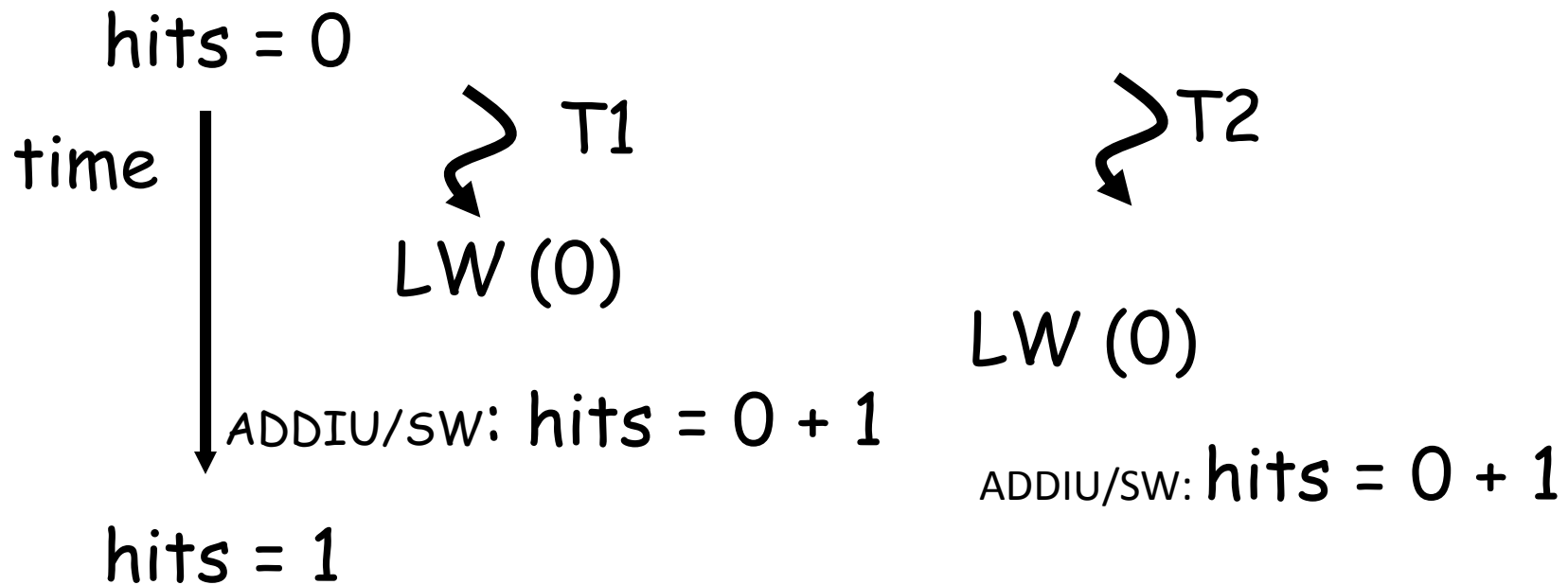
- each thread increments a shared counter to track number of hits
- |                         |                          |
|-------------------------|--------------------------|
| ...                     | ...                      |
| <b>hits = hits + 1;</b> | <b>LW R0, addr(hits)</b> |
| ...                     | <b>ADDI R0, r0, 1</b>    |
|                         | <b>SW R0, addr(hits)</b> |

What happens when two threads execute concurrently?

# Two threads, one counters

---

**Possible result: lost update!**



Timing-dependent failure  $\Rightarrow$  race condition

- **Very hard to reproduce  $\Rightarrow$  Difficult to debug**

# Race conditions

---

Def: timing-dependent error involving access to shared state

Whether a Race condition happens depends on

- how threads scheduled
- i.e. who wins “races” to instruction that updates state vs. instruction that accesses state

Challenges about Race conditions

- 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



# Takeaway

---

Need parallel abstraction like threads to take advantage of parallel resources like multicore.  
Writing parallel programs are hard to get right!  
Need to prevent data races, timing dependent updates that result in errors in programs.

# Next Goal

---

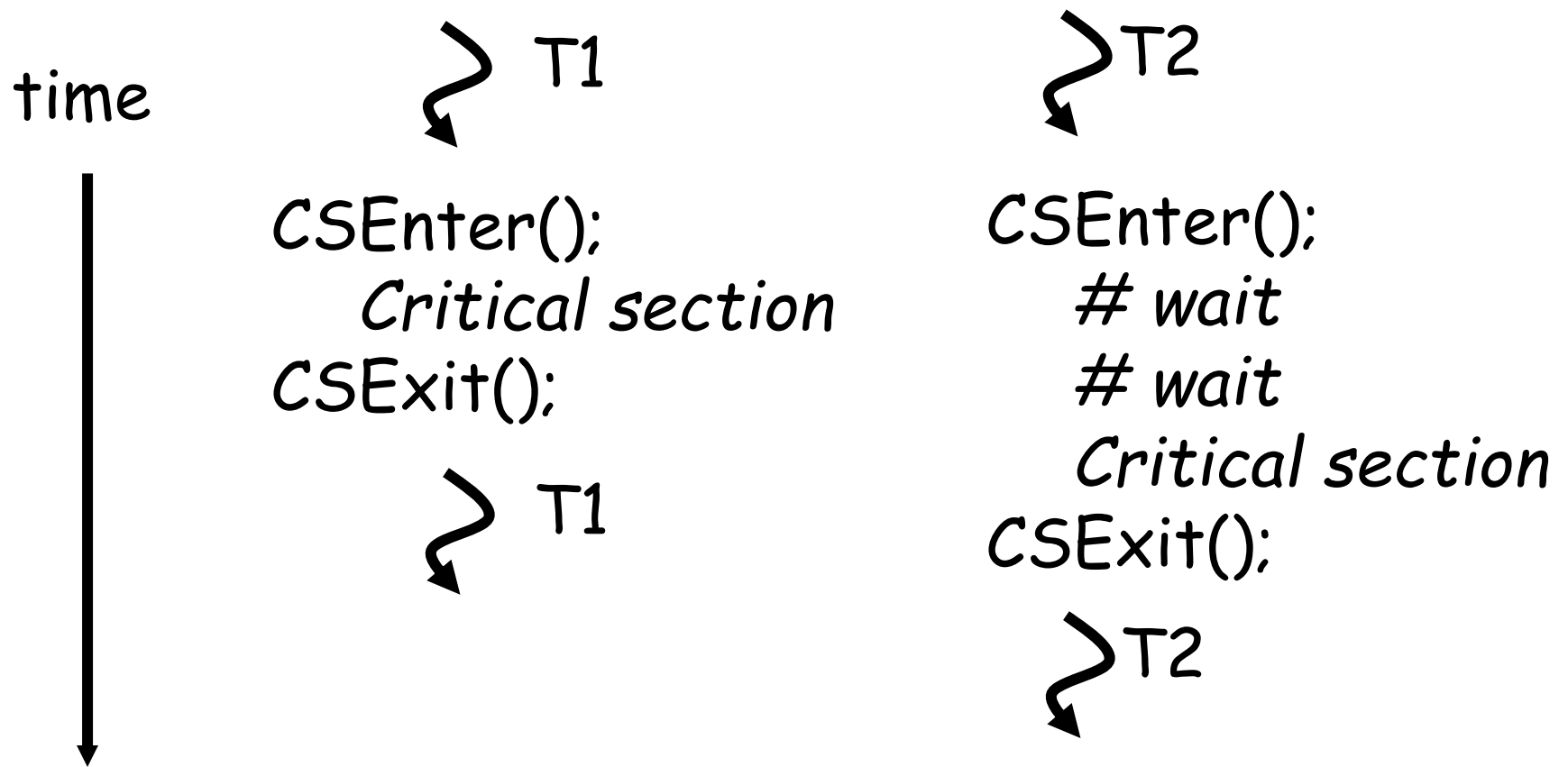
How to prevent data races and write correct parallel programs?

# 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;          # wait  
pthread_mutex_unlock(&m);  # wait  
    hits = hits+1;  
pthread_mutex_unlock(&m);  
    ↪ T1  
    ↪ T2
```

# Takeaway

---

Need parallel abstraction like threads to take advantage of parallel resources like multicore.

Writing parallel programs are hard to get right!

Need to prevent data races, timing dependent updates that result in errors in programs.

Need critical sections where prevent data races and write parallel safe programs. Mutex, mutual exclusion, can be used to implement critical sections, often implemented via a lock abstraction.

## Next Goal

---

How to implement mutex locks?

What are the hardware primitives?

Then, use these mutex locks to implement critical sections, and use critical sections to write parallel safe programs.

# Mutexes

---

Q: How to implement critical section in code?

# 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

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in \$t0, causing the code to try again.



# 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 incrementor

Time Step	Thread A	Thread B	Thread A \$t0	Thread B \$t0	Memory M[\$s0]
0					0
1	try: LL \$t0, 0(\$s0)	try: LL \$t0, 0(\$s0)			
2	ADDIU \$t0, \$t0, 1	ADDIU \$t0, \$t0, 1			
3	SC \$t0, 0(\$s0)	SC \$t0, 0 (\$s0)			
4	BEQZ \$t0, try	BEQZ \$t0, try			

# Mutex from LL and SC

---

Linked load / Store Conditional

`m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked`

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

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

# Mutex from LL and SC

---

Linked load / Store Conditional

m = 0;

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

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

# Mutex from LL and SC

---

Linked load / Store Conditional

```
m = 0;
```

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

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

# Mutex from LL and SC

---

Linked load / Store Conditional

m = 0;

```
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

m = 0;

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)

}

This is called a  
Spin lock  
Aka spin waiting



# Mutex from LL and SC

Linked load / Store Conditional

m = 0;

mutex\_lock(int \*m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							0
1	try: LI \$t0, 1	try: LI \$t0, 1					
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)					
3	BNEZ \$t1, try	BNEZ \$t1, try					
4	SC \$t0, 0(\$a0)	SC \$t0, 0 (\$a0)					
5	BEQZ \$t0, try	BEQZ \$t0, try					
6							

# Mutex from LL and SC

Linked load / Store Conditional

m = 0;

mutex\_lock(int \*m) {

Time Step	Thread A	Thread B	Thread A \$t0	Thread A \$t1	Thread B \$t0	Thread B \$t1	Mem M[\$a0]
0							1
1	try: LI \$t0, 1	try: LI \$t0, 1					
2							
3							
4							
5							
6							
7							
8							
9							



# 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

# Takeaway

---

Need parallel abstraction like threads to take advantage of parallel resources like multicore. Writing parallel programs are hard to get right! Need to prevent data races, timing dependent updates that result in errors in programs.

Need critical sections where prevent data races and write parallel safe programs. Mutex, mutual exclusion, can be used to implement critical sections, often implemented via a lock abstraction.

We need synchronization primitives such as LL and SC (load linked and store conditional) instructions to efficiently implement parallel and correct programs.

## Next Goal

---

How do we use synchronization primitives to build concurrency-safe data structure?

# Attempt#1: Producer/Consumer

---

Access to shared data must be synchronized

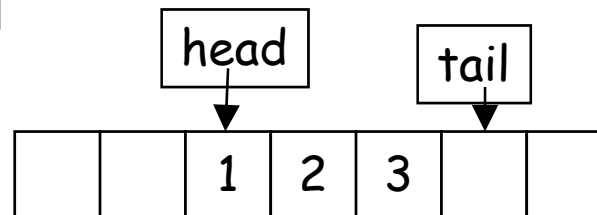
- goal: enforce datastructure invariants

// invariant:

// data is in  $A[h \dots t-1]$

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

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



// producer: add to list tail

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

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

```
    t = (t+1)%n;
```

```
}
```

# Attempt#1: Producer/Consumer

---

Access to shared data must be synchronized

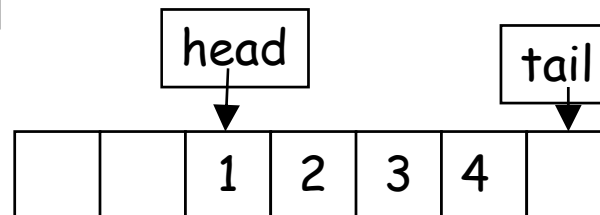
- goal: enforce datastructure invariants

// invariant:

// data is in  $A[h \dots t-1]$

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 = (t+1)%n;

}

char get() {

    while (h == t) { };

    char c = A[h];

    h = (h+1)%n;

    return c;

}

# Attempt#1: Producer/Consumer

Access to **shared data** must be synchronized

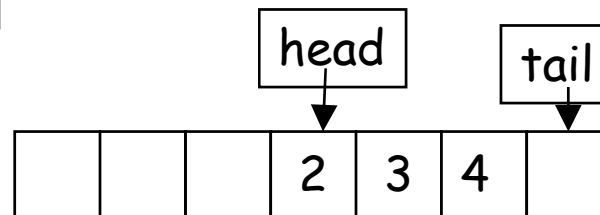
- goal: enforce datastructure invariants

// invariant:

// data is in  $A[h \dots t-1]$

```
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 = (t+1)%n;
```

```
}
```

```
char get() {
```

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

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

```
    h = (h+1)%n;
```

```
    return c;
```

Error: could miss an update to  $t$  or  $h$  due to lack of synchronization

Current implementation will **break invariant**:

only produce if not full and only consume if not empty

***Need to synchronize access to shared data***

## Attempt#2: Protecting an invariant

---

```
// invariant: (protected by mutex 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 = (t+1)%n;                 char c = A[h];
    pthread_mutex_unlock(m);      h = (h+1)%n;
}                                pthread_mutex_unlock(m);
                                return c;
}
```

Rule of thumb: all access and 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);	Circular Wait
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!

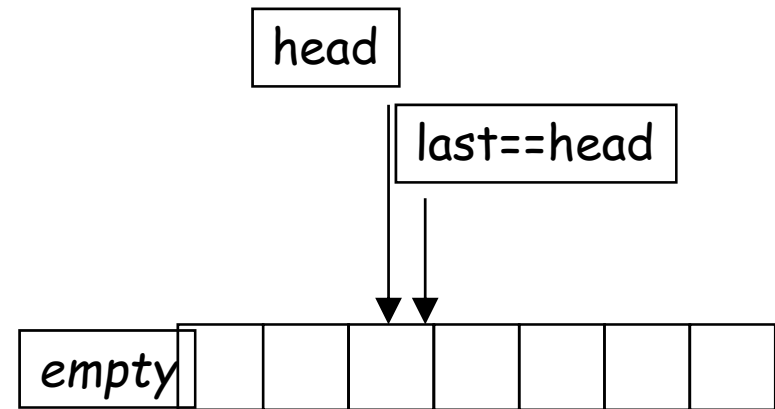
# Attempt#3: Beyond mutexes

---

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 = (h+1)%n;  
    release(L);  
    return c;  
}
```



## Attempt#3: 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 = (h+1)%n;  
    release(L);  
    return c;  
}
```

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

## Attempt#4: 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 = (h+1)%n;  
        }  
        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`

## Attempt#5: 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, ...

# Administrivia

---

Project3 *due next week*, Monday, April 22<sup>nd</sup>

- **Games night Friday, April 26<sup>th</sup>, 5-7pm. Location: B17 Upson**
- **Come, eat, drink, have fun and be merry!**

Prelim3 is *next week*, Thursday, April 25<sup>th</sup>

- Time and Location: 7:30pm in Phillips 101 and Upson B17
- Old prelims are online in CMS
- Prelim Review Session:

Monday, April 22, 6-8pm in B17 Upson Hall

Tuesday, April 23, 6-8pm in 101 Phillips Hall

Project4: Final project out next week

- Demos: May 14 and 15
- ***Will not be able to use slip days***

# Administrivia

---

## Next three weeks

- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

## Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due by May 15th