# Synchronization II

**Hakim Weatherspoon**

**CS 3410, Spring 2015**

Computer Science

Cornell University

P&H Chapter 2.11

# Administrivia

Project3 *due tomorrow*, Friday, April 24th

- **Games night Monday, May 4th, 5-7pm. Location: B17 Upson**
- **Come, eat, drink, have fun and be merry!**

Prelim2 is *next week*, Thursday, April 30th

- Time and Location: 7:30pm in Statler Auditorium
- Old prelims are online in CMS
- Prelim Review Session:

  Sunday, April 26, 7-9pm in B14 Hollister Hall

  Tuesday, April 28, 7-8pm in B14 Hollister Hall

Project4: Final project out next week

- Demos: May 12 and 13
- ***Will NOT be able to use slip days***

# Announcements

Next three weeks

- Week 12 (Apr 21): Lab4 due in-class, Proj3 due Fri, HW2 due Sat
- Week 13 (Apr 28): Proj4 release, Prelim2
- Week 14 (May 5): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

- Week 15 (May 12): Proj4 due Wed, May 13th

# Cache Coherency and Synchronization Problem

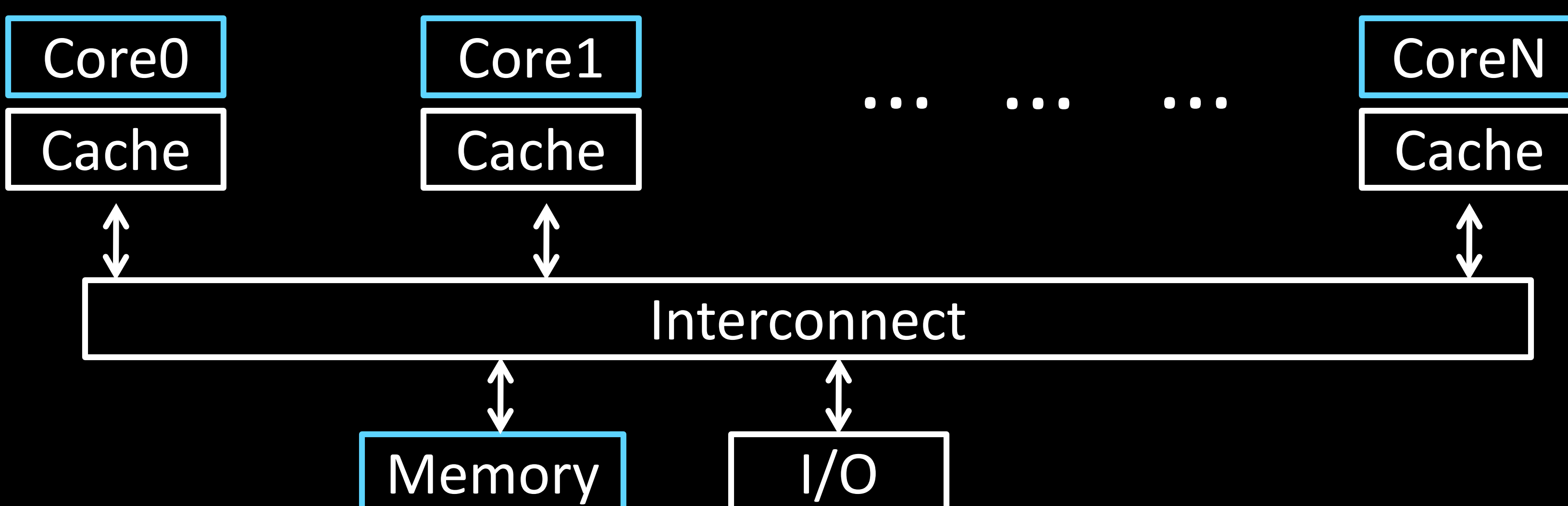
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1
}
```

x should be greater than 1 after both threads loop at least once!



# Cache Coherency and Synchronization Problem

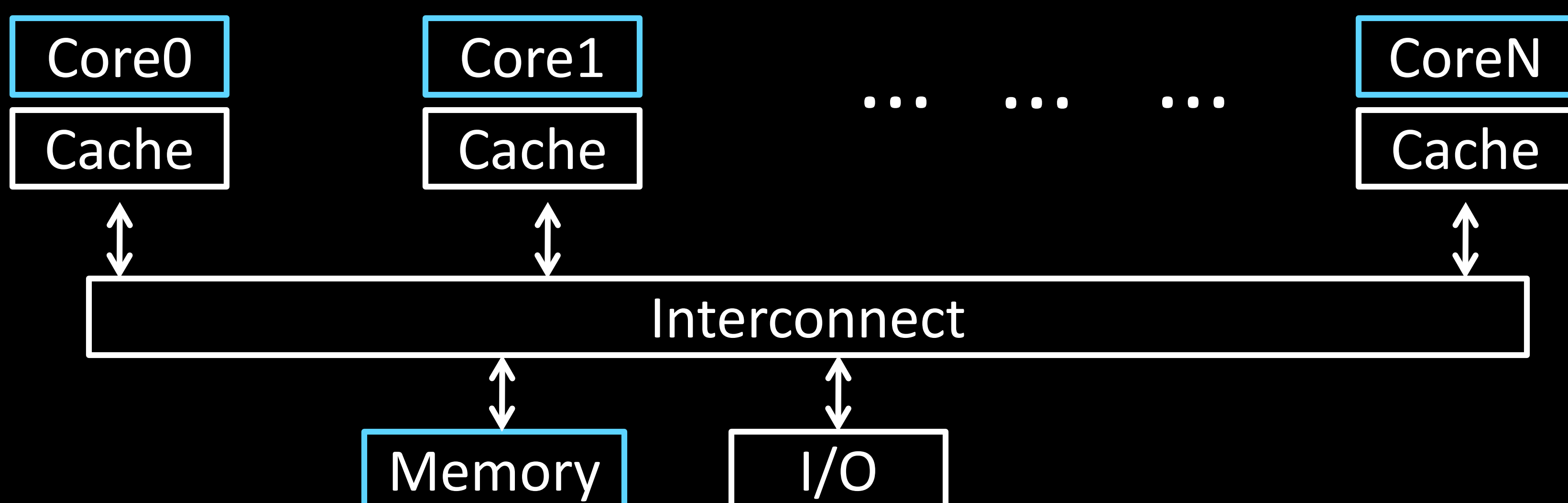
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
$t0=0    LW $t0, addr(x)
$t0=1    ADDIU $t0, $t0, 1
  x=1    SW $t0, addr(x)
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
$t0=0    LW $t0, addr(x)
$t0=1    ADDIU $t0, $t0, 1
  x=1    SW $t0, addr(x)
}
```

x should be greater than 1 after both threads loop at least once!



# 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

# 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
- Language level synchronization

# 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

# HW support for critical sections

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?

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)
}
```

# 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.

i.e. use this value 0 in $t0 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

i++

↓

```
LW $t0, 0($s0)
ADDIU $t0, $t0, 1
SW $t0, 0($s0)
```

→

atomic(i++)

↓

```
try: LL $t0, 0($s0)
     ADDIU $t0, $t0, 1
     SC $t0, 0($s0)
     BEQZ $t0, try
```

# 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;     LL    Atomic
     *m = 1;       SC
     return old;
}
```

# 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) {
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; // 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) {
 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; // m=0 means lock is free; otherwise, if m=1, then lock locked

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

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

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

```
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 | | | | | | | 1 |
| 1 | try: LI $t0, 1 | try: LI $t0, 1 | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |

# Now we can write parallel and correct programs

Thread A

```
for(int i = 0, i < 5; i++) {

    mutex_lock(m);
        x = x + 1;
    mutex_unlock(m);

}
```

Thread B

```
for(int j = 0; j < 5; j++) {

    mutex_lock(m);
        x = x + 1;
    mutex_unlock(m);

}
```

# 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

# Summary

Need parallel abstractions, especially for multicore

Writing correct programs is hard

- Need to prevent data races

Need critical sections to prevent data races

- Mutex, mutual exclusion, implements critical section
- Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as **LL** and **SC** (load linked and store conditional) instructions to efficiently implement locks

# Next Goal

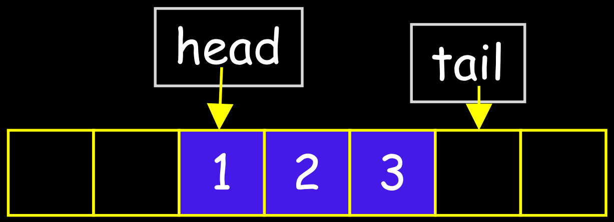
How do we use synchronization primitives to build concurrency-safe data structure?

# Attempt#1: Producer/Consumer

Access to shared data must be synchronized

- goal: enforce datastructure invariants

```
// invariant:
// data is in A[h … t-1]
char A[100];
int h = 0, t = 0;
```



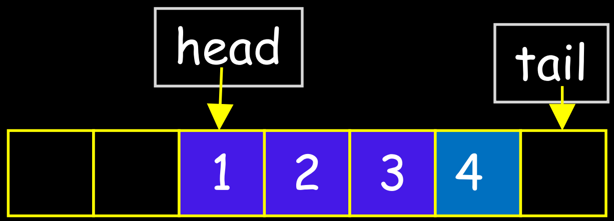

```
// producer: add to list tail
void put(char c) {
  A[t] = c;
  t = (t+1)%n;
}
```

# Attempt#1: Producer/Consumer

Access to shared data must be synchronized

- goal: enforce datastructure invariants

```
// invariant:
// data is in A[h … 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;
}
```

```
// consumer: take from list head
char get() {
  while (h == t) { };
  char c = A[h];
  h = (h+1)%n;
  return c;
}
```

# 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
void put(char c) {
  pthread_mutex_lock(m);
  A[t] = c;
  t = (t+1)%n;
  pthread_mutex_unlock(m);
}
```

```
// consumer: take from list head
char get() {
  pthread_mutex_lock(m);
  while(h == t) {}
  char c = A[h];
  h = (h+1)%n;
  pthread_mutex_unlock(m);
  return c;
}
```

# 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);
```

Circular Wait

- 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!

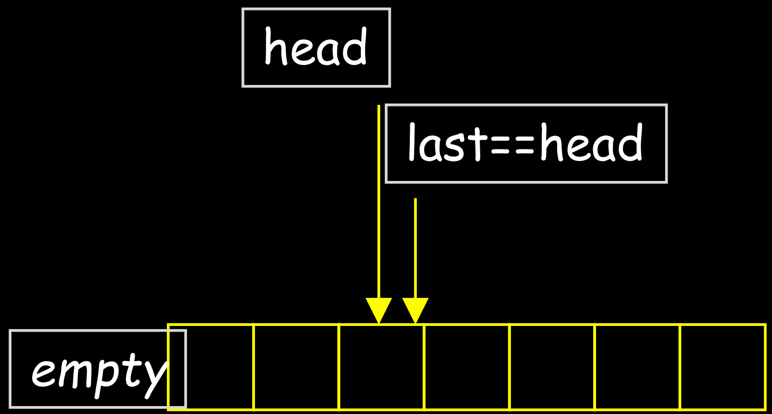
# Attempt#3: Beyond mutexes

Writers must check for full buffer

& Readers must check if for empty buffer

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

```
char get() {
  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;
}
```

# 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, ...