

Synchronization 2

CS 3410, Spring 2014

Computer Science

Cornell University

See P&H Chapter: 2.11, 6.5

Administrivia

Next 3 weeks

- Week 12 (this week): Proj3 due ~~Fri~~Sun
 - Note Lab 4 is now IN CLASS
 - Prelim 2 review Sunday and Monday
- Week 13 (Apr 29): Proj4 release, ~~Lab4 due Tue~~, Prelim2
- Week 14 (May 6): Proj3 tournament Mon, Proj4 design doc due

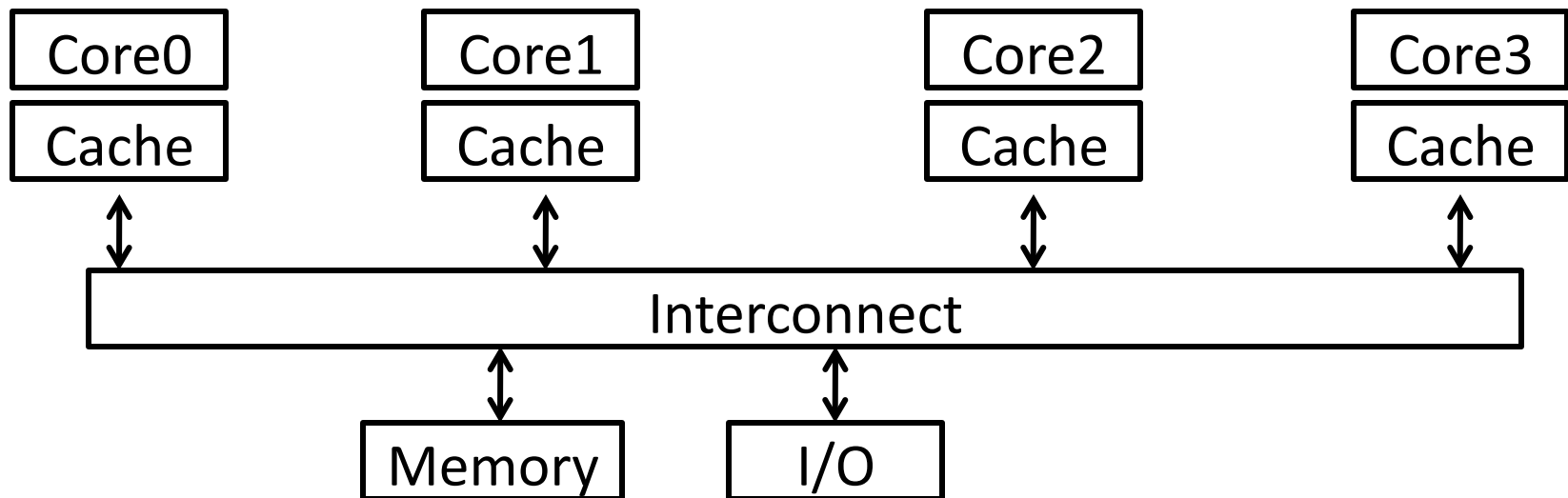
Final Project for class

- Week 15 (May 13): Proj4 due Wed
- Remember: No slip days for PA4

Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume uniform memory access (UMA) (ignore NUMA)



Cache Coherency Problem

Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {  
    A1) LW $t0, addr(x)  
    A2) ADDIU $t0, $t0, 1  
    A3) SW $t0, addr(x)  
}
```

Thread B (on Core1)

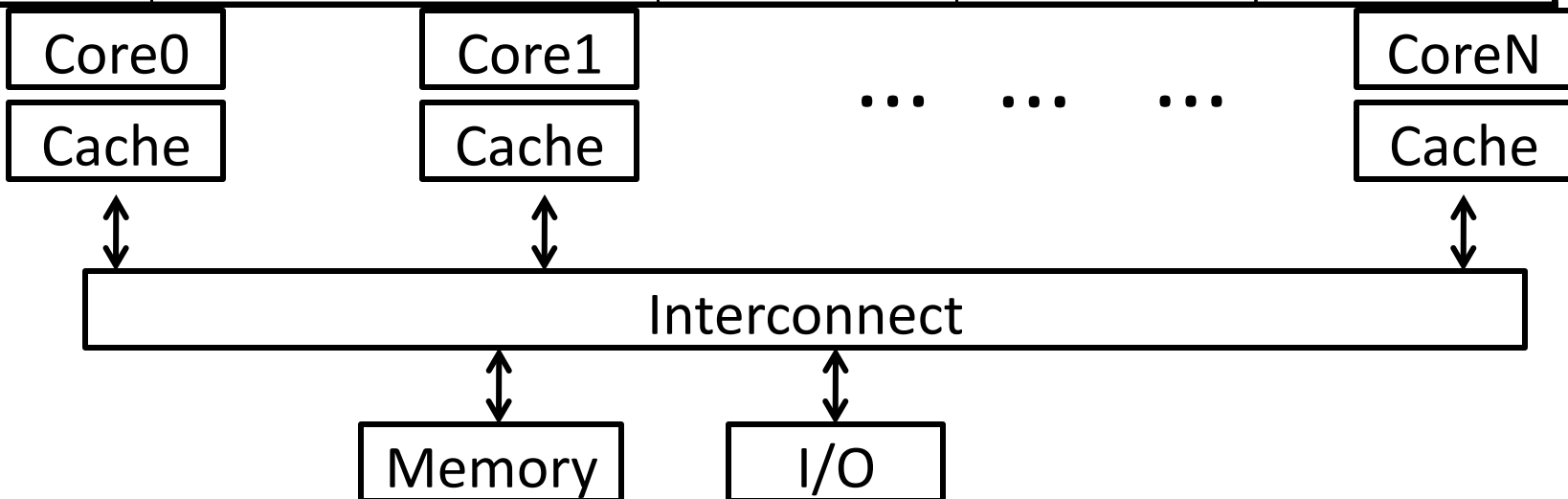
```
for(int j = 0; j < 5; j++) {  
    B1) LW $t0, addr(x)  
    B2) ADDIU $t0, $t0, 1  
    B3) SW $t0, addr(x)  
}
```

Cache Coherence Problem

Suppose two CPU cores share a physical address space

- Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1



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
 - Aka write serialization

Coherence Defined

Formal: For concurrent processes P_1 and P_2

- P writes X before P reads X (with no intervening writes)
⇒ read returns written value
 - (preserve program order)
- P_1 writes X before P_2 reads X
⇒ read returns written value
 - (coherent memory view, can't read old value forever)
- 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
 - Aka write serialization
 - (else X can see P_2 's write before P_1 and Y can see the opposite; their final understanding of state is wrong)

Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence and support shared memory

- Migration of data to local caches
 - Reduces bandwidth for shared memory (performance)
- Replication of read-shared data
 - Reduces contention for access (performance)

Snooping protocols

- Each cache monitors bus reads/writes (correctness)

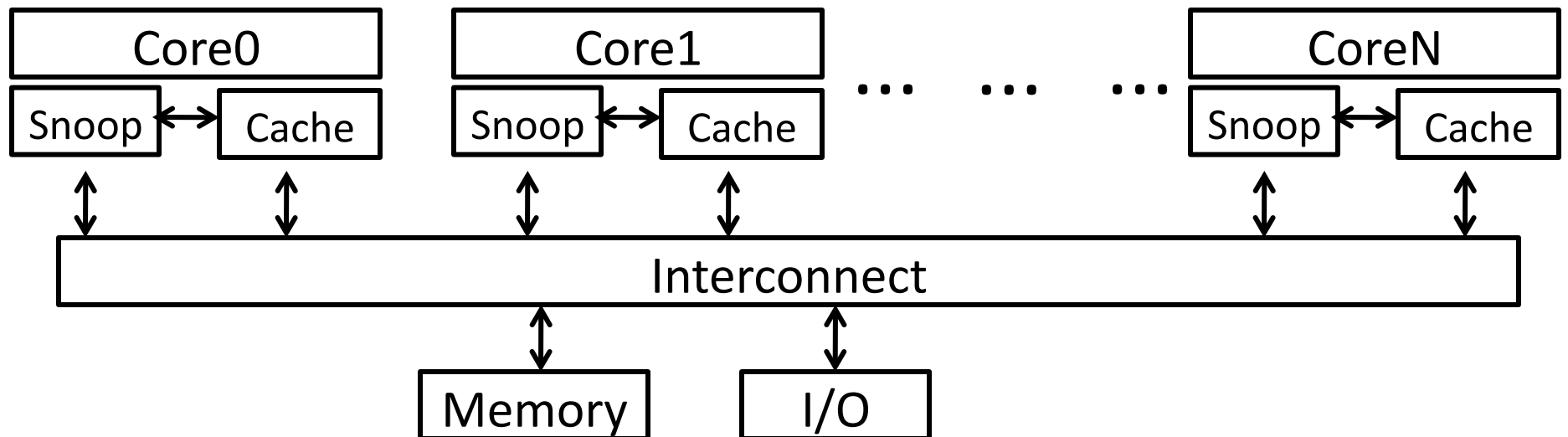
Snooping

Snooping for Hardware Cache Coherence

- All caches monitor bus and all other caches

Write invalidate protocol




- Bus read: respond if you have dirty data
- Bus write: update/invalidate your copy of data



Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written





- Broadcasts an invalidate message on the bus
- Subsequent read is another cache miss
 - Owning cache supplies updated value

Time Step	CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
0					0
1	CPU A reads X	Cache miss for X	0		0
2	CPU B reads X	Cache miss for X	0	0	0
3	CPU A writes 1 to X	Invalidate for X	1		0
4	CPU B read X	Cache miss for X			

Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read is another cache miss
 - Owning cache supplies updated value

Time Step	CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
0					0
1	CPU A reads X	Cache miss for X	0		0
2	CPU B reads X	Cache miss for X	0	0	0
3	CPU A writes 1 to X	Invalidate for X	1		0
4	CPU B read X	Cache miss for X			

Writing

Write-back policies for bandwidth

Write-invalidate coherence policy

- First invalidate all other copies of data
- Then write it in cache line
- Anybody else can read it

Works with one writer, multiple readers

In reality: many coherence protocols

- Snooping doesn't scale
- MOESI, MOSI, ... (mod, own, exclusive, share, inv)
- Directory-based protocols
 - Caches and memory record sharing status of blocks in a directory

Summary of cache coherence

Cache coherence requires that reads return most recently written value

Cache coherence is hard

Snooping protocols are one approach

Cache coherence protocols alone are not enough

Need more for consistency

Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
 - HW support for synchronization
 - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes

Programming with Threads

Need it to exploit multiple processing units

...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

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

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

Cache coherency isn't sufficient...

Need explicit synchronization to make sense of concurrency!

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

Example: Multi-Threaded Program

Apache web server

```
void main() {  
    setup();  
    while (c = accept_connection()) {  
  
        req = read_request(c);  
        hits[req]++;  
        send_response(c, req);  
  
    }  
    cleanup();  
}
```

Example: web server

Each client request handled by a separate thread
(in parallel)

- Some shared state: hit counter, ...

```
Thread 52  
read hits  
addi  
write hits
```

(look familiar?)

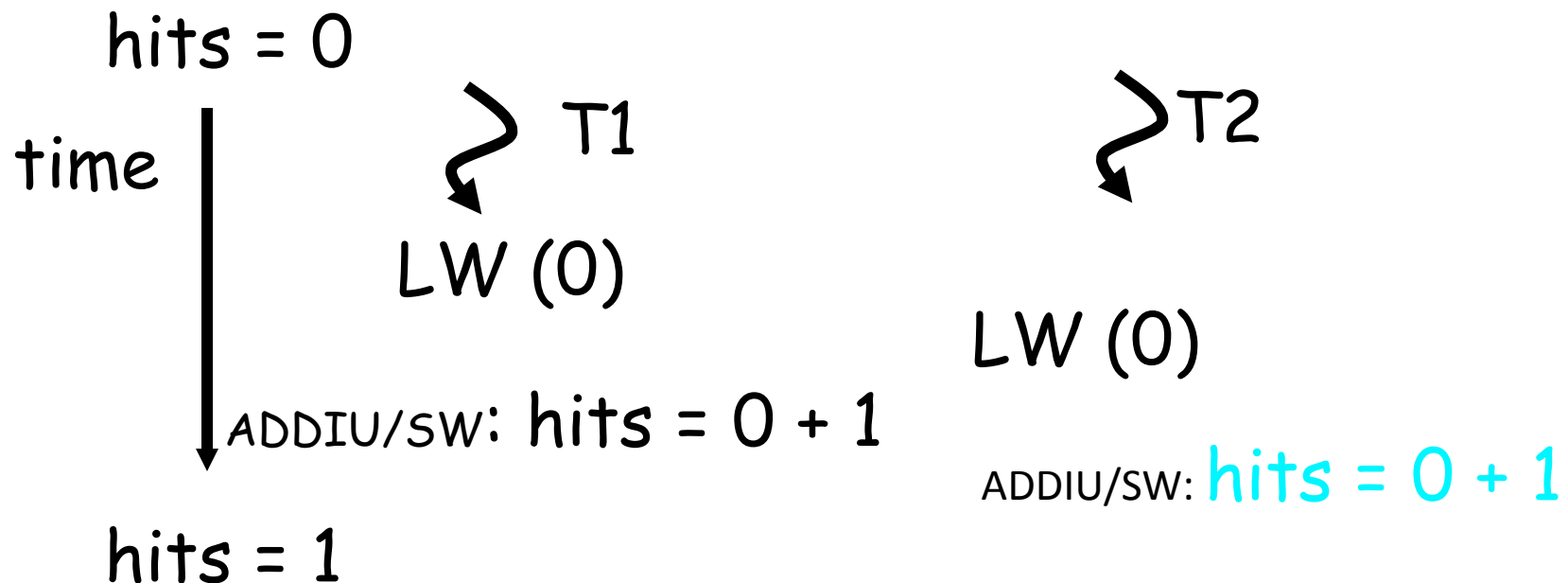
```
Thread 205  
read hits  
addi  
write hits
```

Timing-dependent failure \Rightarrow race condition

- hard to reproduce \Rightarrow hard to debug

Two threads, one counter

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

Critical sections

What if we can designate parts of the execution as critical sections

- Rule: only one thread can be “inside” a critical section

Thread 52

**read hits
addi
write hits**

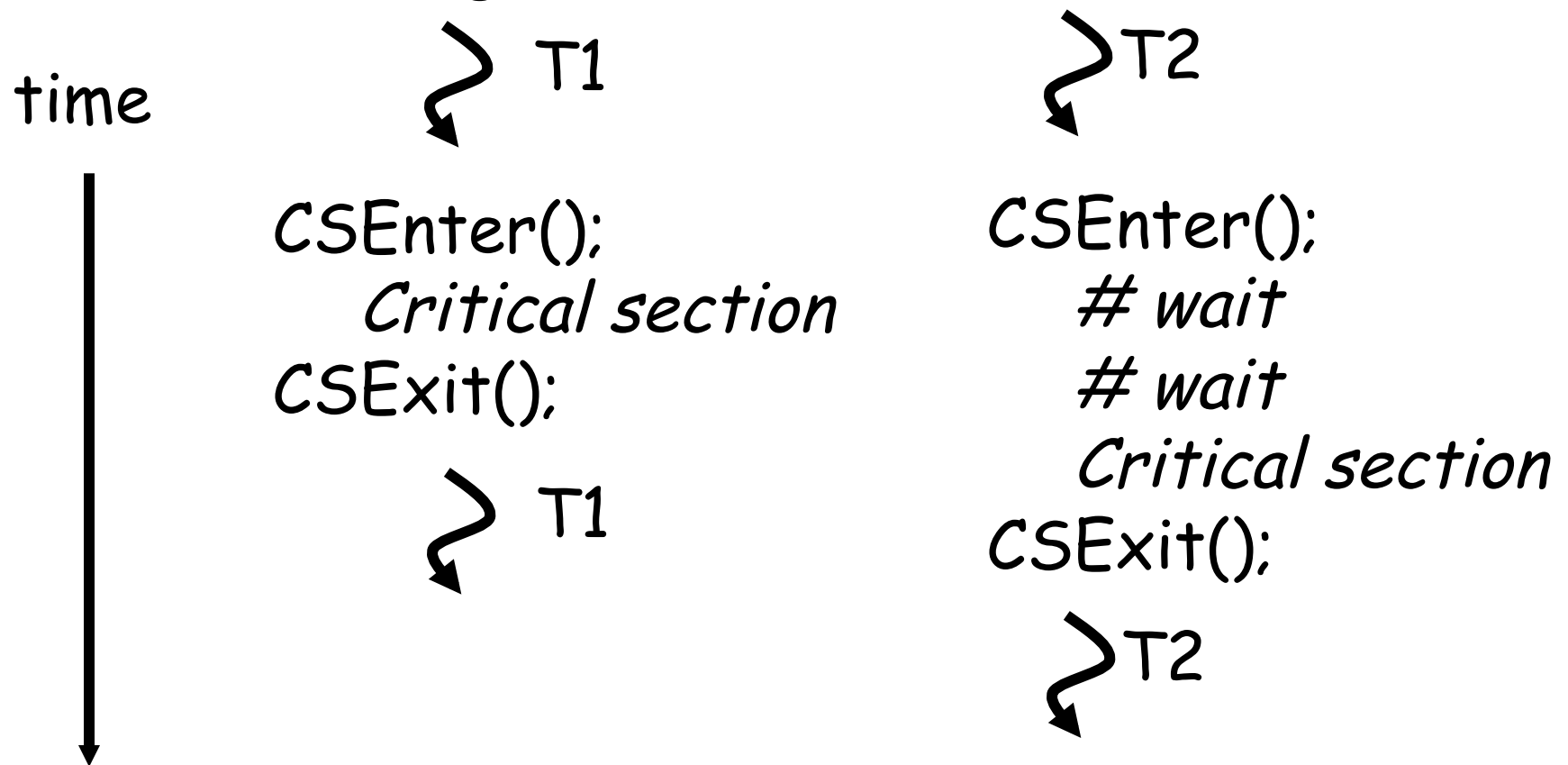
Thread 205

**read hits
addi
write hits**

Critical Sections

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

- Contending threads must wait to enter



Mutexes

Q: How to implement critical sections 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);  
}
```

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

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

Synchronization

Synchronization requires hardware support

- 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
ATS, 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

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in \$t0

Use this value 0 to try again

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // 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  
}
```

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			

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)	0	0	0
2	ADDIU \$t0, \$t0, 1	ADDIU \$t0, \$t0, 1	1	1	0
3	SC \$t0, 0(\$s0)	SC \$t0, 0 (\$s0)	0	1	1
4	BEQZ \$t0, try	BEQZ \$t0, try	0	1	1


Mutex from LL and SC

```
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

```
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

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

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	1		1		0
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	0	1	0	0
3	BNEZ \$t1, try	BNEZ \$t1, try	1	0	1	0	0
4	SC \$t0, 0(\$a0)	SC \$t0, 0 (\$a0)	0	0	1	0	1
5	BEQZ \$t0, try	BEQZ \$t0, try	0	0	1	0	1
6							

Mutex from LL and SC

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	1		1		0
2	LL \$t1, 0(\$a0)	LL \$t1, 0(\$a0)	1	0	1	0	0
3	BNEZ \$t1, try	BNEZ \$t1, try	1	0	1	0	0
4	SC \$t0, 0(\$a0)	SC \$t0, 0 (\$a0)	0	0	1	0	1
5	BEQZ \$t0, try	BEQZ \$t0, try	0	0	1	0	1
6	try: LI \$t0, 1	Critical section					

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, ...)

Summary

Need parallel abstraction like 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

Topics

Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
 - HW support for synchronization
 - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes

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

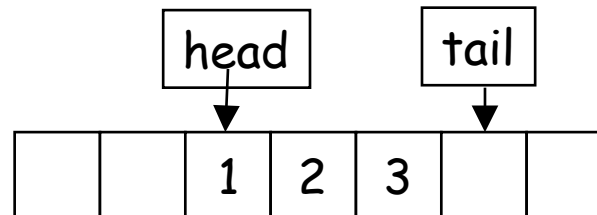
- goal: enforce data structure invariants

```
// 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

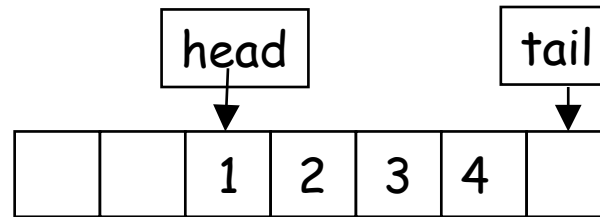
- goal: enforce datastructure invariants

```
// 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) {
```

```
    // Need: check if list full
```

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

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

```
}
```

Attempt#1: Producer/Consumer

Access to shared data must be synchronized

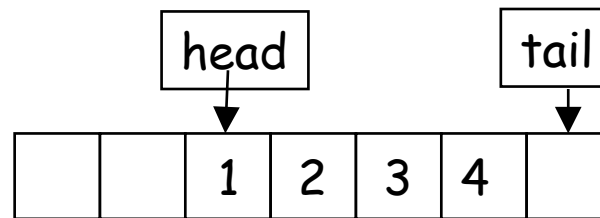
- goal: enforce datastructure invariants

```
// 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) {
```

```
    // Need: check if list full
```

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

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

```
}
```

```
// consumer: take from list head
```

```
char get() {
```

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

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

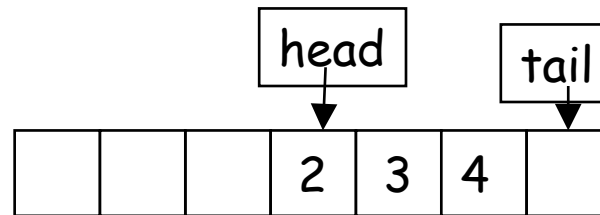
```
    h = (h+1)%n;
```

```
    return c;
```

```
}
```

Attempt#1: Producer/Consumer

```
// 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 = (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;

// 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;
}
```

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

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;
```

BUG: Can't wait while holding lock



```
// 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;  
}
```

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!

But 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

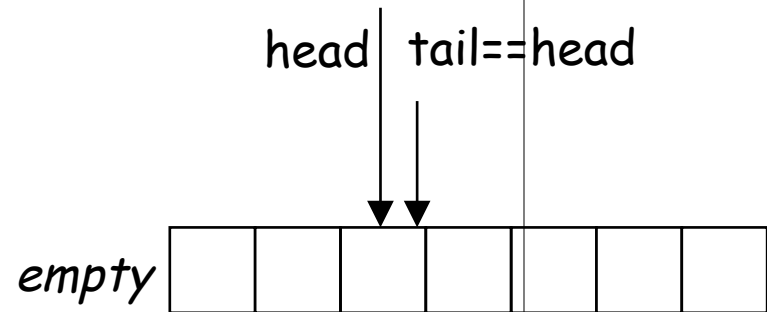
Writers must check for full buffer

& Readers must check if for empty buffer

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

```
char get() {  
    while (h == t) { };  
    lock (L);  
    char c = A[h];  
    h = (h+1)%n;  
    unlock (L);  
    return c;  
}
```

Cannot check condition while
Holding the lock,
BUT, empty condition may no
longer hold in critical section



Dilemma: Have to check while holding lock

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() {  
    lock (L);  
    while (h == t) { };  
    char c = A[h];  
    h = (h+1)%n;  
    unlock (L);  
    return c;  
}
```

Dilemma: Have to check while holding lock,
but cannot wait while holding 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 {  
        lock (L);  
        empty = (h == t);  
        if (!empty) {  
            c = A[h];  
            h = (h+1)%n;  
        }  
        unlock (L);  
    } while (empty);  
    return c;  
}
```

Language-Level Synchronization

Condition variables

- Wait for condition to be true

- Thread sleeps while waiting

- Can wake up one thread or all threads

Monitors

...

Summary

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

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

Language Constructs: monitors, signals, ...