Multicore, Parallelism, and Synchronization

Hakim Weatherspoon

CS 3410, Spring 2015

Computer Science

Cornell University

P&H Chapter 2.11, 5.10, and 6.5

Announcements

- HW2 Review Sessions!
  - TODAY, Tue, April 21st, Hollister B14@7pm
- HW2-P5 (Pre-Lab4) was due yesterday!

- PA3 due Friday, April 24th
  - The Lord of the Cache!
  - **Tournament, Monday, May 4, 5-7pm**

Announcements

- Prelim 2 is ***next week*** on April 30th at 7 PM at Statler Hall!

- If you have a conflict e-mail me:

deniz@cs.cornell.edu

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

Big Picture: Parallelism and Synchronization

How do I take advantage of *parallelism*?

How do I write (**correct**) parallel programs?

What primitives do I need to implement correct parallel programs?

Topics: Goals for Today

Understand Cache Coherency

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

Parallelism and Synchronization

Cache Coherency Problem: What happens when to two or more processors cache ***shared*** data?

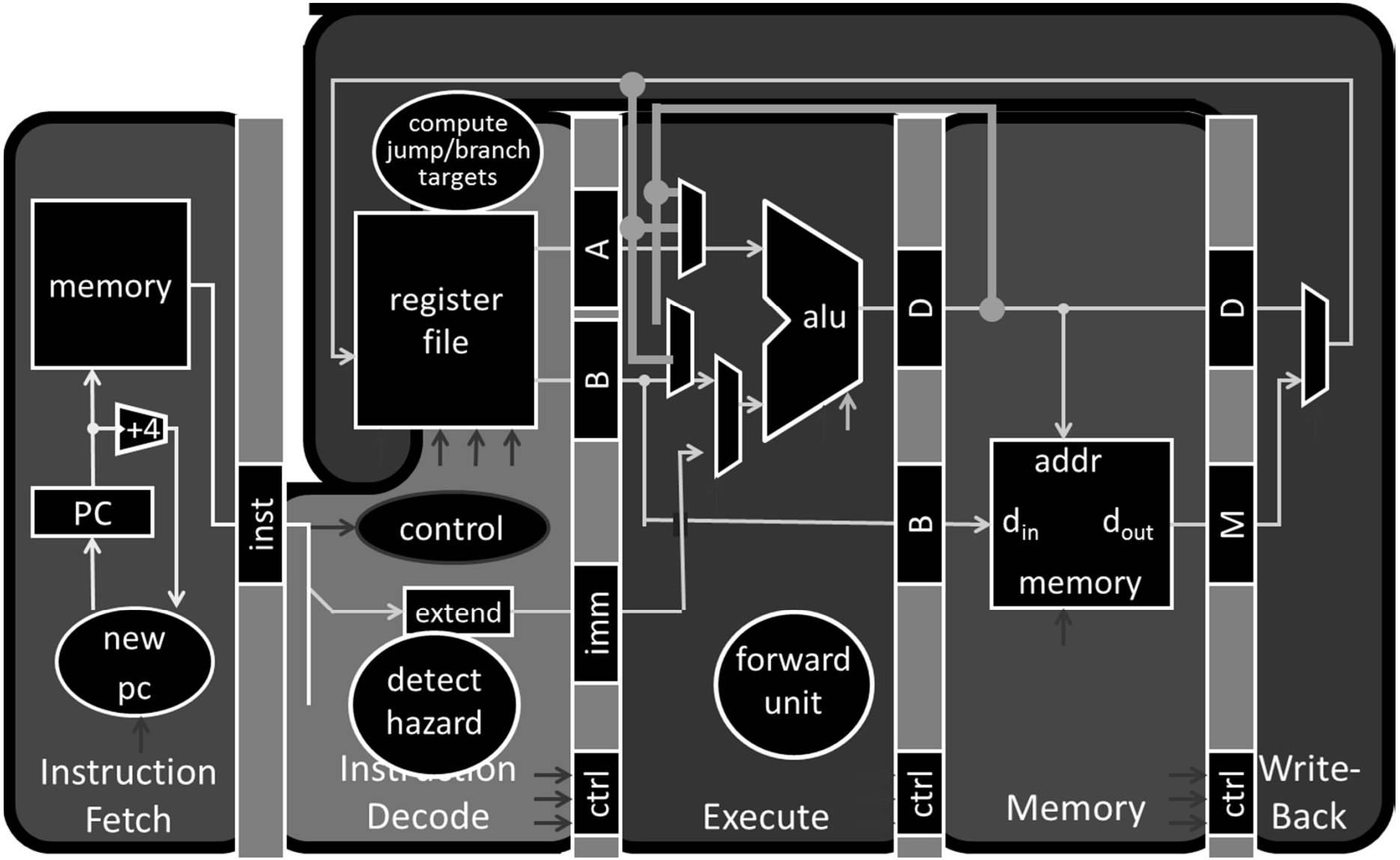
Parallelism and Synchronization

Cache Coherency Problem: What happens when to two or more processors cache *shared* data?

i.e. the view of memory held by two different processors is through their individual caches.

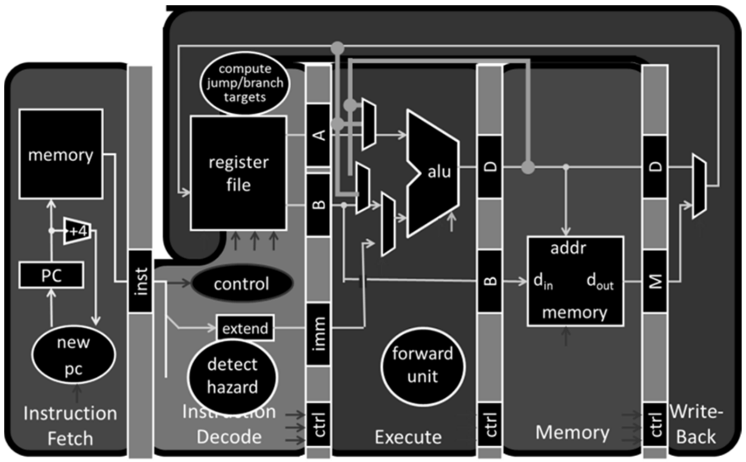
As a result, processors can see different (incoherent) values to the *same* memory location.

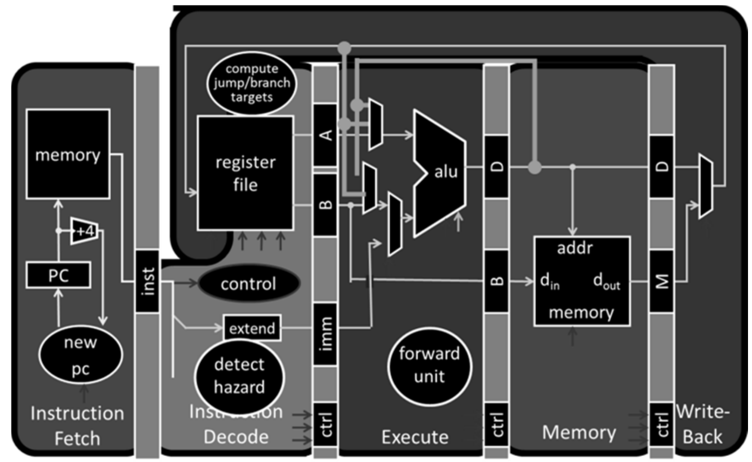
Parallelism and Synchronization

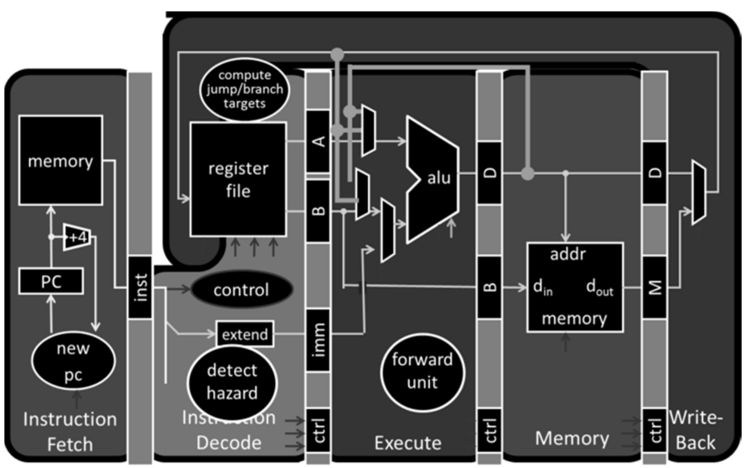


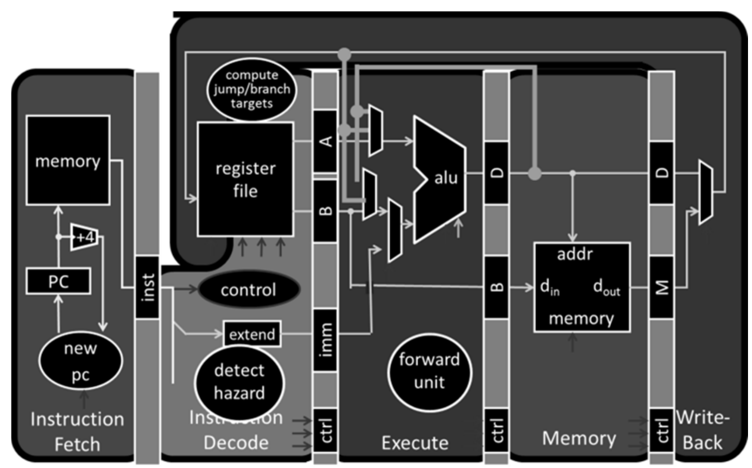
Parallelism and Synchronization

Each processor core has its own L1 cache



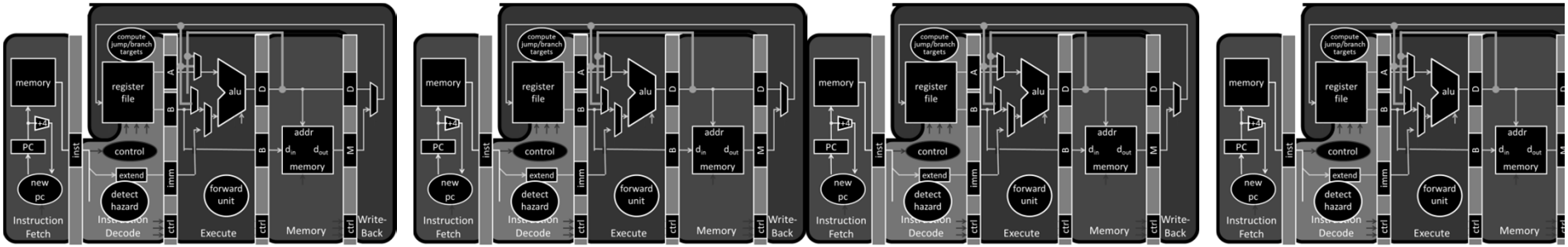






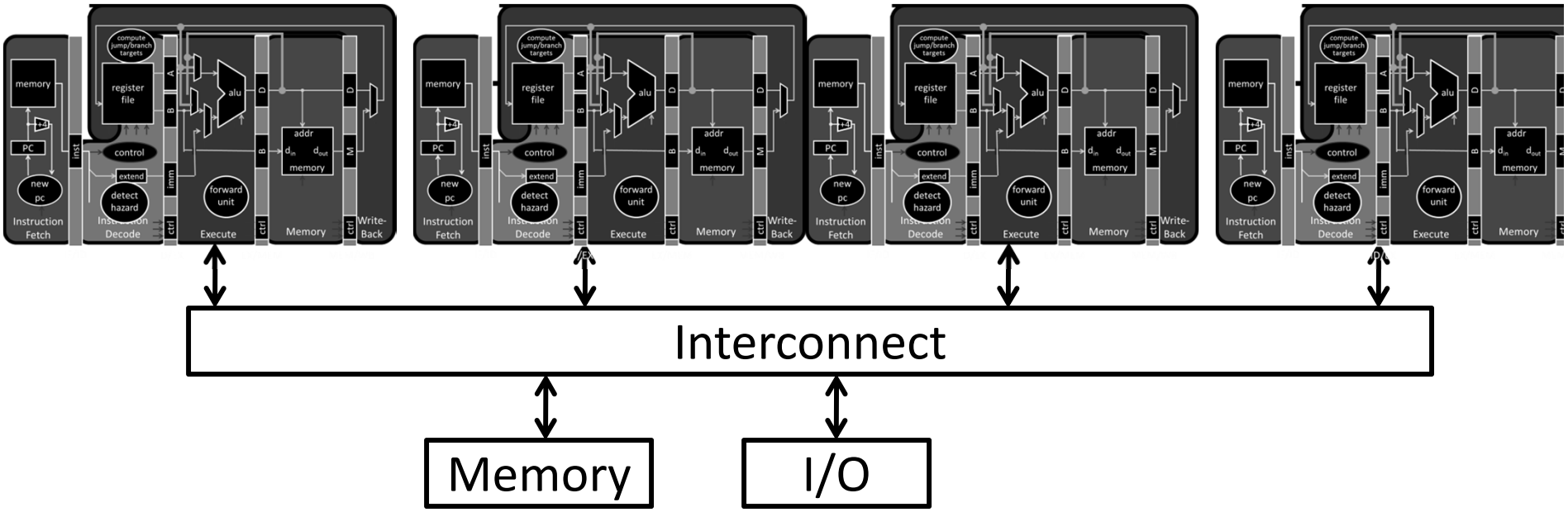
Parallelism and Synchronization

Each processor core has its own L1 cache



Parallelism and Synchronization

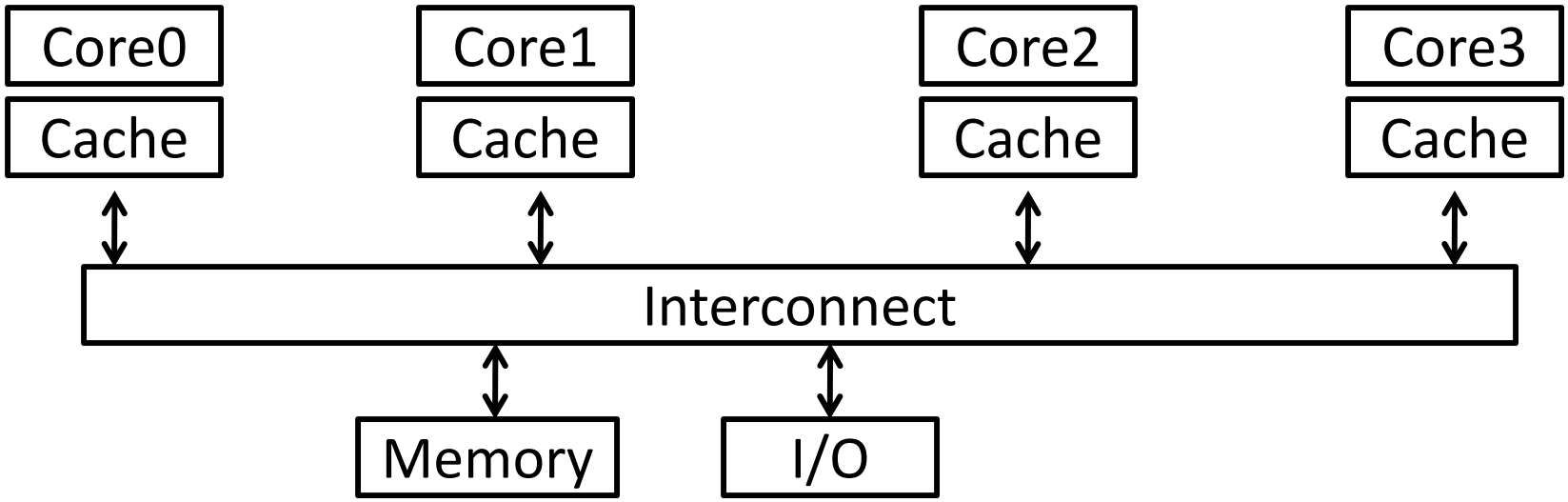
Each processor core has its own L1 cache



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

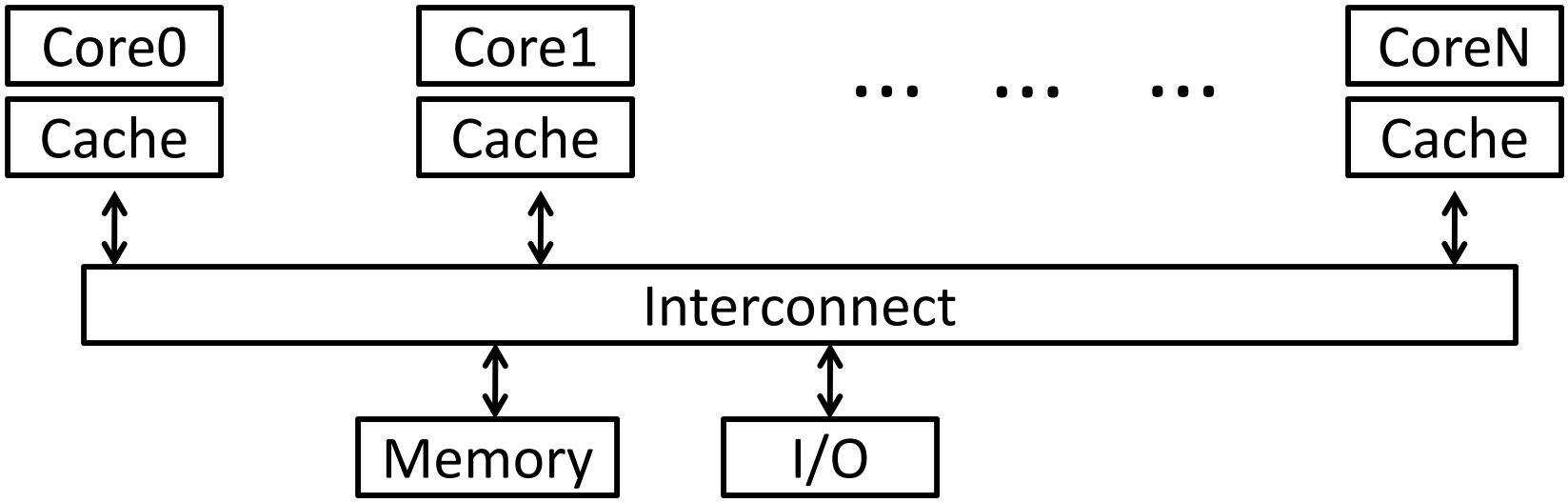
- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)



Cache Coherency Problem

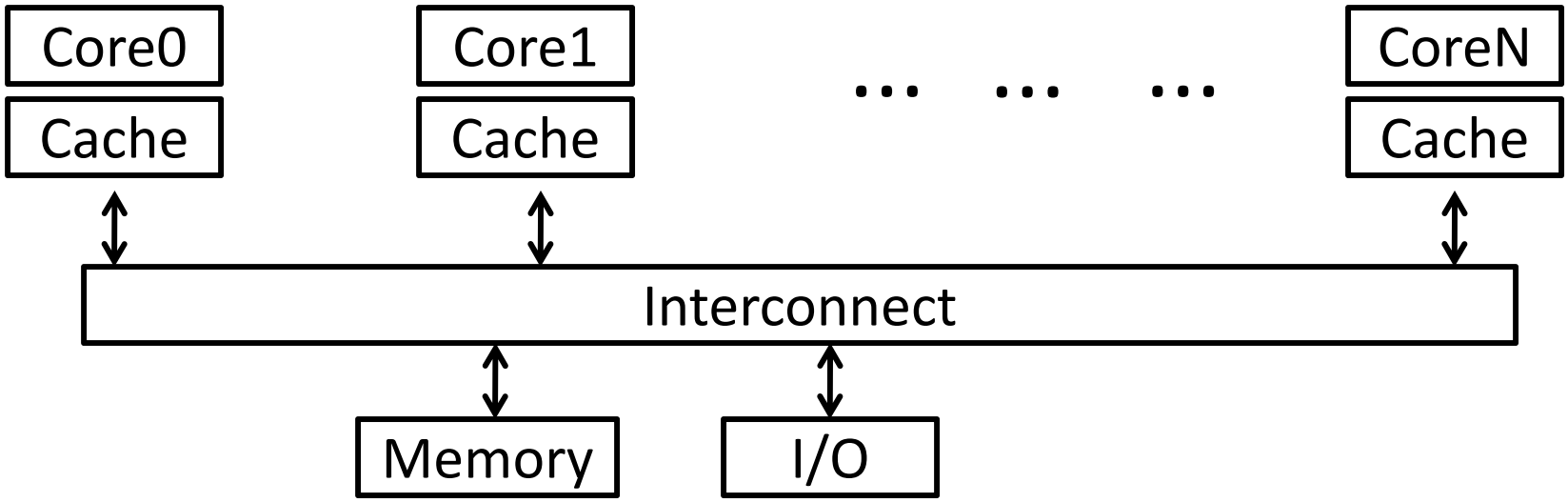
Thread A (on Core0)

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

Thread B (on Core1)

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

What will the value of x be after both loops finish?

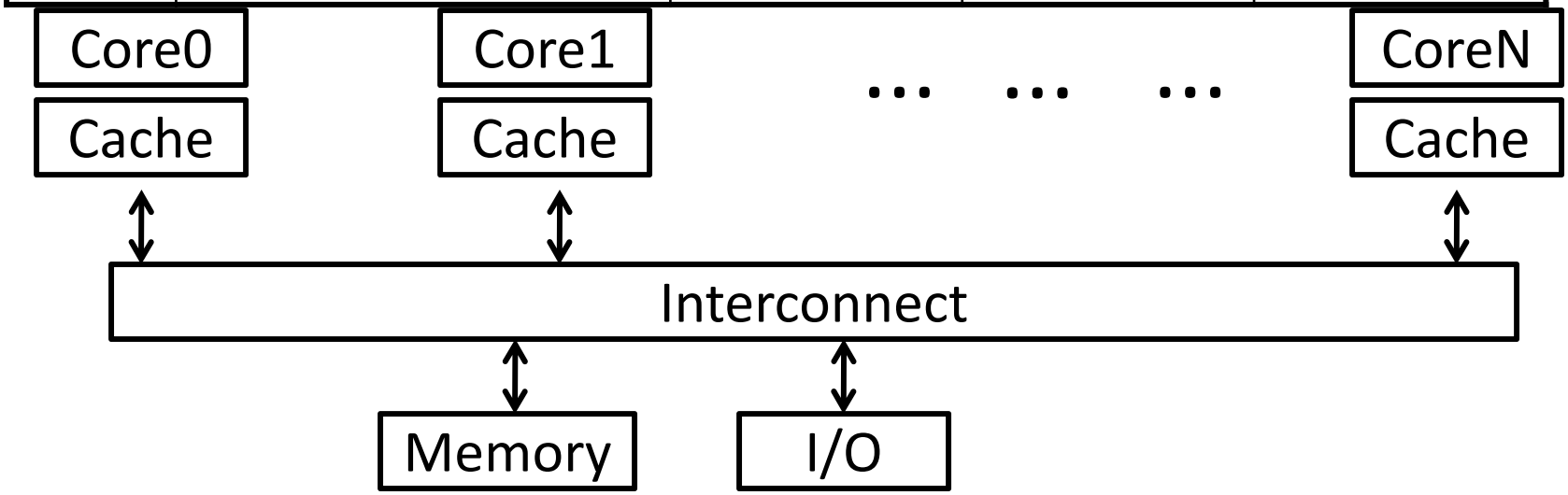


Cache Coherence Problem

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



Two issues

Coherence

What values can be returned by a read

Consistency

When a written value will be returned by a read

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)
  $\Rightarrow$ read returns written value
  - (preserve program order)
- P_1 writes X before P_2 reads X
  $\Rightarrow$ read returns written value
  - (coherent memory view, can't read old value forever)
- P_1 writes X and P_2 writes X
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for X
  - Aka write serialization
  - (else P_A can see P_2's write before P_1's and P_B can see the opposite; their final understanding of state is wrong)

Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence

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

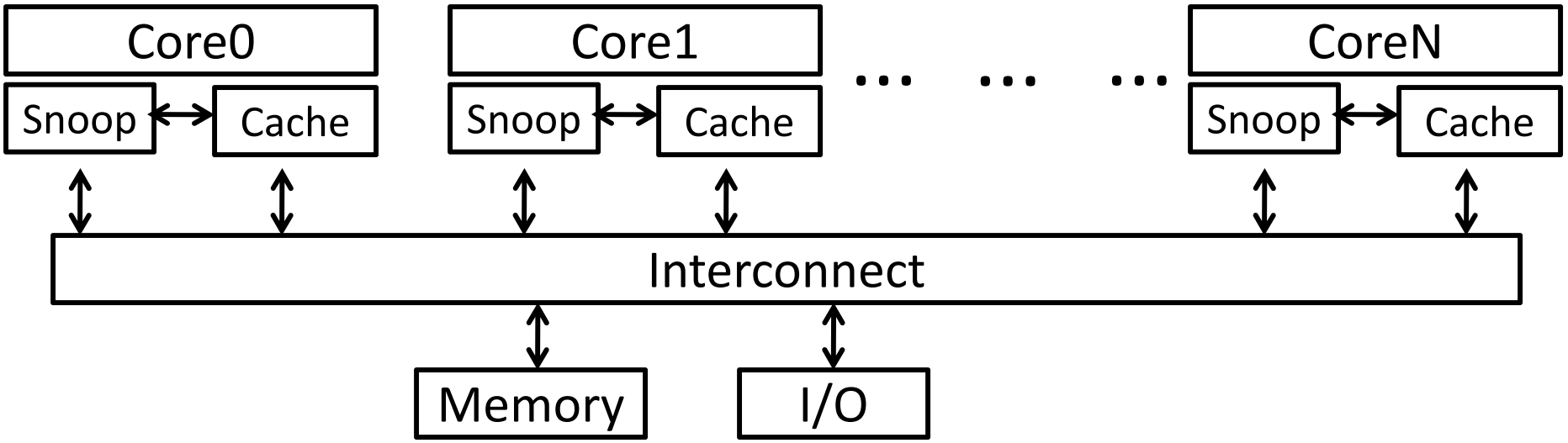
Snooping protocols

- Each cache monitors bus reads/writes

Snooping

Snooping for Hardware Cache Coherence

- All caches monitor bus and all other caches
- 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 in another cache misses
  - 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				
2	CPU B reads X				
3	CPU A writes 1 to X				
4	CPU B read 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

Permits one writer, multiple readers

In reality: many coherence protocols

- Snooping doesn't scale
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Takeaway: Summary of cache coherence

Informally, Cache Coherency requires that reads return most recently written value

Cache coherence hard problem

Snooping protocols are one approach

Next Goal: Synchronization

Is cache coherency sufficient?

i.e. Is cache coherency (***what*** values are read) sufficient to maintain consistency (***when*** a written value will be returned to a read). Both coherency and consistency are required to maintain consistency in shared memory programs.

Is Cache Coherency Sufficient?

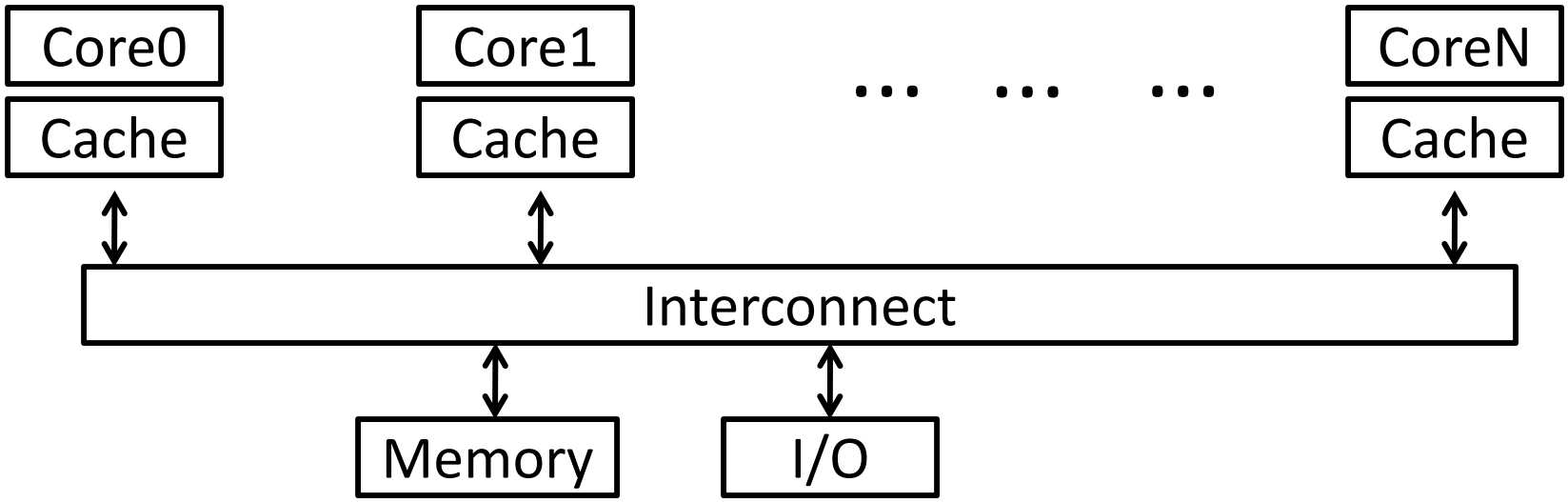
Thread A (on Core0)

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

Thread B (on Core1)

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

Very expensive and difficult to maintain 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 is **not** 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
addiu
write hits
```

```
Thread 205
read hits
addiu
write hits
```

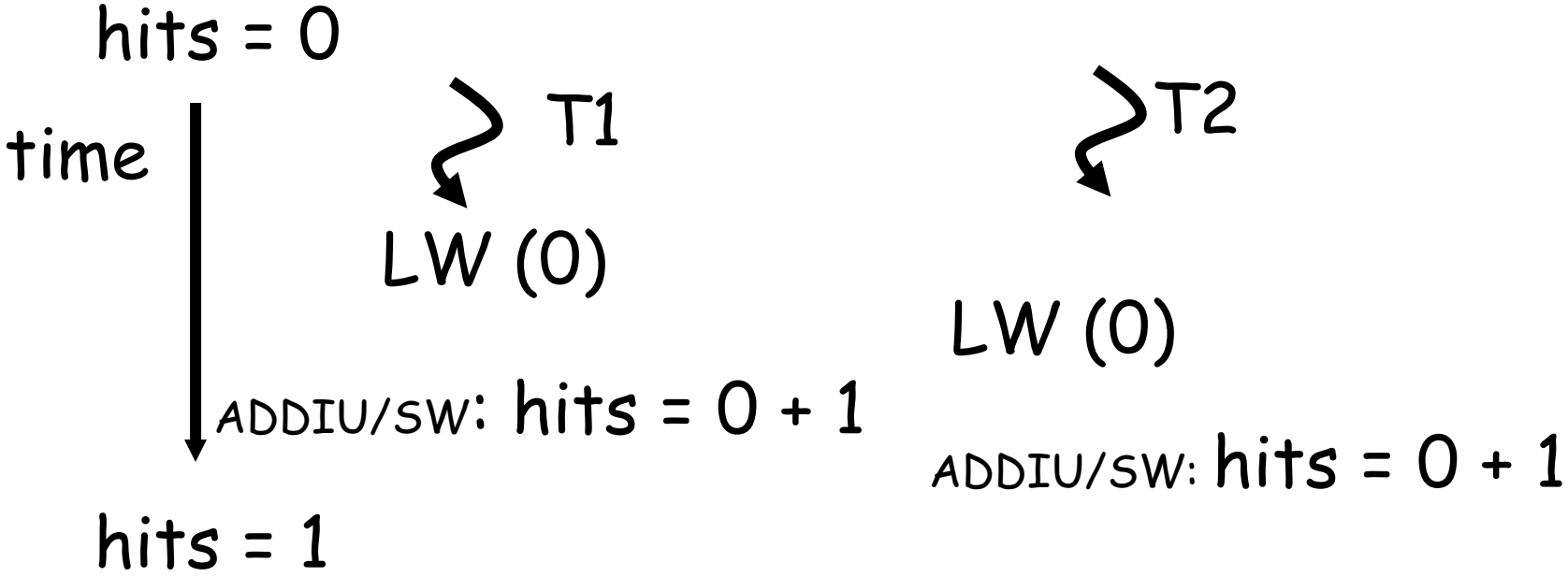
(look familiar?)

Timing-dependent failure $\Rightarrow$ race condition

- hard to reproduce $\Rightarrow$ hard to debug

Two threads, one counter

Possible result: lost update!



Timing-dependent failure ⇒ race condition

- **Very hard to reproduce ⇒ 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

```
CSEnter()
read hits
addi
write hits
CSExit()
```

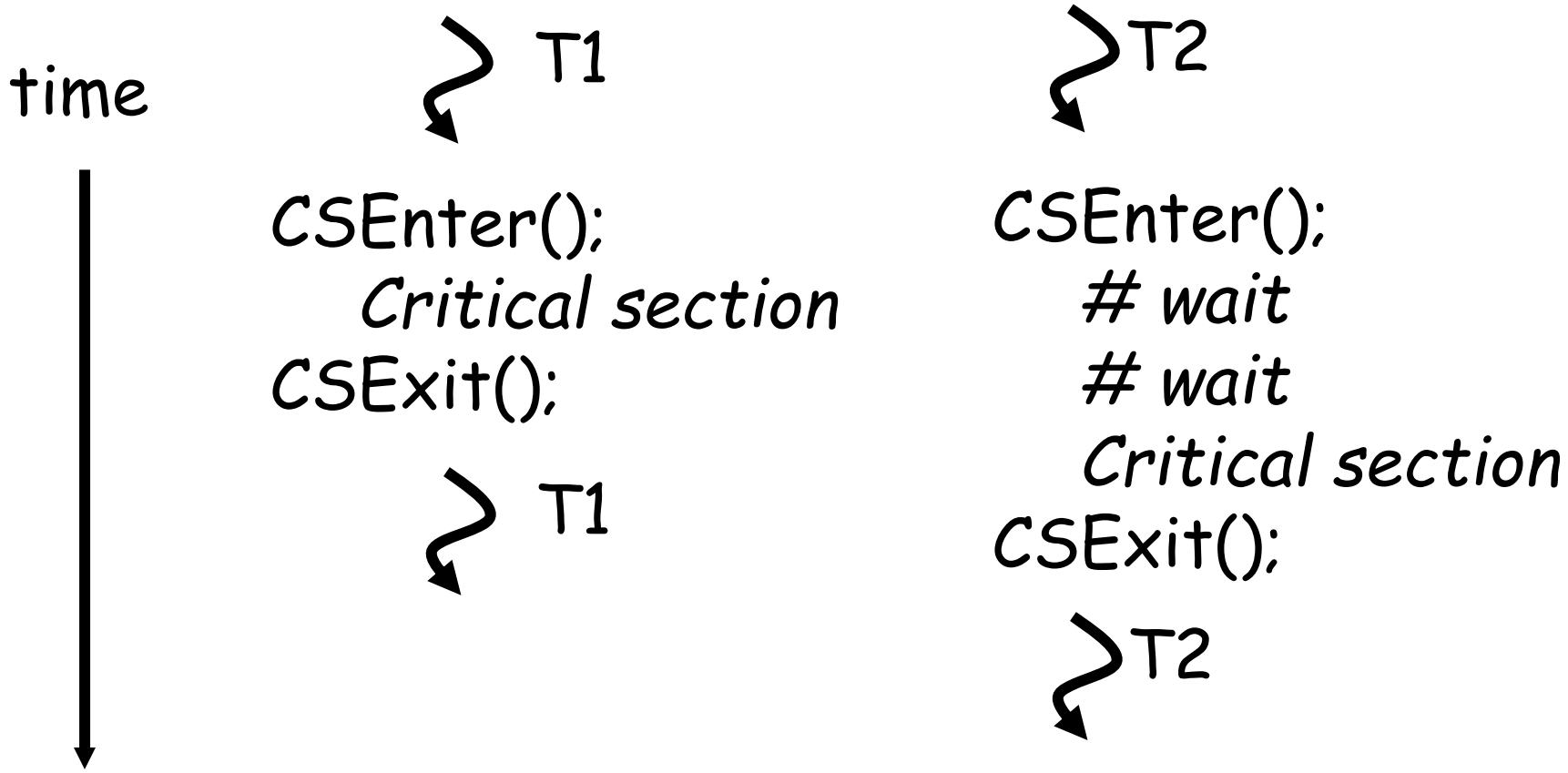
Thread 205

```
CSEnter()
read hits
addi
write hits
CSExit()
```

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

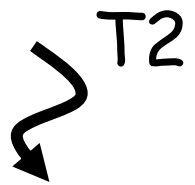
Acquire (lock) mutex on entry to critical section

- 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);
```



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 ↔ memory (e.g. 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, 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

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)
      MOVE $v0, $t1
}
```

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) {
 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) {
    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