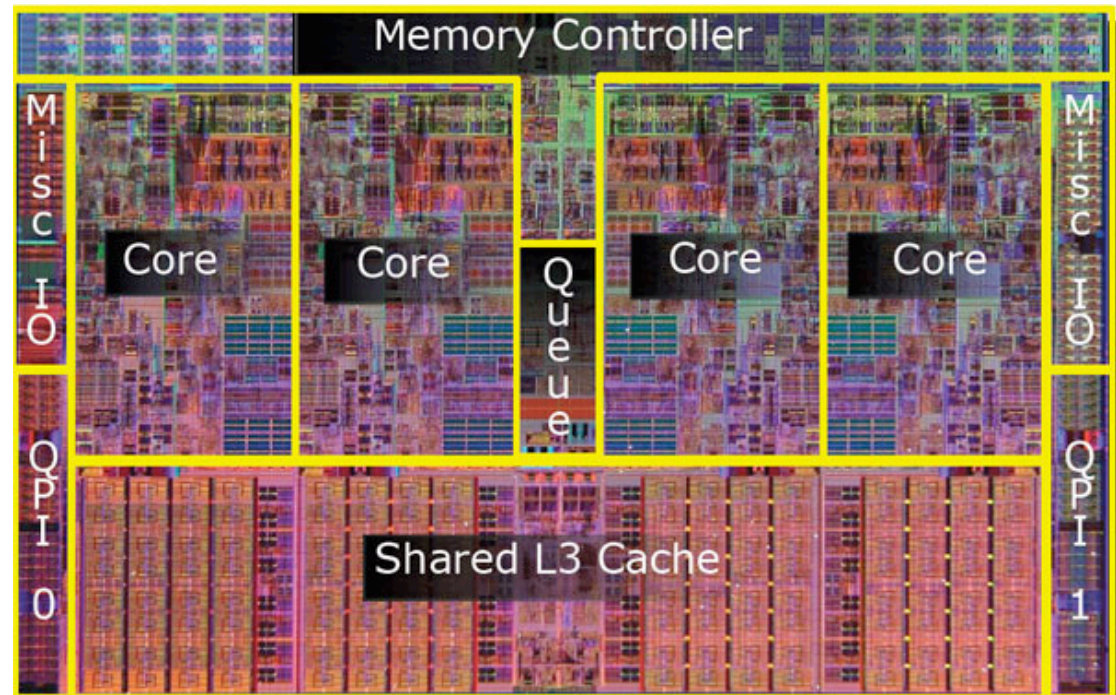


# Cache Coherence and Atomic Operations in Hardware

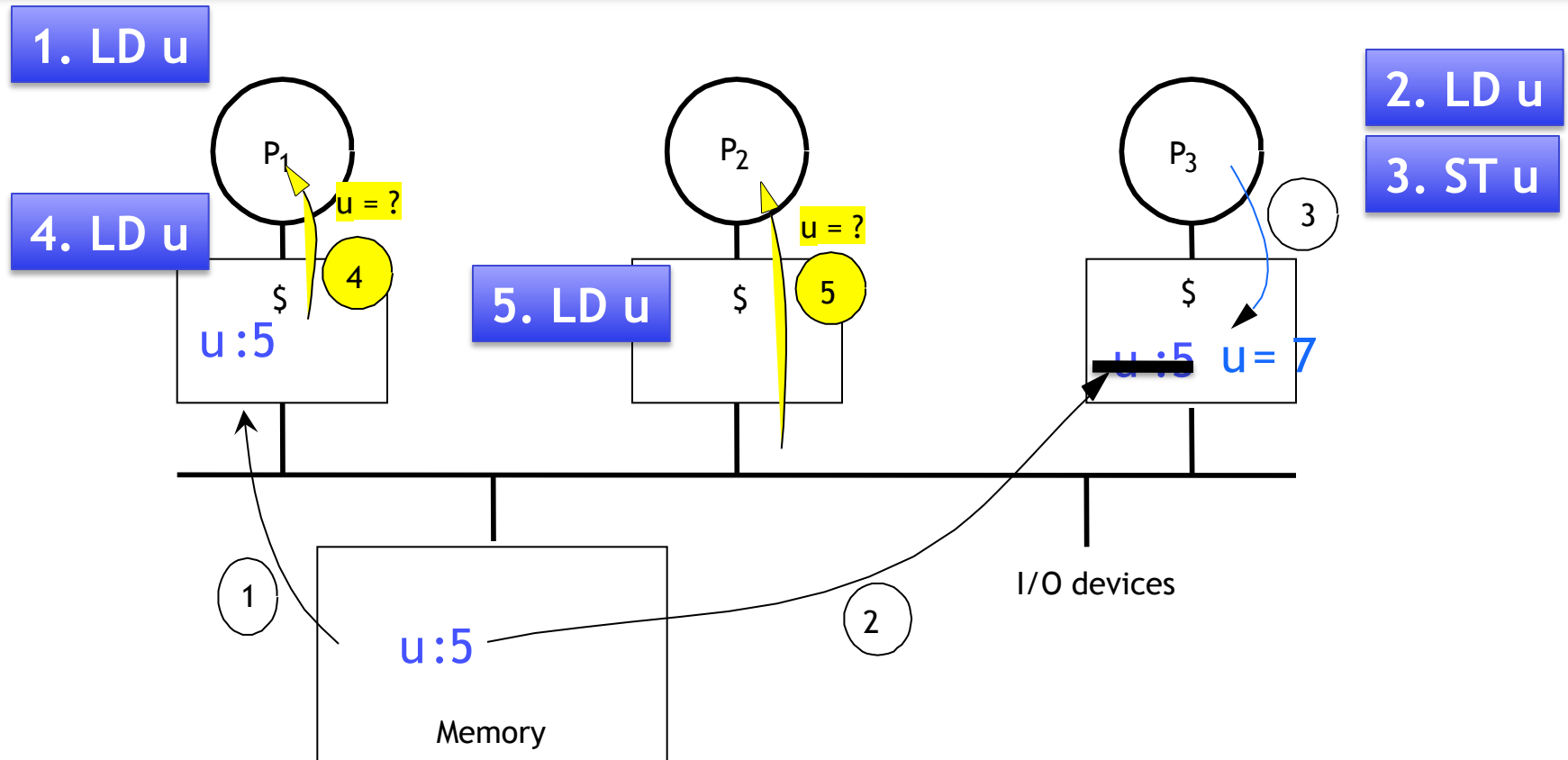
- Previously, we introduced multi-core parallelism.
  - Today we'll look at 2 things:
    - Cache coherence
    - Instruction support for synchronization.
  - And some pitfalls of parallelization.
  - And solve a few mysteries.

Pick up a  
handout !

Intel Core i7



# The Cache Coherence Problem



- Caches are critical to modern high-speed processors
- Multiple copies of a block can easily get inconsistent
  - Processor writes; I/O writes
- Processors could see different values for *u* after event 3

# Cache Coherence Invariant

---

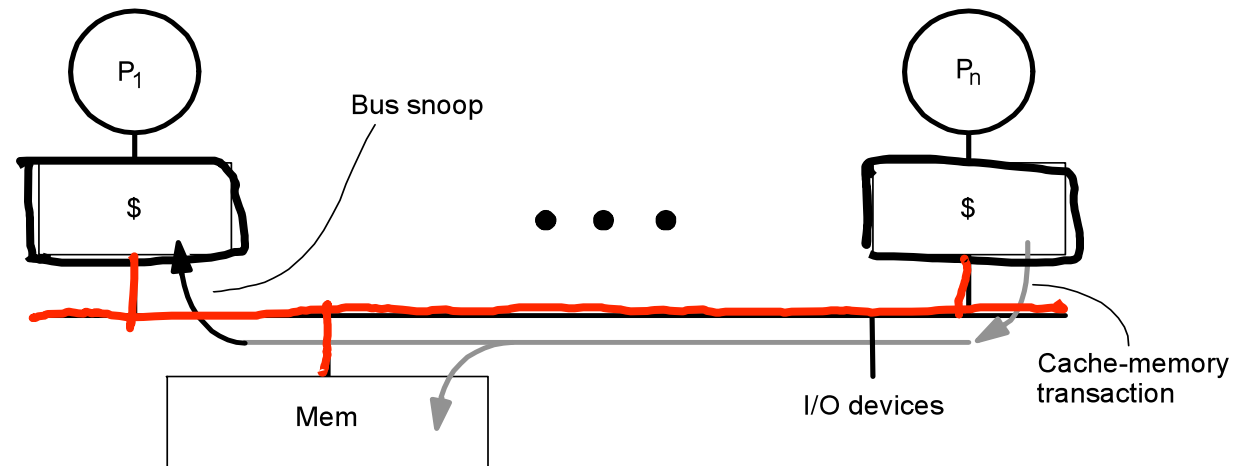
- Each block of memory is in exactly one of these 3 states:
  1. **Uncached:** Memory has the only copy
  2. **Writable:** Exactly 1 cache has the block and only that processor can write to it.
  3. **Read-only:** Any number of caches can hold the block, and their processors can read it.

**invariant** | in've(ə)rēənt |

noun Mathematics

a function, quantity, or property that remains unchanged when a specified transformation is applied.

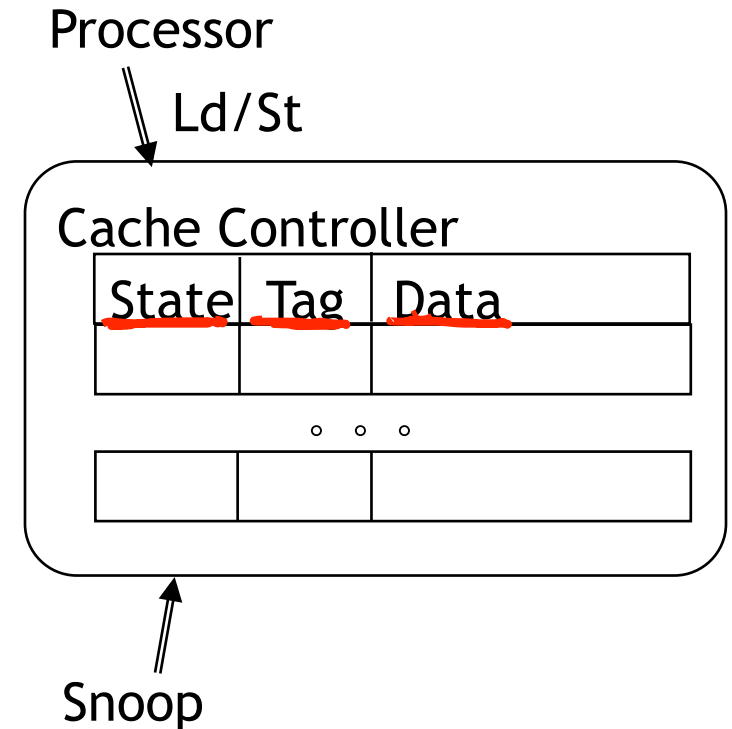
# Snoopy Cache Coherence Schemes



- Bus is a broadcast medium & caches know what they have
  - Cache controller “snoops” all transactions on the shared bus
    - Relevant transaction if for a block it contains
    - Take action to ensure coherence
      - Invalidate or supply value
- Depends on state of the block and the protocol

# Maintain the invariant by tracking “state”

- Every cache block has an associated state
  - This will supplant the valid and dirty bits
- A cache controller updates the state of blocks in response to processor and snoop events and generates bus transactions
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions



# MSI protocol

---

This is the simplest possible protocol, corresponding directly to the 3 options in our invariant

- Invalid State: the data in the cache is not valid. valid dirty
- Shared State: multiple caches potentially have copies of this data; they will all have it in **shared** state. Memory has a copy that is consistent with the cached copy. valid dirty read-only
- **Dirty** or Modified: only 1 cache has a copy. Memory has a copy that is inconsistent with the cached copy. Memory needs to be updated when the data is displaced from the cache or another processor wants to read the same data. valid dirty read or write

# Actions

---

## Processor Actions:

- Load
- Store
- **Eviction:** processor wants to replace cache block

## Bus Actions:

- **GETS:** request to get data in shared state
- **GETX:** request for data in modified state (*i.e.*, eXclusive access)
- **UPGRADE:** request for exclusive access to data owned in shared state

## Cache Controller Actions:

- **Source Data:** this cache provides the data to the requesting cache
- **Writeback:** this cache updates the block in memory

# MSI Protocol

---

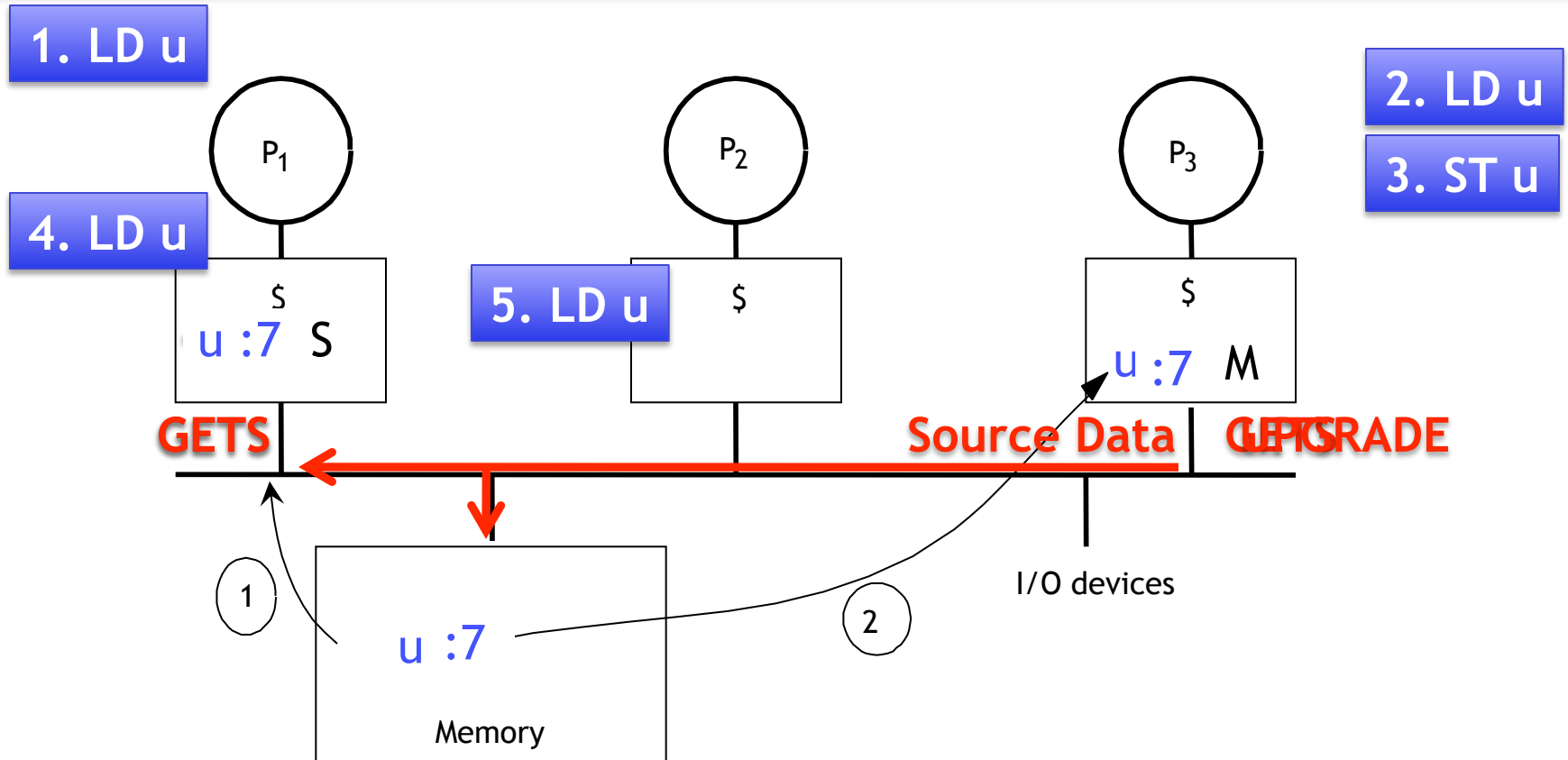
Invalid

Shared

Modified



# The Cache Coherence Problem **Solved**



# Real Cache Coherence Protocols

---

- Are more complex than MSI (see MESI and MEOSI)
- Some modern chips don't use buses (too slow)
  - Directory based: Alternate protocol doesn't require snooping
- But this gives you the basic idea.

*exclusive, clean*

*shared, dirty*

# A simple piece of code

---

```
unsigned counter = 0;
```

```
void *do_stuff(void * arg) {  
    for (int i = 0 ; i < 2000000000 ; ++ i) {  
        counter ++;  
    }  
    return arg;  
}
```

← adds one to counter

How long does this program take?

.45s

How can we make it faster?

try to parallelize

# A simple piece of code

---

```
unsigned counter = 0;
```

```
void *do_stuff(void * arg) {  
    for (int i = 0 ; i < 2000000000 ; ++ i) {  
        counter ++;  
    }  
    return arg;  
}
```

← adds one to counter

How long does this program take? Time for 2000000000 iterations

How can we make it faster? Run iterations in *parallel*

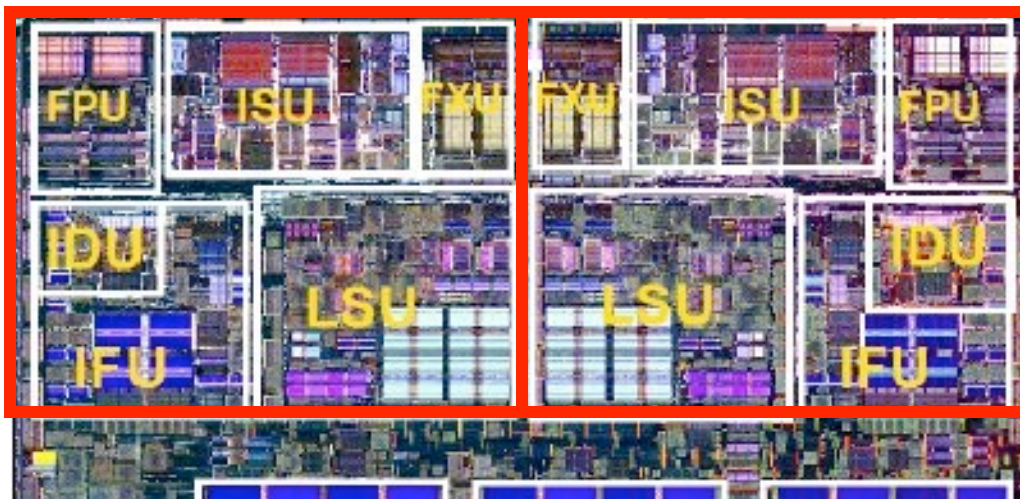
# Exploiting a multi-core processor

```
unsigned counter = 0;
```

```
void *do_stuff(void * arg) {  
    for (int i = 0 ; i < 2000000000 ; ++ i) {  
        counter ++;  
    }  
    return arg;  
}
```

Split for-loop across  
multiple threads running  
on separate cores

#1



#2

## How much faster?

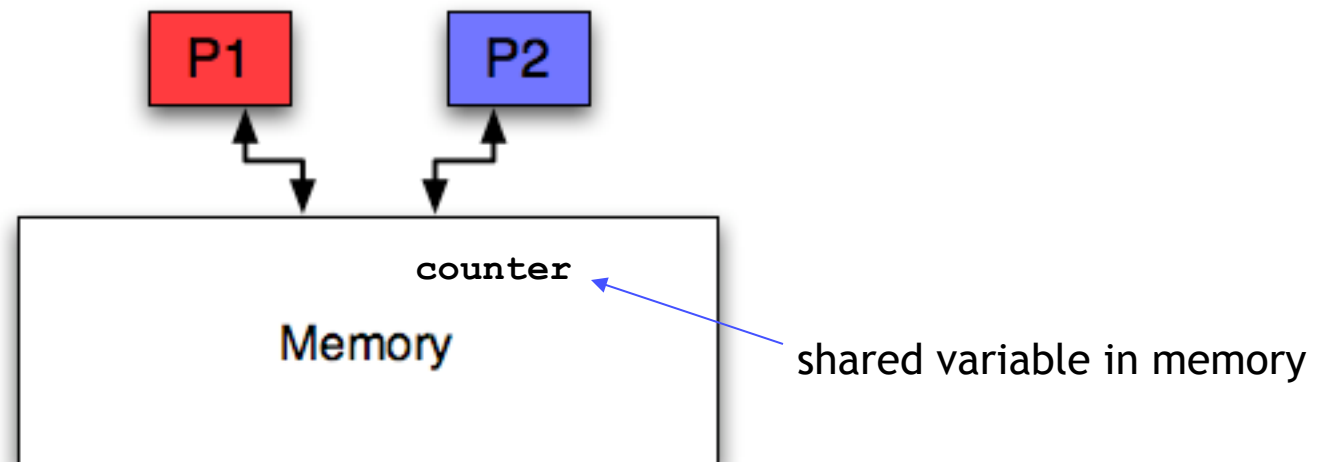
---

almost twice as fast

# How much faster?

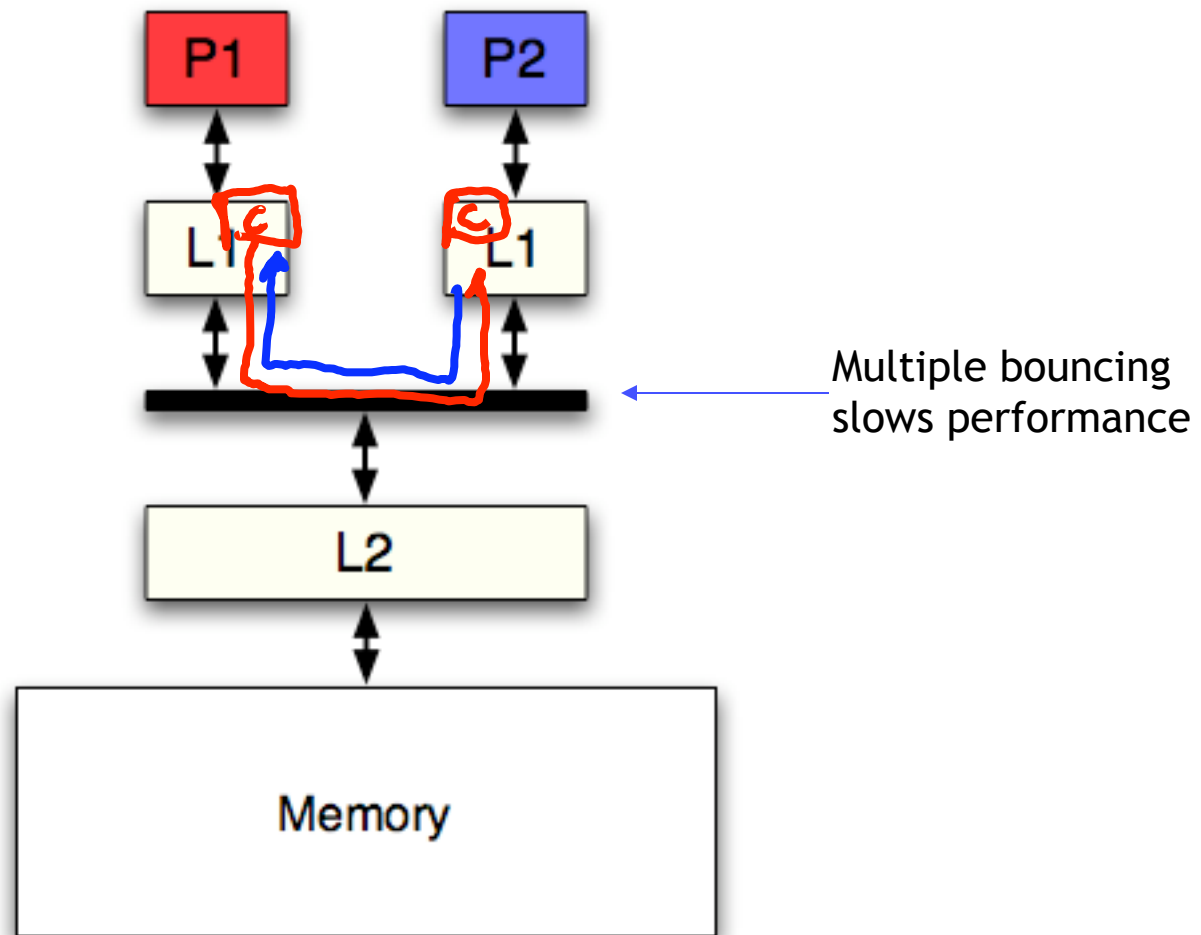
---

- We're expecting a speedup of 2
- OK, perhaps a little less because of Amdahl's Law
  - overhead for forking and joining multiple threads
- But its actually **slower!!** Why??
- Here's the mental picture that we have - two processors, shared memory



# This mental picture is wrong!

- We've forgotten about **caches**!
  - The memory may be shared, but each processor has its own L1 cache
  - As each processor updates `counter`, it bounces between L1 caches





# The code is not only slow, its WRONG!

- Since the variable `counter` is *shared*, we can get a **data race**
- Increment operation: `counter++` MIPS equivalent:  

```
lw    $t0, counter
addi  $t0, $t0, 1
sw    $t0, counter
```
- A data race occurs when data is **accessed** and **manipulated** by multiple processors, and the outcome depends on the sequence or timing of these events.

## Sequence 1

### Processor 1

```
lw    $t0, counter
addi  $t0, $t0, 1
sw    $t0, counter
```

### Processor 2

```
lw    $t0, counter
addi  $t0, $t0, 1
sw    $t0, counter
```

`counter` increases by 2

## Sequence 2

### Processor 1

```
lw    $t0, counter
addi  $t0, $t0, 1
```

```
sw    $t0, counter
```

### Processor 2

```
lw    $t0, counter
addi  $t0, $t0, 1
sw    $t0, counter
```

`counter` increases by 1 !!

# What is the minimum value at the end of the program?

desired = 200 million

100M

①

LD C → ∅  
ADD → 2  
ST → 1  
LD  
ST  
99.99 M

LD → 1

Context switch

ADD  
ST → 2

②

②

LD C → ∅

Content switched

ADD  
ST → 1

LD → 2

ADD  
ST → 2

⋮

ST → 100M

# Atomic operations

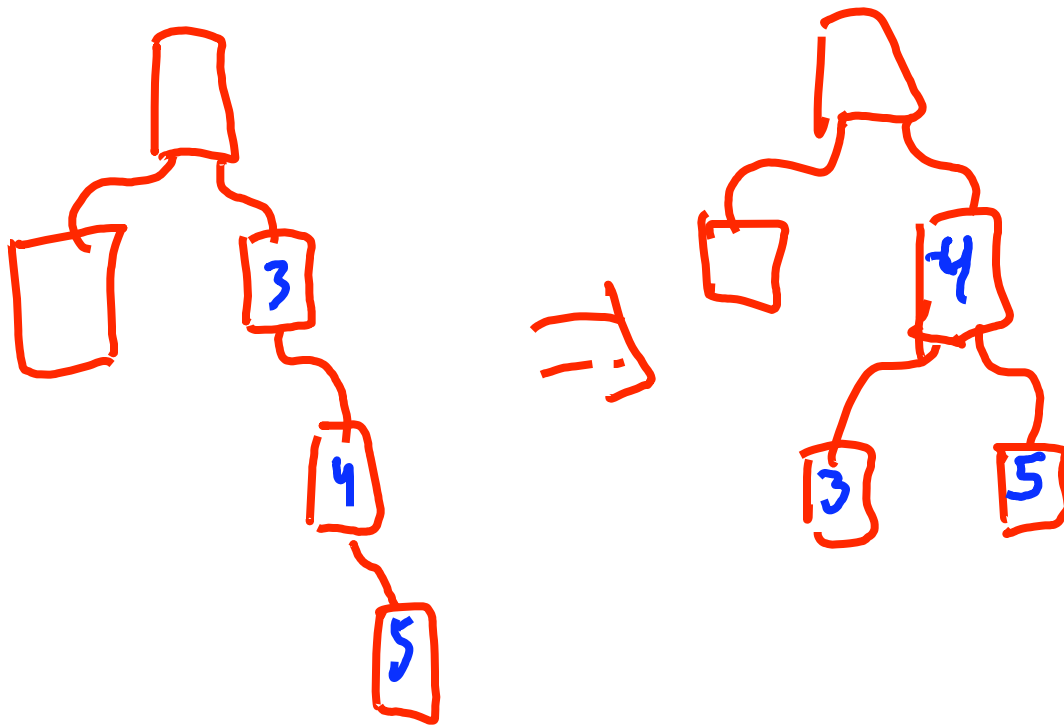
---

- You can show that if the sequence is particularly nasty, the final value of `counter` may be as little as 2, instead of 200000000.
- To fix this, we must do the load-add-store in a *single* step
  - We call this an atomic operation
  - We're saying: "Do this, and don't allow other processors to interleave memory accesses while doing this."
- "Atomic" in this context means "as if it were a single operation"
  - either we succeed in completing the operation with no interruptions or we fail to even begin the operation (because someone else was doing an atomic operation)
  - Furthermore, it should be "isolated" from other threads.
- x86 provides a "lock" prefix that tells the hardware:
  - "don't let anyone read/write the value until I'm done with it"
  - Not the default case (because it is slower!)

# What if we want to generalize beyond increments?

---

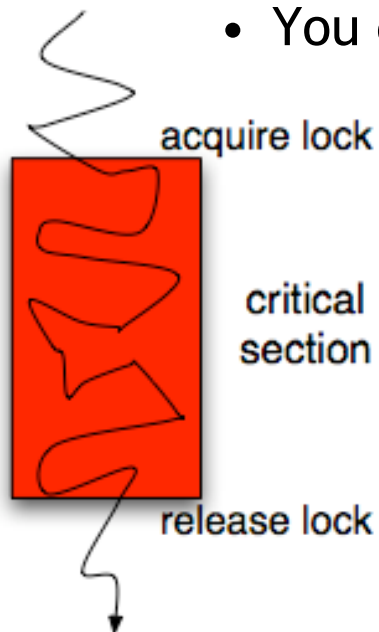
- The lock prefix only works for individual x86 instructions.
- What if we want to execute an arbitrary region of code without interference?
  - Consider a red-black tree used by multiple threads.



# What if we want to generalize beyond increments?

---

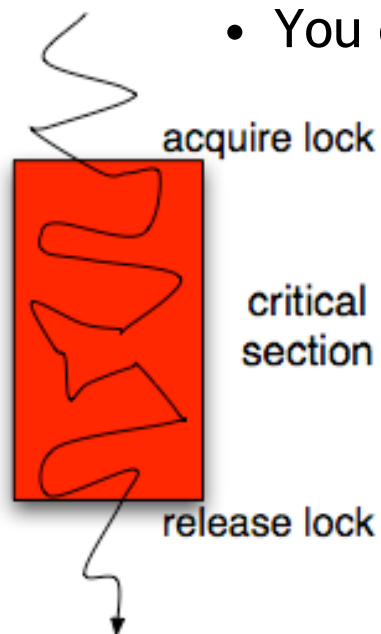
- The lock prefix only works for individual x86 instructions.
- What if we want to execute an arbitrary region of code without interference?
  - Consider a red-black tree used by multiple threads.
- Best mainstream solution: **Locks**
  - Implements **mutual exclusion**
    - You can't have it if I have it, I can't have it if you have it



# What if we want to generalize beyond increments?

---

- The lock prefix only works for individual x86 instructions.
- What if we want to execute an arbitrary region of code without interference?
  - Consider a red-black tree used by multiple threads.
- Best mainstream solution: **Locks**
  - Implement “mutual exclusion”
    - You can’t have it if I have, I can’t have it if you have it



**when lock = 0, set lock = 1, continue**

**lock = 0**

# Lock acquire code

---


## High-level version

```
unsigned lock = 0;

while (1) {
    if (lock == 0) {
        lock = 1;
        break;
    }
}
```

## MIPS version

```
spin: lw      $t0, 0($a0)
      bne     $t0, 0, spin
      li      $t1, 1
      sw      $t1, 0($a0)
```

 &lock

- What problem do you see with this?

lock = 1

## Race condition in lock-acquire

1  
spin: lw \$t0, 0(\$a0)  $\rightarrow \emptyset$   
bne \$t0, 0, spin  
li \$t1, 1  
sw \$t1, 0(\$a0)



2  
lw \$t0, 0(\$a0)  $\rightarrow \emptyset$   
bne  
li  
sw \$t1, 0(\$a0)





# Doing “lock acquire” atomically

---

- Make sure no one gets between load and store
- Common primitive: **compare-and-swap** (old, new, addr)
  - If the value in memory matches “old”, write “new” into memory

```
temp = *addr;  
if (temp == old) {  
    *addr = new;  
} else {  
    old = temp;  
}
```

- x86 calls it CMPXCHG (compare-exchange)
  - Use the lock prefix to guarantee it's atomicity

# Using CAS to implement locks

---

- Acquiring the lock:

lock\_acquire:

li \$t0, 0      # old

li \$t1, 1      # new

cas \$t0, \$t1, lock

beq \$t0, \$t1, lock\_acquire # failed, try again

- Releasing the lock:

sw \$0, lock

# Conclusions

---

- When parallel threads access the same data, potential for data races
  - Even true on uniprocessors due to context switching
- We can prevent data races by enforcing **mutual exclusion**
  - Allowing only one thread to access the data at a time
  - For the duration of a critical section
- Mutual exclusion can be enforced by locks
  - Programmer allocates a variable to “protect” shared data
  - Program must perform:  $0 \rightarrow 1$  transition before data access
  - $1 \rightarrow 0$  transition after
- Locks can be implemented with atomic operations
  - (hardware instructions that enforce mutual exclusion on 1 data item)
  - compare-and-swap
    - If address holds “old”, replace with “new”