Atomic Operations in Hardware

- Previously, we introduced multi-core parallelism.
  - Today we’ll look at instruction support for synchronization.
  - And some pitfalls of parallelization.
  - And solve a few mysteries.
unsigned counter = 0;

void *do_stuff(void * arg) {
    for (int i = 0 ; i < 200000000 ; ++ i) {
        counter ++;
    }
    return arg;
}

How long does this program take?

How can we make it faster?
A simple piece of code

```c
unsigned counter = 0;

void *do_stuff(void * arg) {
    for (int i = 0 ; i < 200000000 ; ++ i) {
        counter ++;
    }
    return arg;
}
```

How long does this program take? Time for 200000000 iterations

How can we make it faster? Run iterations in parallel
unsigned counter = 0;

void *do_stuff(void * arg) {
    for (int i = 0 ; i < 200000000 ; ++ i) {
        counter ++;
    }
    return arg;
}
How much faster?
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 it’s actually faster!! 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

![Diagram showing two processors, L1 caches, and L2 cache]
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.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor 1</strong></td>
<td><strong>Processor 2</strong></td>
</tr>
<tr>
<td><code>lw $t0, counter</code></td>
<td><code>lw $t0, counter</code></td>
</tr>
<tr>
<td><code>addi $t0, $t0, 1</code></td>
<td><code>addi $t0, $t0, 1</code></td>
</tr>
<tr>
<td><code>sw $t0, counter</code></td>
<td><code>sw $t0, counter</code></td>
</tr>
</tbody>
</table>

`counter` increases by 2  

`counter` increases by 1 !!
What is the minimum value at the end of the program?
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 get interrupted while doing this.”

- “Atomic” in this context means “all or nothing”
  - 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)
  - We really mean “atomic” AND “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 slow!)
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

```c
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)
```

- What problem do you see with this?
Race condition in lock-acquire

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

0
lw → 0
bne
li
sw → 1

lock

0
lw → 0
bne
li
sw → 1
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

```c
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
  fall through on success
  ```

- Releasing the lock:
  ```
  sw $a0, 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 → 1 transition before data access
    - 1 → 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”