The Java volatile Keyword

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Deferred Cancellation Example

Victim

```
public class Victim extends Thread {
  private boolean shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello World!");
    while(true) {
      // check if I should stop
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

Killer

```
. . .
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {}

. . .
```

- How come this program doesn't work????
  - Let's try it….

Why Doesn’t it Work???

- The reason why the previous program doesn’t work takes us down a path that:
  - You probably have never quite encountered
  - And yet is pervasive in computing

Why Doesn't it Work???

- The reason why the previous program doesn't work takes us down a path that:
  - You probably have never quite encountered
  - And yet is pervasive in computing
- It doesn't work because of performance optimizations
- We're in a world in which we're trying to get as much performance as possible from the machine
- As a result, we play hardware / compiler tricks that can break code!
  - We will come back to this with a vengeance later in the semester with other examples
- For now, let's just try to understand what's going on with our program!

What are the Symptoms?

- One thread is looking at the **`shouldStop`** variable in a **`Victim`** object
- Another thread is setting the **`shouldStop`** variable in that *same* object
- And yet, the first thread is not seing the update
- Conclusion: The two threads are not looking at the same memory location!
- But how could this be???

What are the Symptoms?

- One thread is looking at the **`shouldStop`** variable in a **`Victim`** object
- Another thread is setting the **`shouldStop`** variable in that *same* object
- And yet, the first thread is not seing the update
- Conclusion: The two threads are not looking at the same memory location!
- But how could this be???
- One possibility: compiler optimizations

Compiler Optimization #1

- Say you're a compiler, and you look at the victim's **`run()`** method to optimize
  - Compilers are often myopic: they just look at code in methods, without understanding or analyzing the full program

Victim

```
public void run() {
    System.out.println("Hello World!");
    while(true) {
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
}
```

- Any idea how a compiler may optimize this?
  - Say that the goal is to run as many iterations of the while loop as possible per time unit

Compiler Optimization #1

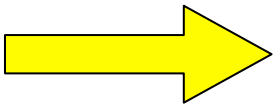
- Say you're a compiler, and you look at the victim's **`run()`** method to optimize
  - Compilers are often myopic: they just look at code in methods, without understanding or analyzing the full program

Victim

```
public void run() {
    System.out.println("Hello World!");
    while(true) {
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
}
```

- In the above, we have an infinite loop, and at each iteration we check the value of a variable that does not change!
  - Yes, as a human, I know that another thread may change it, but as a compiler, I don't
- So the check is useless and we optimize the loop as follows

```
while(true) {
   if (shouldStop) {
     break;
   }
}
```



```
if(not shouldStop) {
   while (true){ }
}
```

Compiler Optimization #1

- Say you're a compiler, and you look at the victim's **`run()`** method to optimize
  - Compilers are often myopic: they just look at code in methods, without understanding or analyzing the full program

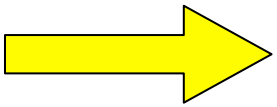
Victim

```
public void run() {
    System.out.println("Hello World!");
    while(true) {
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
}
```

- In the above, we hav…ch iteration we check the value of a …ge!
  - Yes, as a human, I k…nge it, but as a compiler, I don't
- So the check is usele…p as follows

The victim doesn't check whether it should stop!!

```
while(true) {
   if (shouldStop) {
     break;
   }
}
```



```
if(not shouldStop) {
   while (true){ }
}
```

Compiler Optimization #2

- Say the compiler doesn't do the previous optimization, for some reason
- It may choose to keep variable **`shouldStop`** in a register after entering the loop!
  - As you might have done writing assembly by hand in ICS312/ ICS331

Victim

```
public void run() {
    System.out.println("Hello World!");
    while(true) {
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
}
```

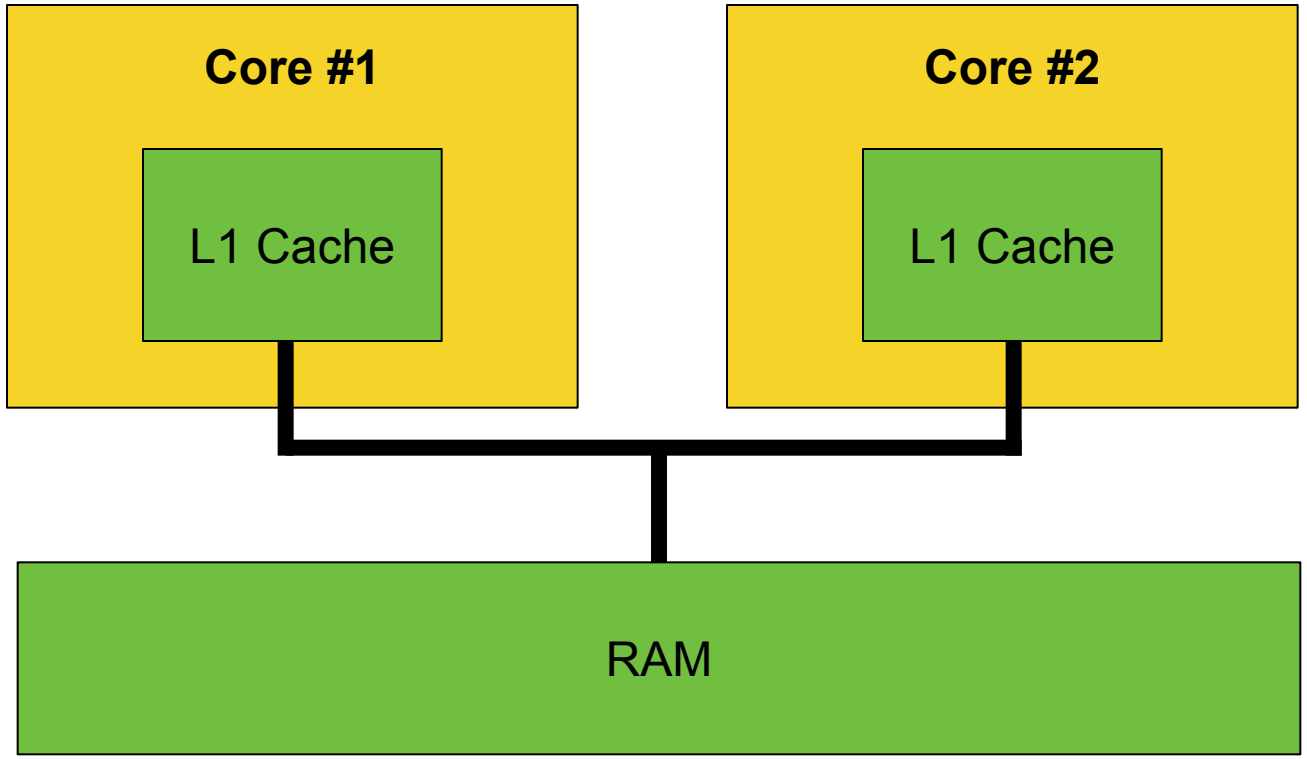
- In this case, it doesn't matter that another thread updates some RAM location!!
- The victim just looks at its own register forever
  - Registers are private to a thread
  - (Remember your ICS332)

Our Broken Program

- In our program, the victim never dies!
- Therefore, it's possible that our compiler does one or both of the previous optimizations
  - Difficult to check, as we'd have too look at (i.e., disassemble) the machine code produced by the JIT (Just-In-Time) compiler from the byte code produced by the Java compiler!
- But **even** if it doesn't/cannot optimize, e.g., for code that's not as simple, there could still be a problem in which the Victim reacts late to the termination request!
- This has to do with caches!

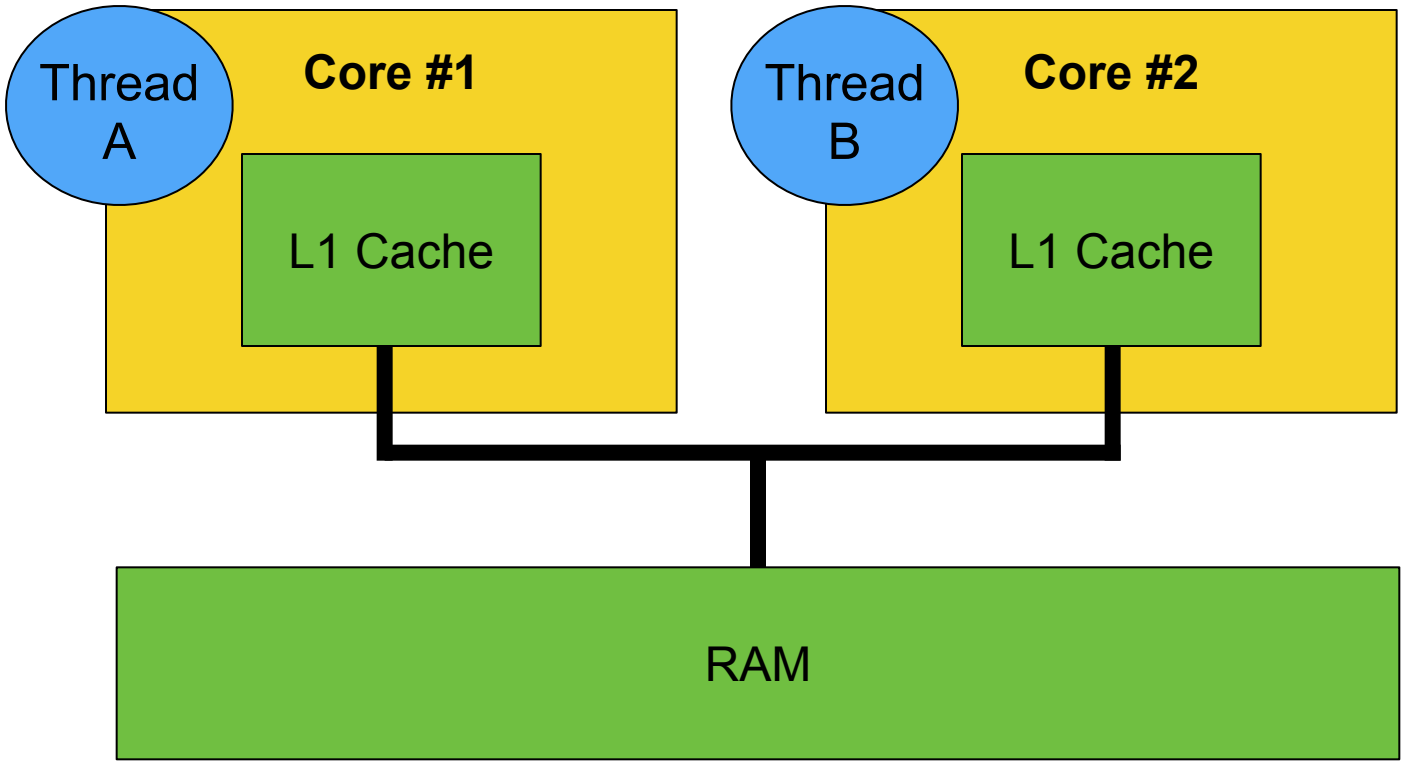
Caching in a NutShell

- Classical cache hierarchy



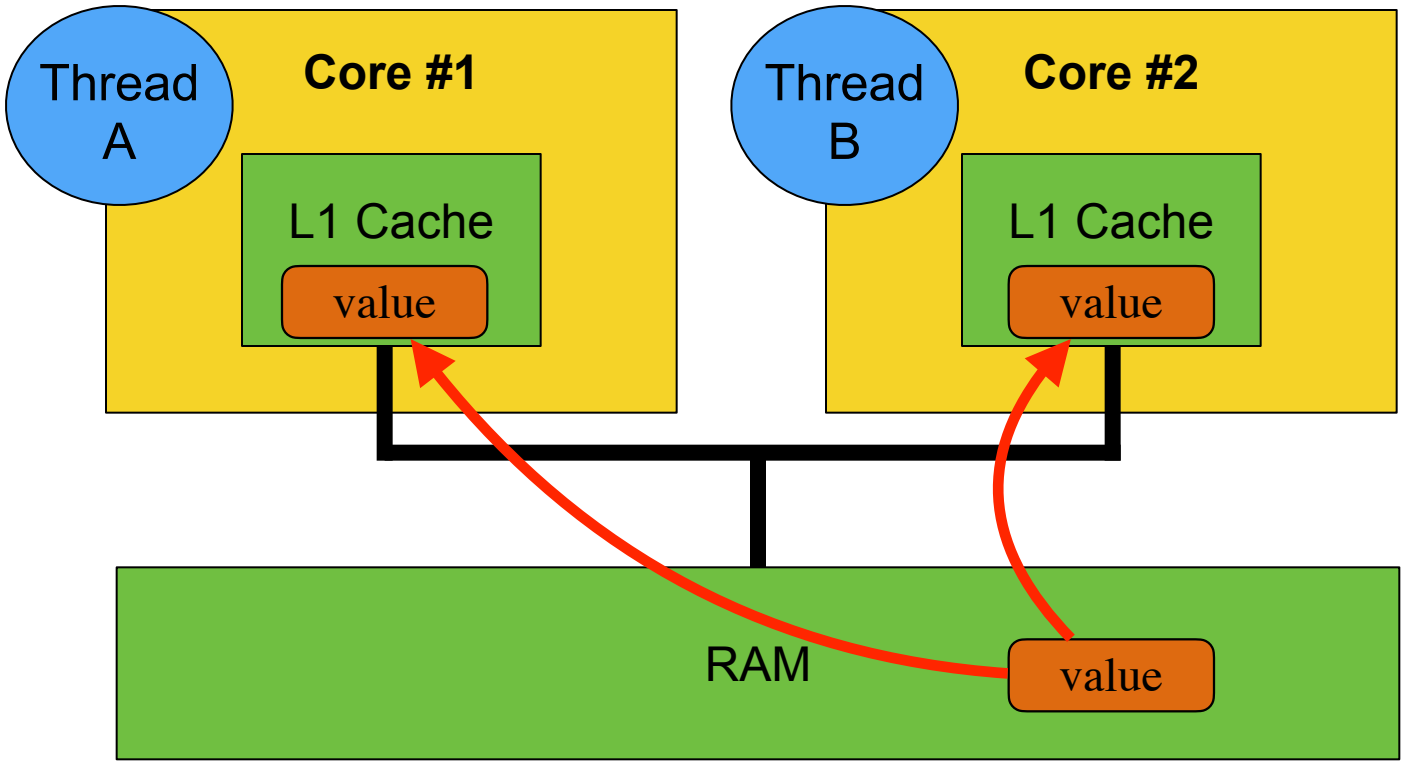
Caching in a NutShell

- One thread on each core



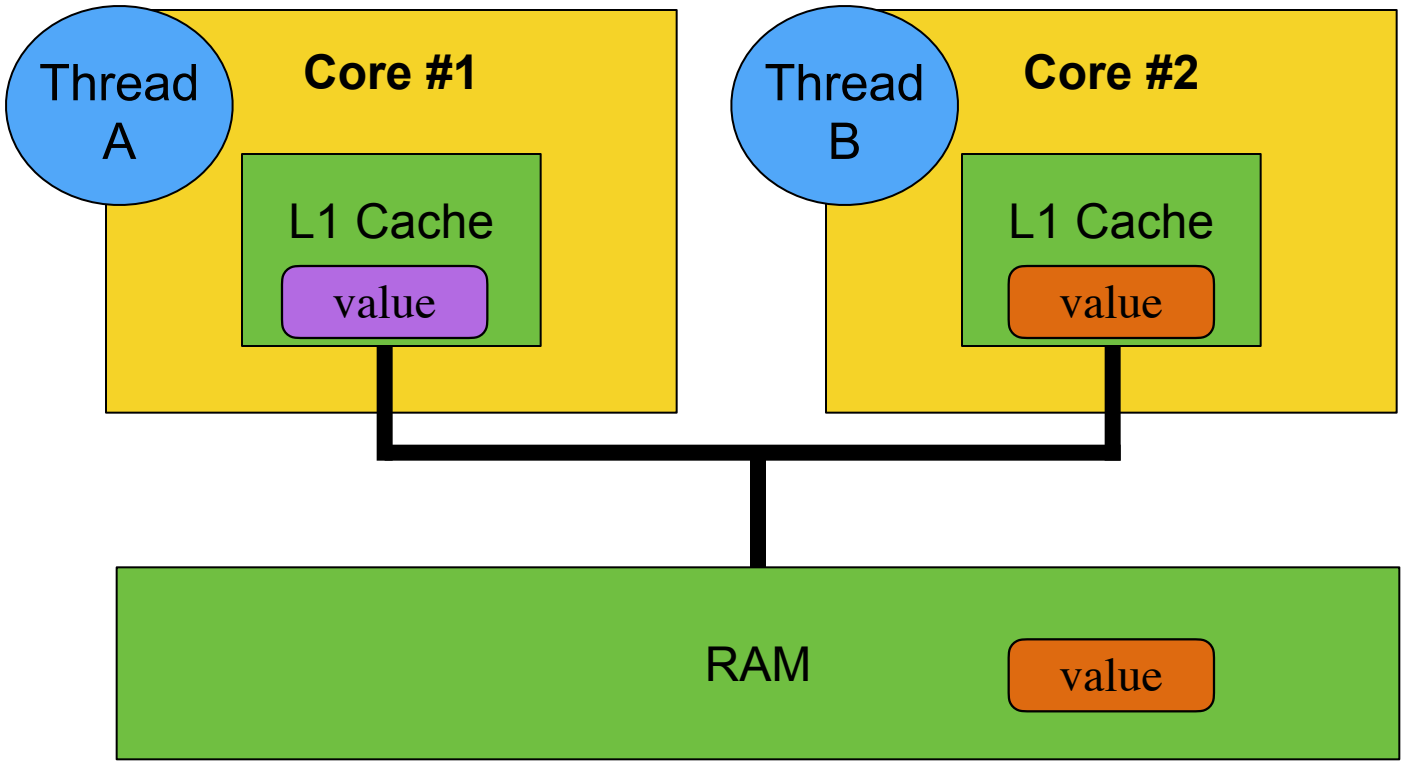
Caching in a NutShell

- Both threads read a value, which is copied into cache



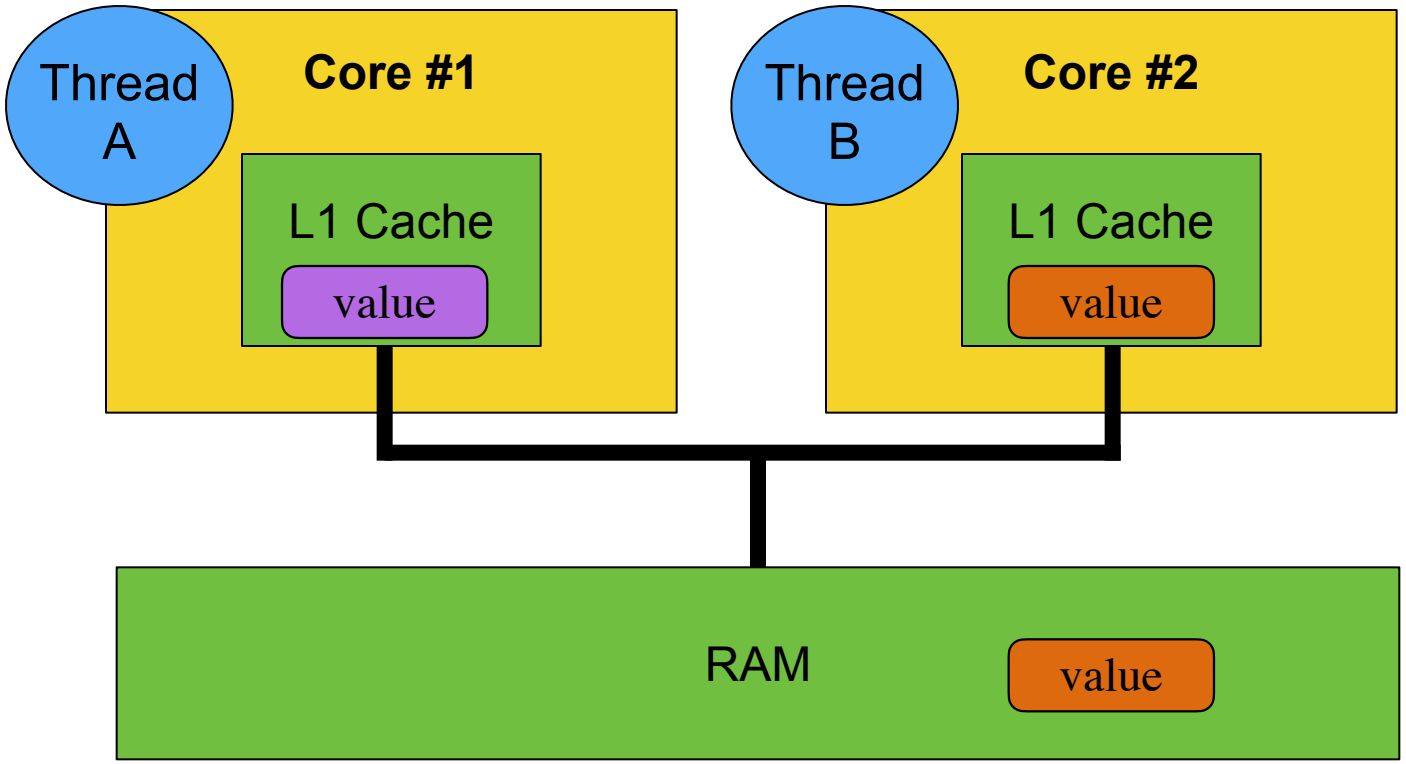
Caching in a NutShell

- Thread A modifies the value
- Now, we have incoherent data!



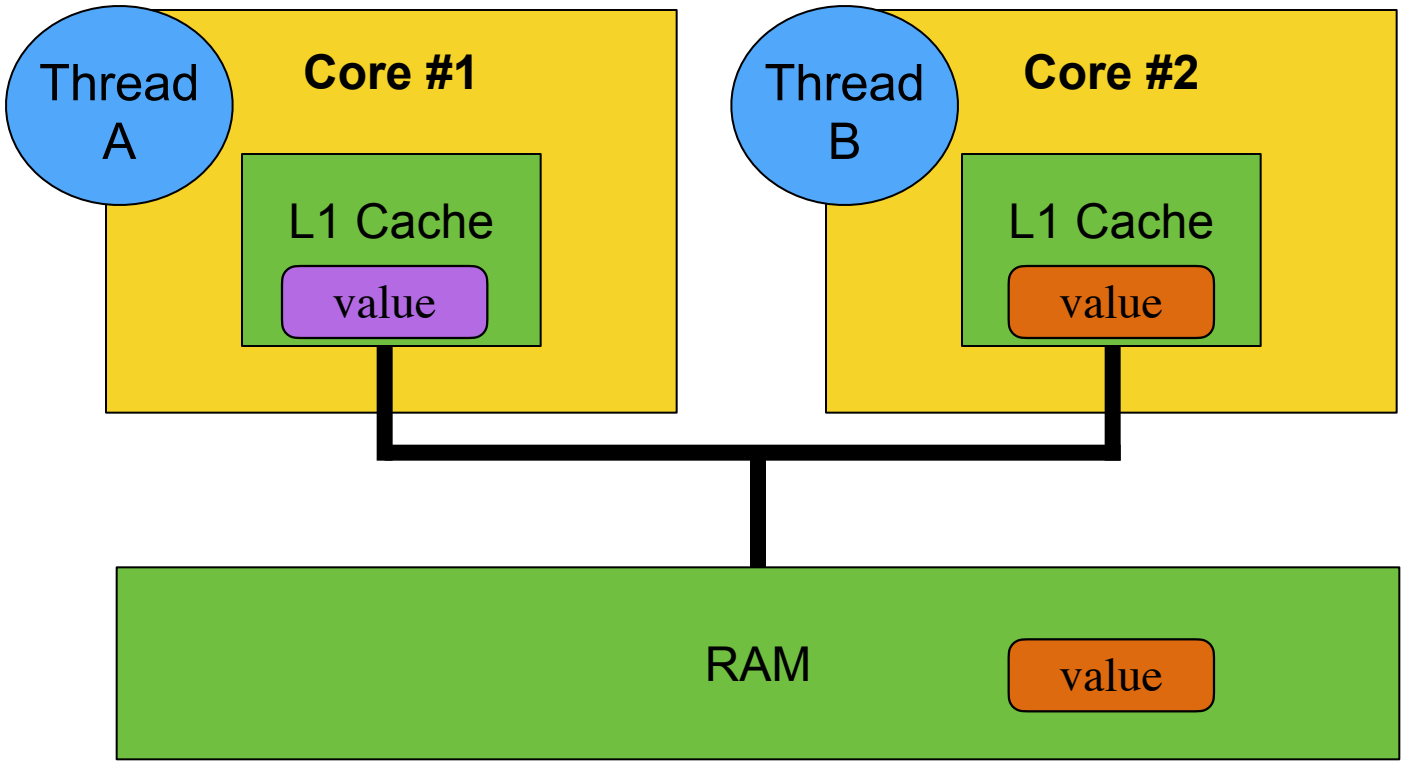
Caching in a NutShell

- Our hardware implements **cache coherency**
  - **Invalidate** the value in Core #2's cache, so that next time Thread B accesses the value, it will not read the one from Core #2's cache (and get the one from Core #1's cache)
  - See a hardware course / textbook for the gory details (and a little bit in this course later in the semester)



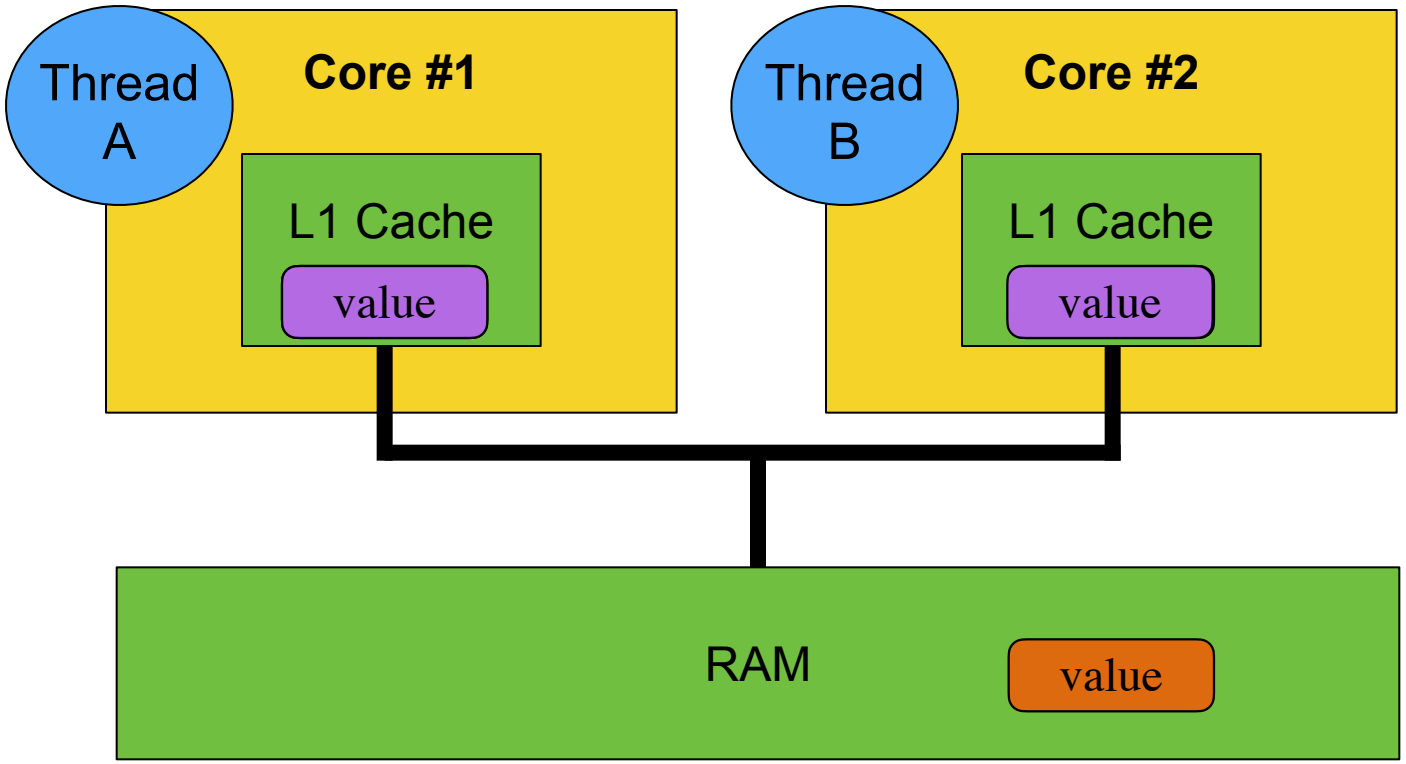
Caching in a NutShell

- But cache coherency is not immediate!
- So Thread B may read a stale value!
  - If Thread B does 1 minute of work in between checks, then the cancellation could be deferred by 1 extra minute!



Caching in a NutShell

- Eventually, the system “catches up”, and all is well until next time



Cache Coherency and Stale Data

- Our processors do not implement *sequential consistency*
  - The cores do not see the same ordering of memory writes
  - Therefore a thread could read "stale" data
- This is because trying to enforce sequential consistency is a speed killer
  - Every time somebody writes, stop everyone, update everybody's cache, and resume everyone, so that we're all on the same page
  - If you keep doing that, your programs are slooooooow
- So for the sake of speed, we let threads get stale data with low probability
  - Turns out, we typically don't care! Amazingly! (Scarily?)
- **But, sometimes, of course, we need sequential consistency!**
  - Like in our broken Java program, where after the killer has set `shouldStop` to true, the victim should see that value next time it accesses the variable
- So, what do we do????

Memory Fences

- Our processors provide us with memory fence instructions
  - also called memory barriers
- You can think of memory fences as expensive “clean up” instructions that make all memory consistent across cores
  - Usable only at the assembly level
- If you were to put a memory fence instruction after every variable write and before every variable read you’d have sequential consistency and very slooooooow programs
- The design principle: make things fast by default, and if programmers really need sequential consistency, then it’s on them to enable it (at the expense of speed)
  - But if as a programmer you don't know any of this, you’re in trouble!

So what do we need?

- To fix our program so that it is as responsive as possible we need two things
  - #1: Tell the compiler to not optimize the code in a way would remove memory accesses to the **`shouldStop`** variable
  - #2: Insert memory fence instructions before reads from the variable and after writes to the variable
    - So that we can have sequential consistency for accesses to that variable
- And Java gives us an easy way to do this…

The `volatile` keyword

- If we declare the **`shouldStop`** variable **`volatile`**, we get **both the things we need!**
  - i.e., the Java compiler won't optimize, and memory fence instructions are inserted

The `volatile` keyword

- If we declare the **`shouldStop`** variable **`volatile`**, we get **both the things we need!**
  - i.e., the Java compiler won't optimize, and memory fence instructions are inserted

```
public class Victim extends Thread {
  private volatile boolean shouldStop = false;

  public void tellToStop() {
    this.shouldStop = true;
  }

  public void run() {
    System.out.println("Hello
    while(true) {
      // check if I should st
      if (shouldStop) {
        break;
      }
    }
    System.out.println("AAARGH!");
  }
}
```

makes the program work!

```
// create the victim thread
Victim victim = new Victim();
victim.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
}

// tell the thread to stop
victim.tellToStop();

// Wait for victim
// to really have stopped
try {
  victim.join();
} catch (InterruptedException e) {
}
```

`volatile` Example

Some Class

```
public class SomeClass {
  private int var1;
  private volatile int var2;

  public int get1() {
    return var1;
  }
  public void set1(int v) {
    var1 = v;
  }
  public int get2() {
    return var2;
  }
  public void set2(int v) {
    var2 = v;
  }
}
```

Threads

```
SomeClass stuff;

// Thread 1
stuff.set1(12);
. . .
stuff.set2(42);
. . .

// Thread 2
stuff.get1();
. . .
stuff.get2();

. . .
```

Cheap, but may get stale value (or never!)

Expensive, but guaranteed to get latest value

How Slow is it???

- The course Web site has a program called **`VolatileStress.java`**
- Let's look at it, run it, and get a sense of how expensive memory fences are…

- Take-away: don't use **`volatile`** when you don't need it!

Conclusion

- Many Java developers don't know much about **`volatile`**
  - And there is a lot of confusion, misinformation, misunderstanding out there (e.g., on StackOverflow)
- Yet **`volatile`** can be crucial (as in previous slides)
- The reason why you might get by without **`volatile`** is that what it does for us is also done (under the cover) in other situations
  - e.g., in our broken program that hangs, if we add print statements, then it works!
  - We'll talk more about why that is later!
- So, **`volatile`** is not always needed and in fact some developers have not heard of it at all
- Until the day it is needed and then all hell breaks loose