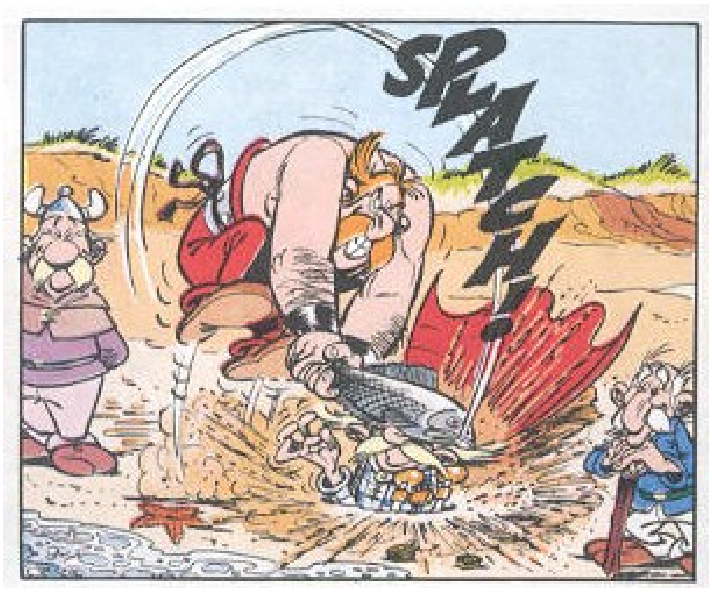
TDT4200 Parallel Programming

Bart van Blokland

Lecture 7

Feedback



Thanks a lot to everyone who sent us feedback! Much appreciated!

We really do need a reference group, though.

And yes, that’s indeed the *bard* receiving constructive comments on potential avenues for further improvement.

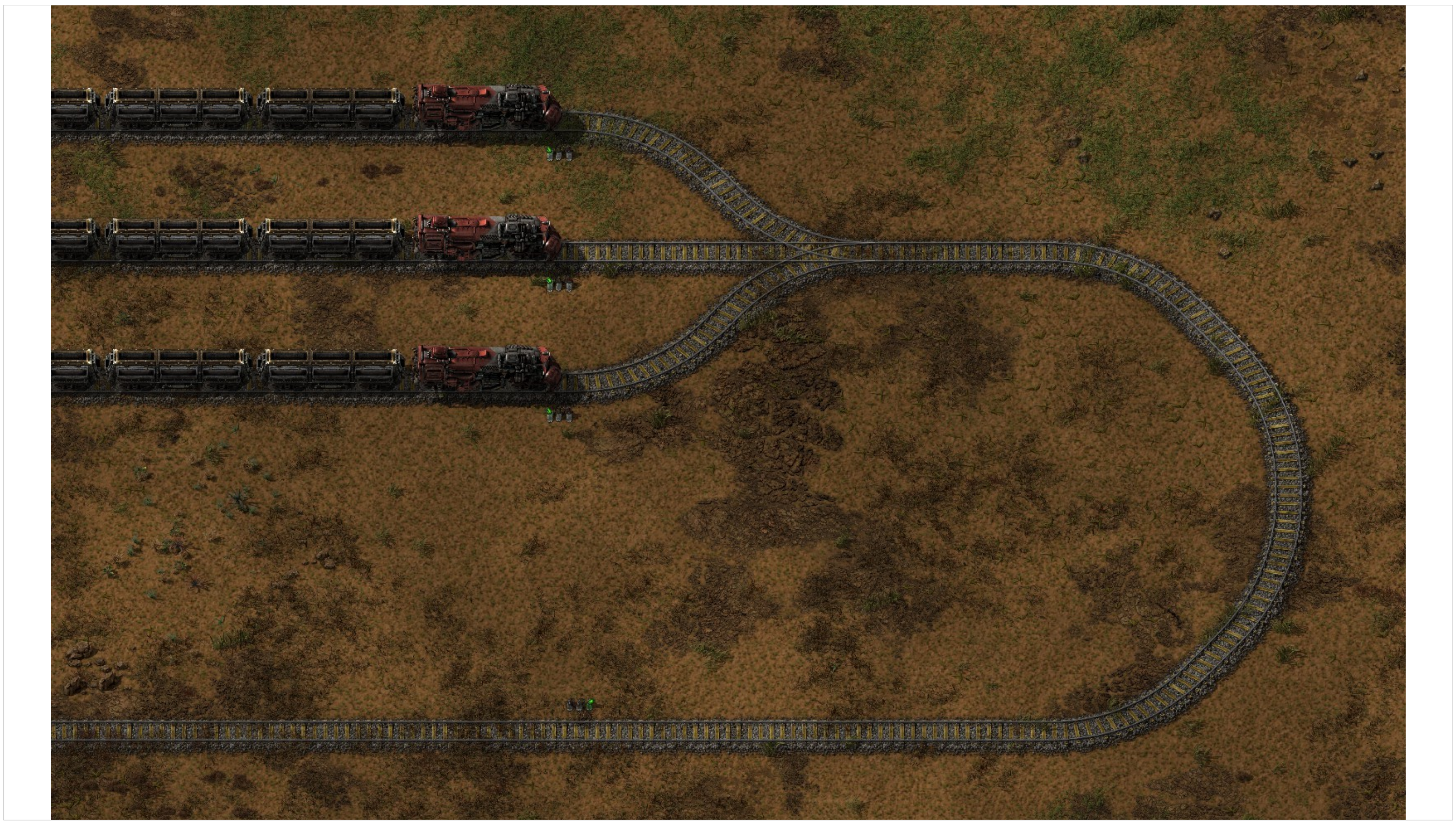
Last week:

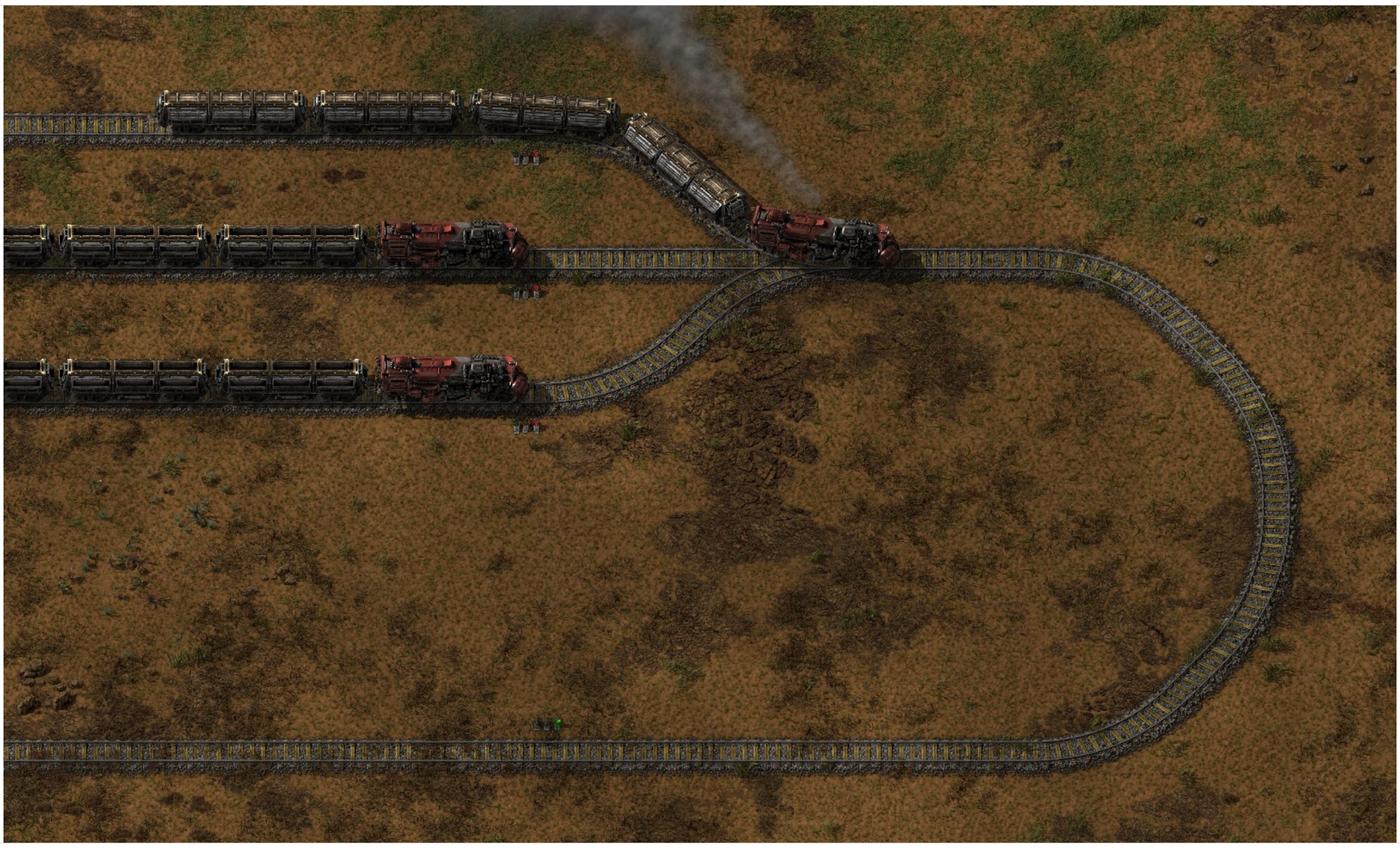
Locks and Atomics

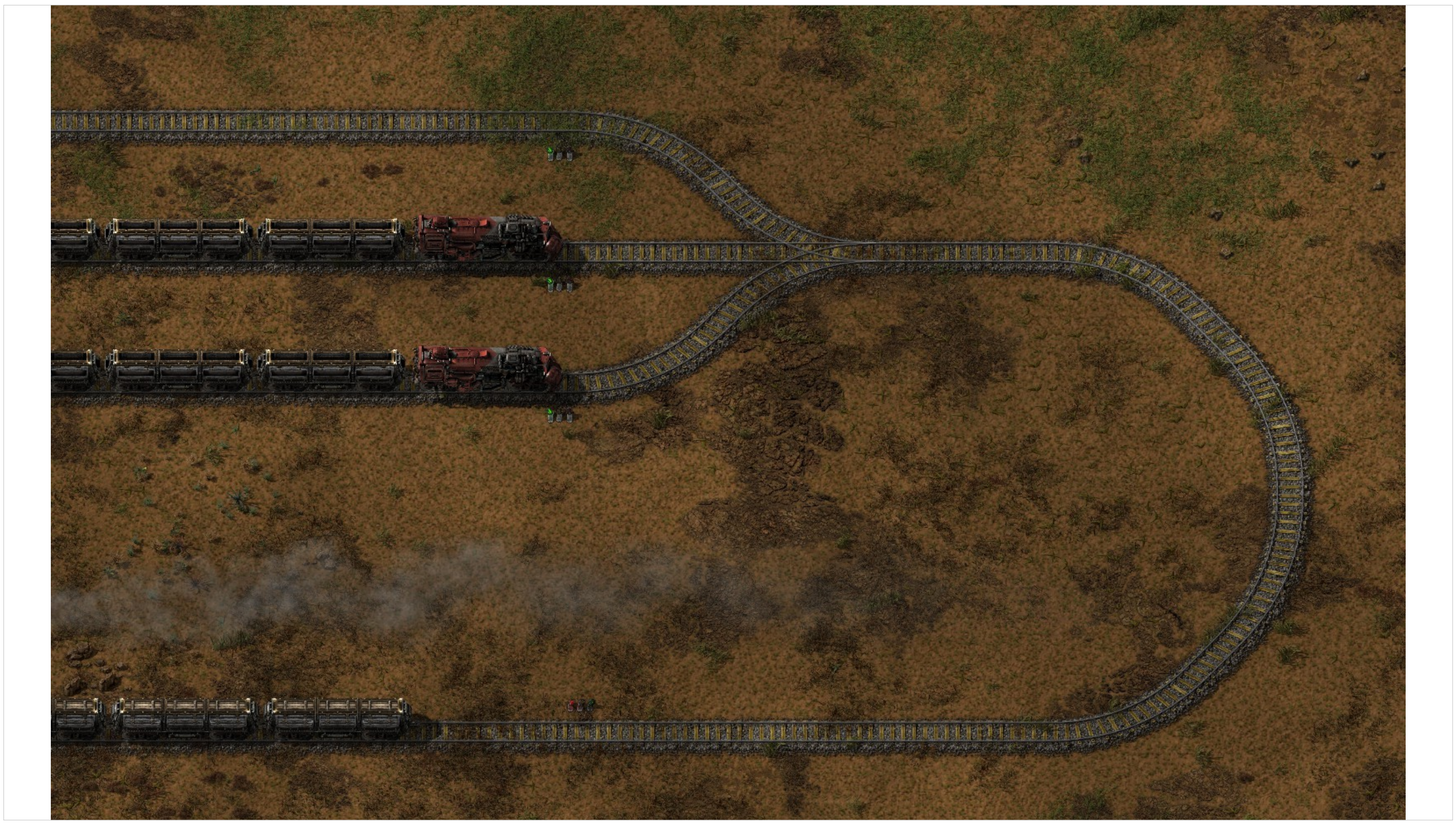
Semaphore

Semaphore: Let n threads through

Mutex: Let 1 thread through

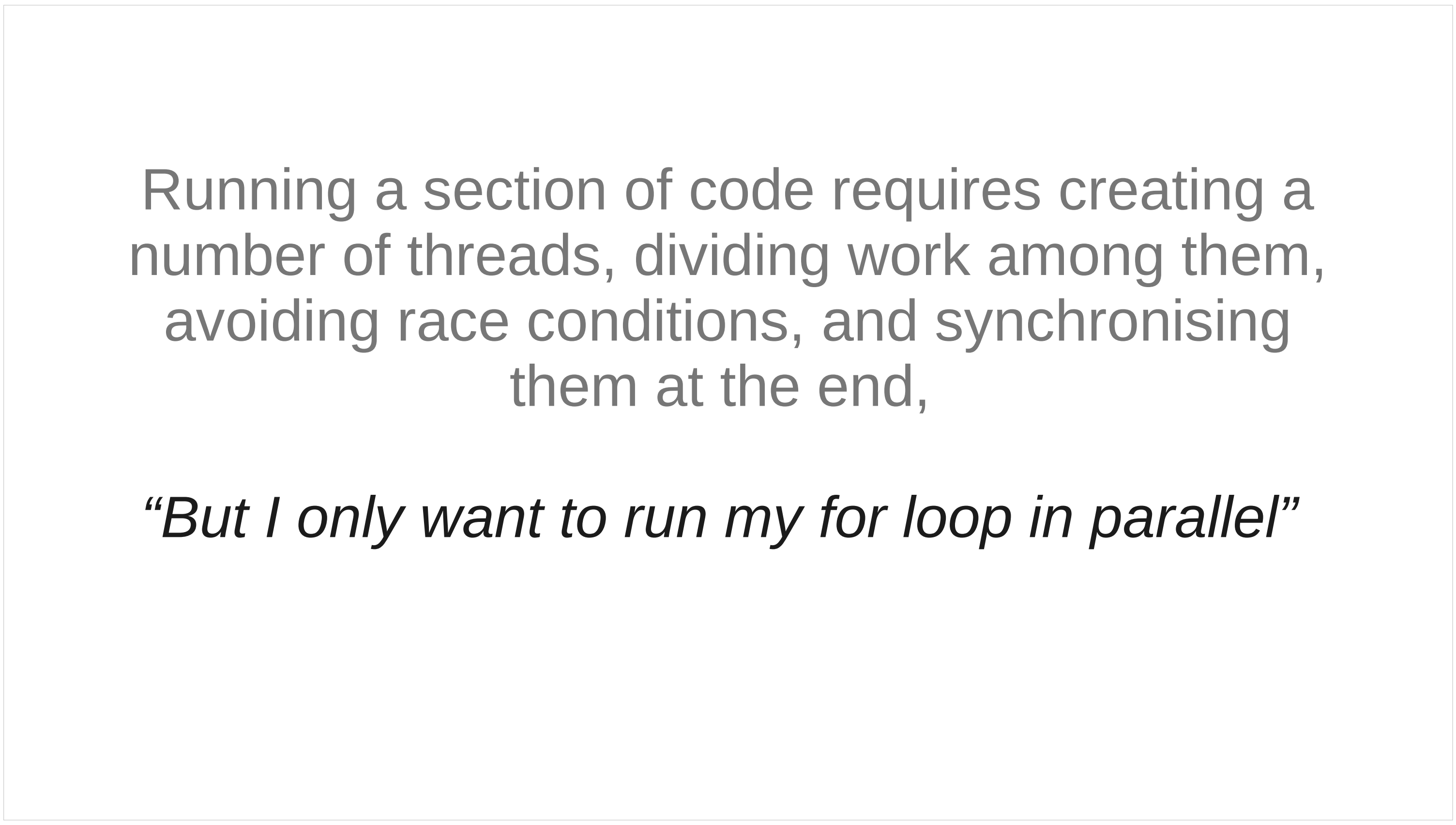






Atomic variable:

guarantees a write is seen by all cores



Running a section of code requires creating a number of threads, dividing work among them, avoiding race conditions, and synchronising them at the end,

“But I only want to run my for loop in parallel”

In practice, a lot of parallel computation comes down to “I want to run a single expensive loop in parallel”. This is where OpenMP is great, which is the objective for today.

```
for(int i = 0; i < veryLargeNumber; i++) {
    expensiveComputation();
}
```

OpenMP®

Enabling HPC since 1997

But first:

- Condition variables and Barriers

- Some more threads

- Recognising race conditions

Condition Variables and Barriers

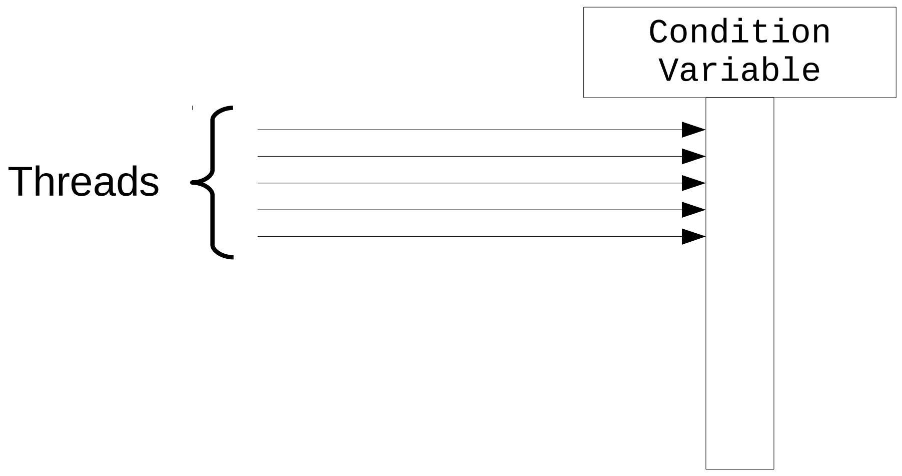
Condition variable:

A data object that allows a thread to suspend execution until a certain *condition* occurs.

Three operations:

Wait
Notify one
Notify all

Threads sleep while waiting.
No busy waiting!

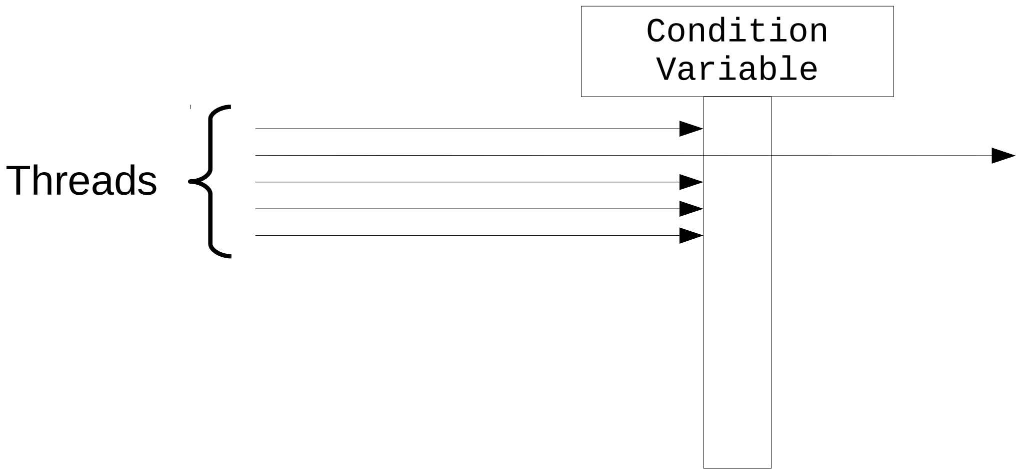


Condition variables have three operations: wait, notify one, and notify all.

Waiting on a condition variable causes the thread to go to sleep. The operating system handles waking it up from here on out. This avoid having the thread do busy waiting, consuming CPU resources unnecessarily.

Three operations:

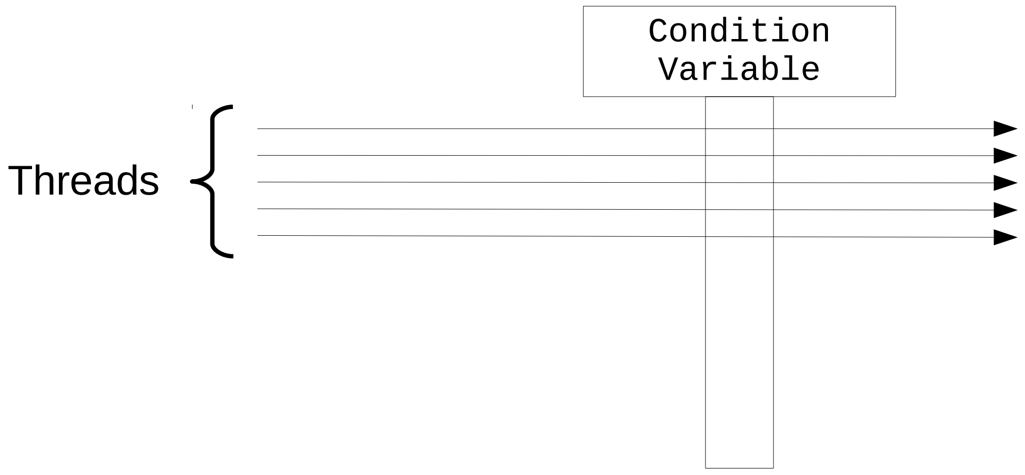
Wait
Notify one
Notify all



Notify one lets one thread through. Usually this is first come first serve to make sure all threads can go through eventually.

Three operations:

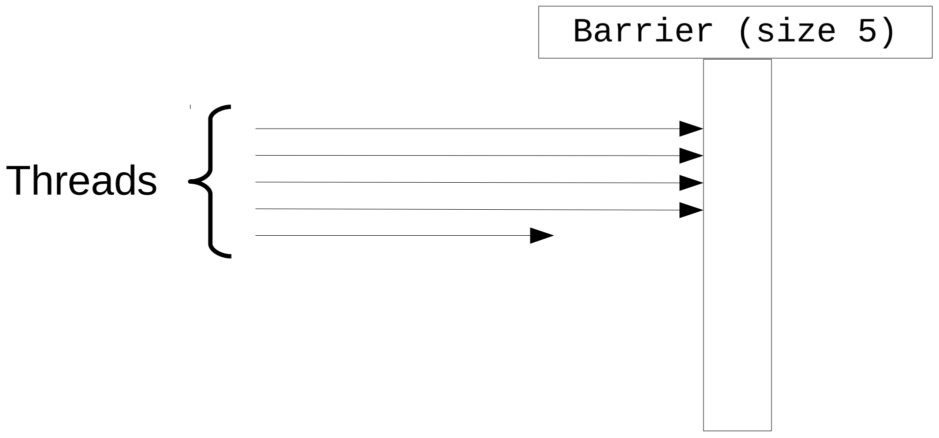
Wait
Notify one
Notify all



Notify all opens the floodgates and lets all threads loose.

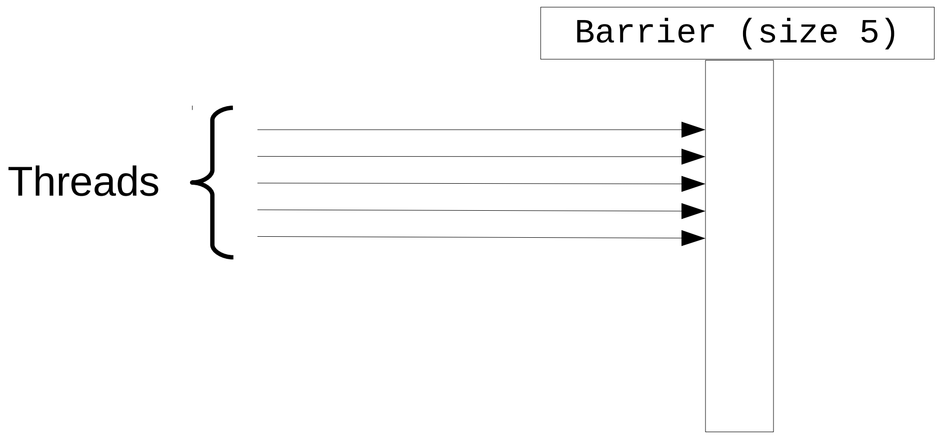
Code sample goes here.. Still missing for the moment though :(

Barrier

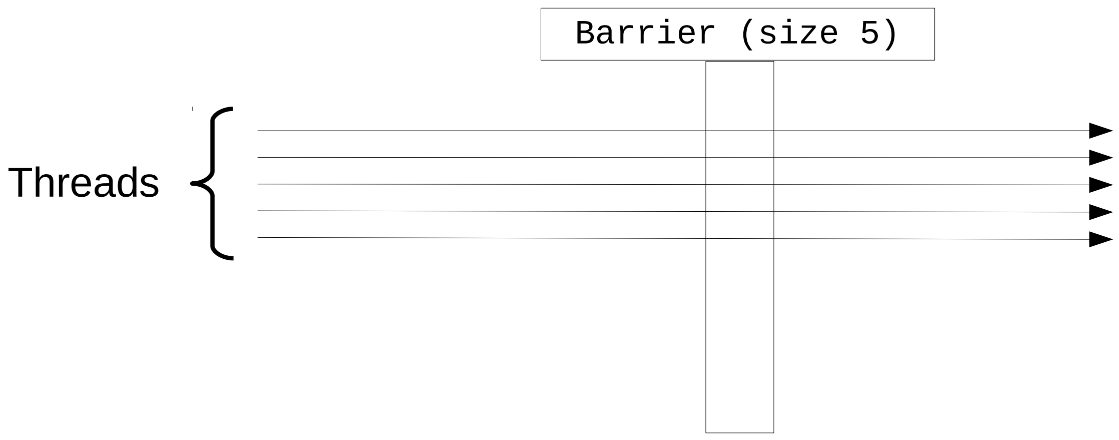


A barrier is an extension to a condition variable. It waits until a predetermined number of threads are waiting on it. Once it has collected that many, it automatically notifies all threads.

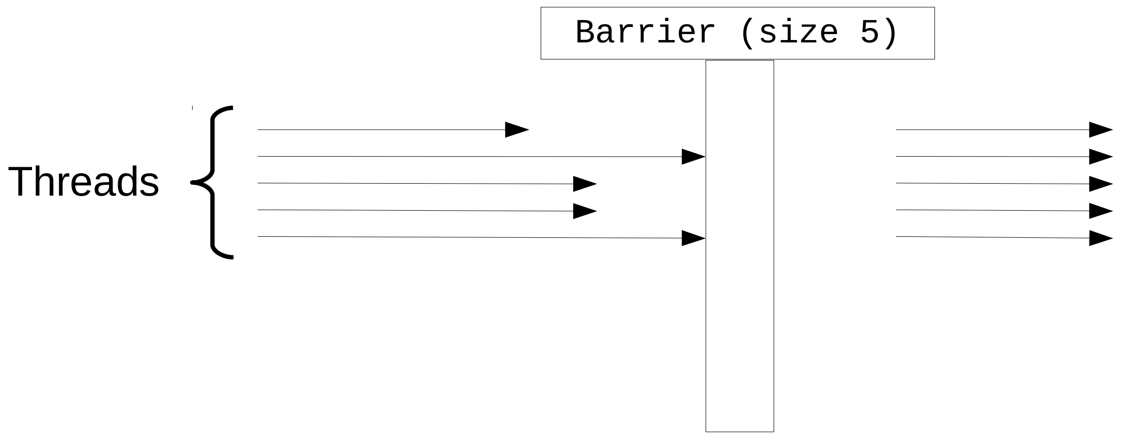
Barrier



Barrier



Barrier



After letting all threads through, it resets to wait until the specified number of threads are waiting on it again.

More on Threads and Locks

Reentrant lock:

A lock which does not block if the thread has already acquired it previously

std::mutex is **not** reentrant.

std::recursive_mutex **is**.

The reentrant property refers to whether a thread can acquire a (mutex) lock multiple times without deadlocking.

Modifying thread priorities is asking for trouble

(can starve threads)

Fiddling with thread priorities on your operating system can open a whole pandora’s box worth of concurrency problems. Avoid doing so unless you’re really sure of what you’re doing.

Copy on write data structures

Can save memory when sharing resources between threads.

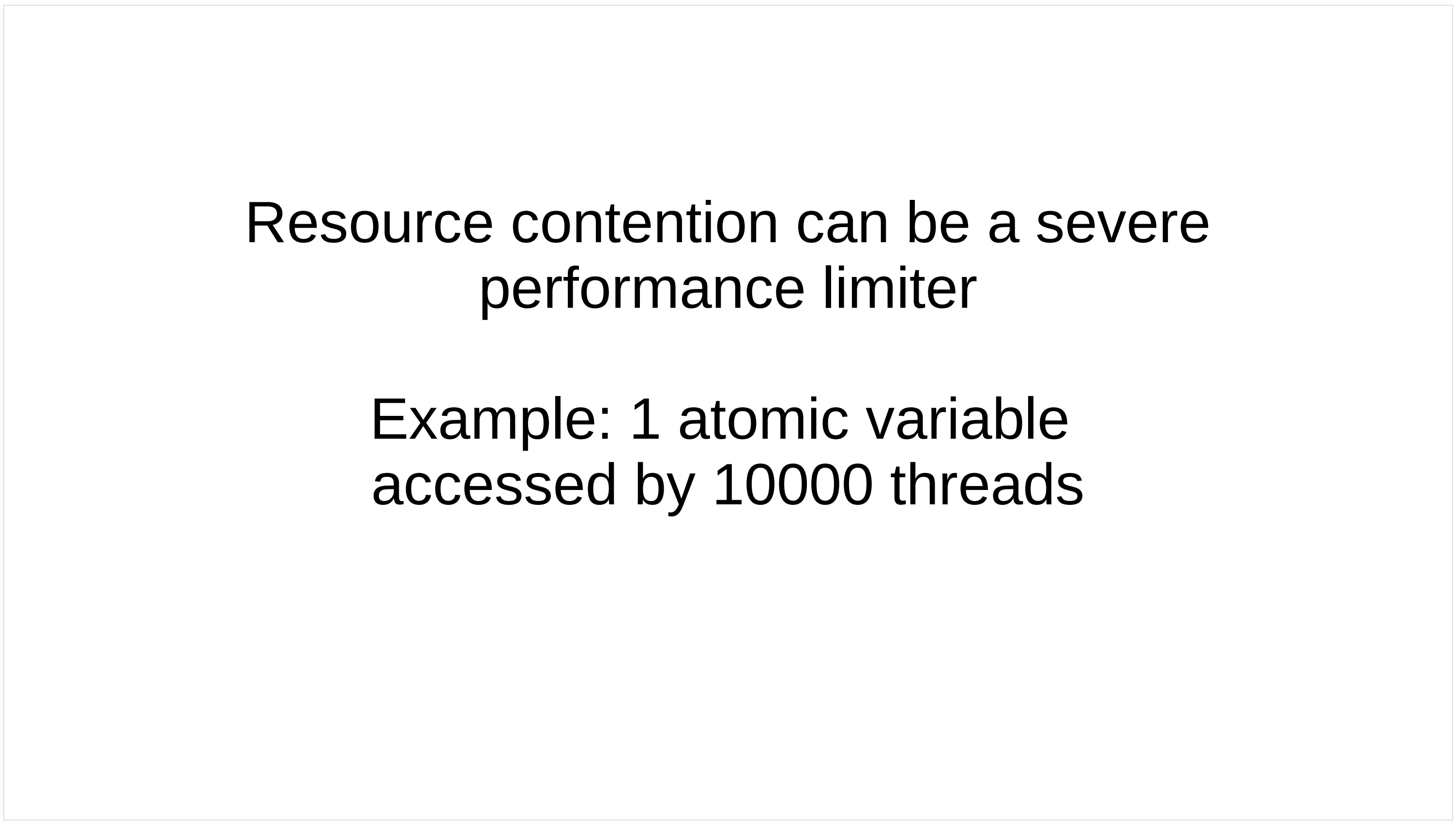
Copy on write data structures is a type of data structure that only creates a copy when its contents are modified. This is useful for sharing larger chunks of data between threads, as creating separate copies used in different threads may not always be necessary.

```
#include <iostream>
#include <thread>
#include <mutex>

thread_local int counter = 0;

void doCount(std::mutex* mut) {
    mut->lock();
    counter++;
    mut->unlock();
}

int main() {
    std::thread threads[10];
    std::mutex mut;

    for(int i = 0; i < 10; i++) {
        threads[i] = std::thread(doCount, &mut);
    }

    for(int i = 0; i < 10; i++) {
        threads[i].join();
    }

    std::cout << counter << std::endl;

    return 0;
}
```

$\longleftarrow$ thread_local global variables are allocated separately per thread

$\longleftarrow$ Program outputs 0

thread_local qualifiers create a copy of a global variable for each thread. You normally should not need to use this, but it can come in handy in specific situations.


Resource contention can be a severe performance limiter

Example: 1 atomic variable accessed by 10000 threads

Resource contention can be a big problem. 10000 threads trying to modify a single atomic variable are mostly going to wait for their turn, rather than doing useful work.

Reducing resource contention:

- Reduce duration locks are held

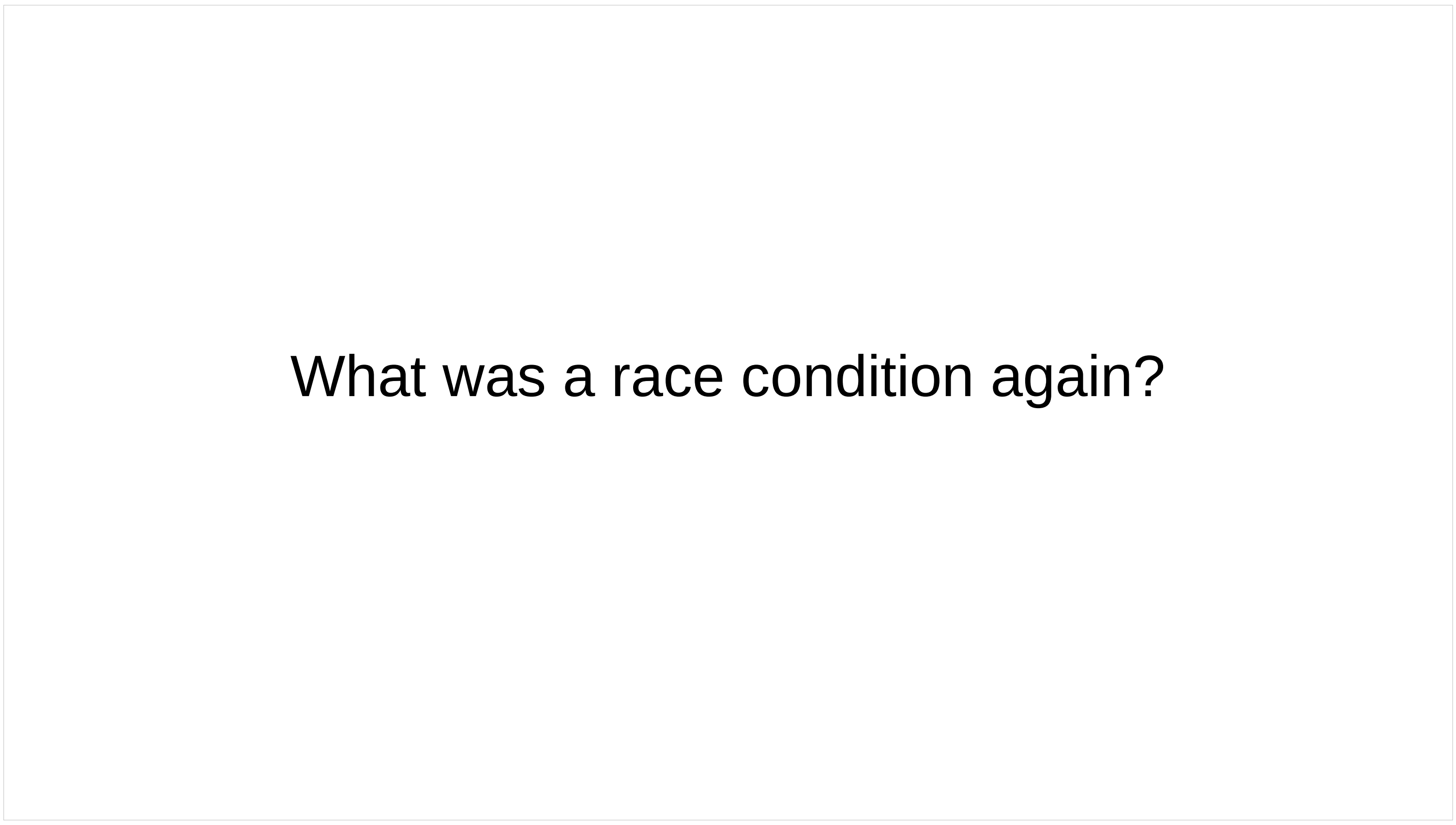
- Reduce frequency locks are requested

- Replace exlusive locks with coordination mechanisms which permit greater concurrency

Common solutions include trying to reduce the time each thread holds on to locks, reduce how often you request one (for example you can aggregate a sum within a thread, only incrementing the shared atomic variable at the end). The final method also adds more complexity to your program, and therefore makes race conditions harder to locate. This should be your last resort.

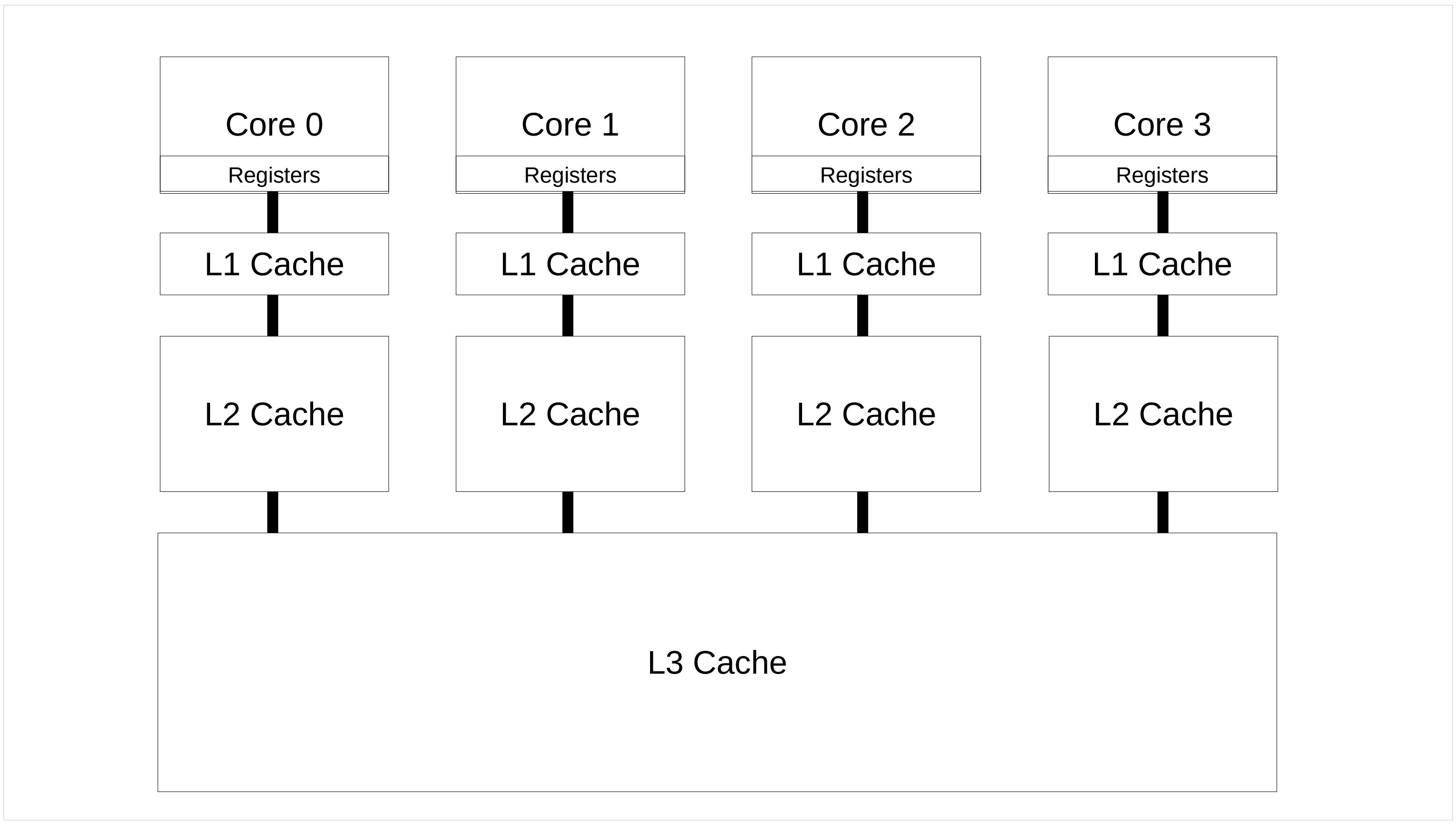
Recognising race conditions

For the exam, I’d like you to be able to recognise race conditions in a given piece of code.



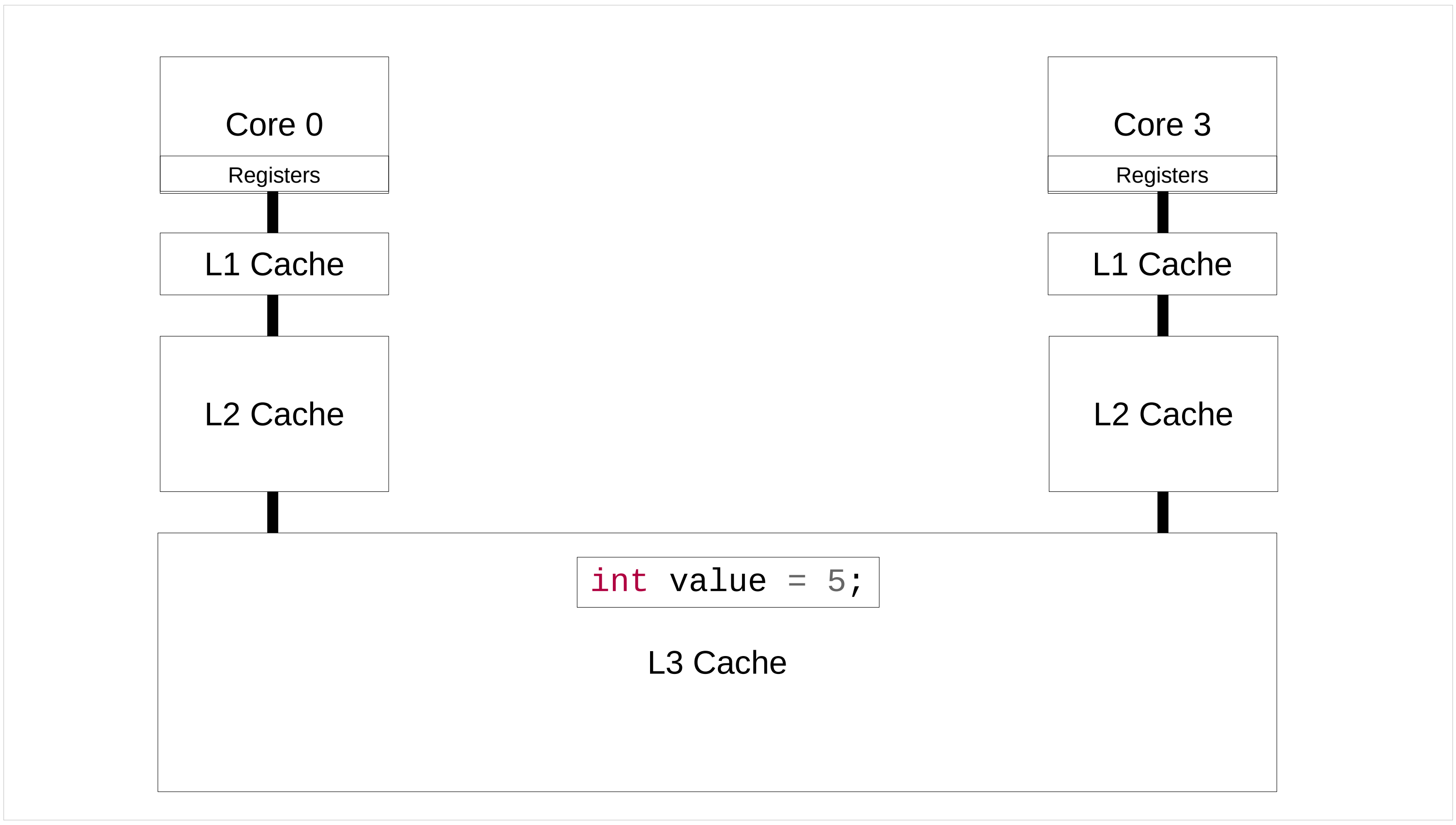
What was a race condition again?

Just as a refresher, what was a race condition again?

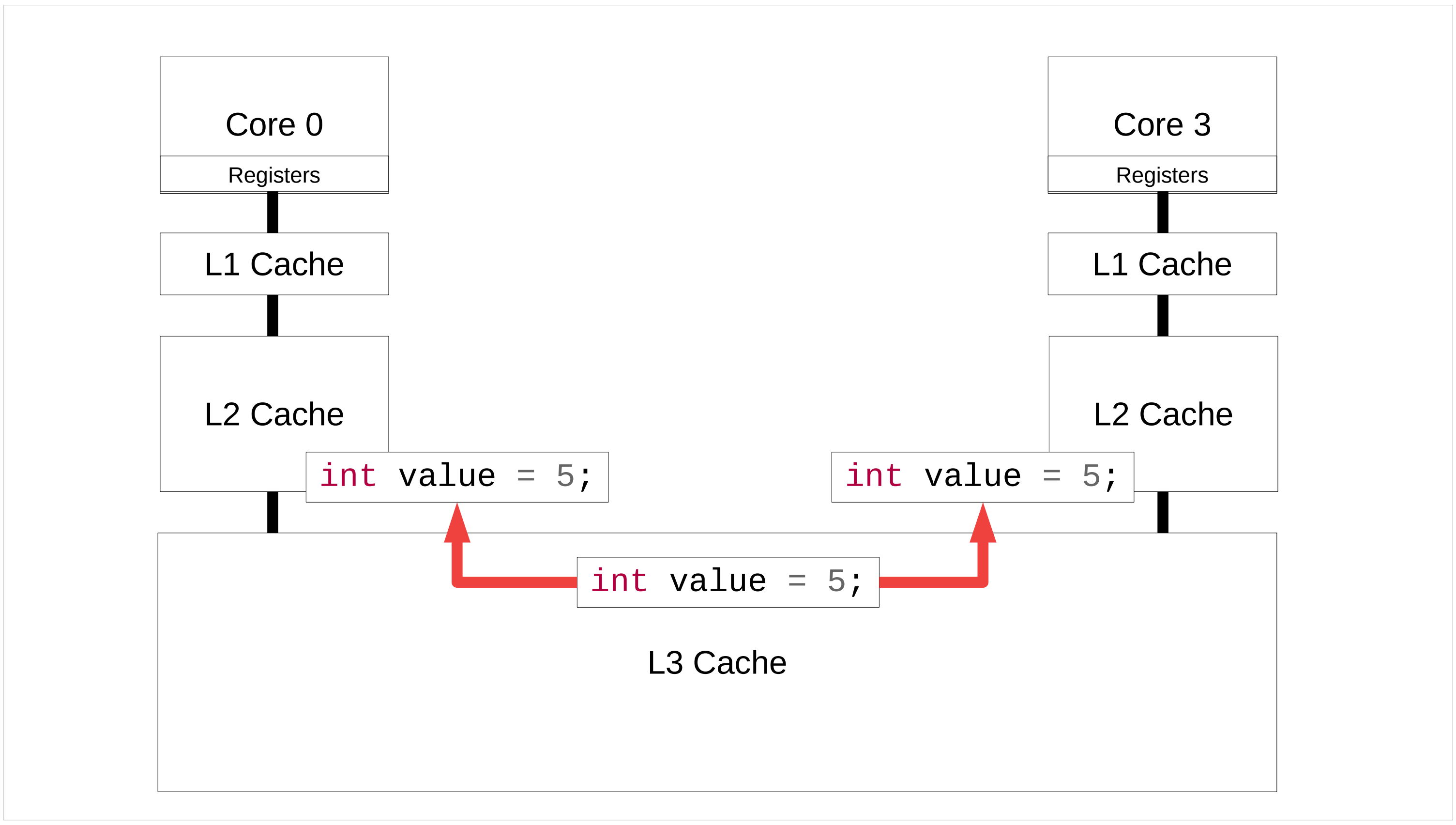


A race condition is caused by a factor outside the control of the code causes the outcome of a computation to be different.

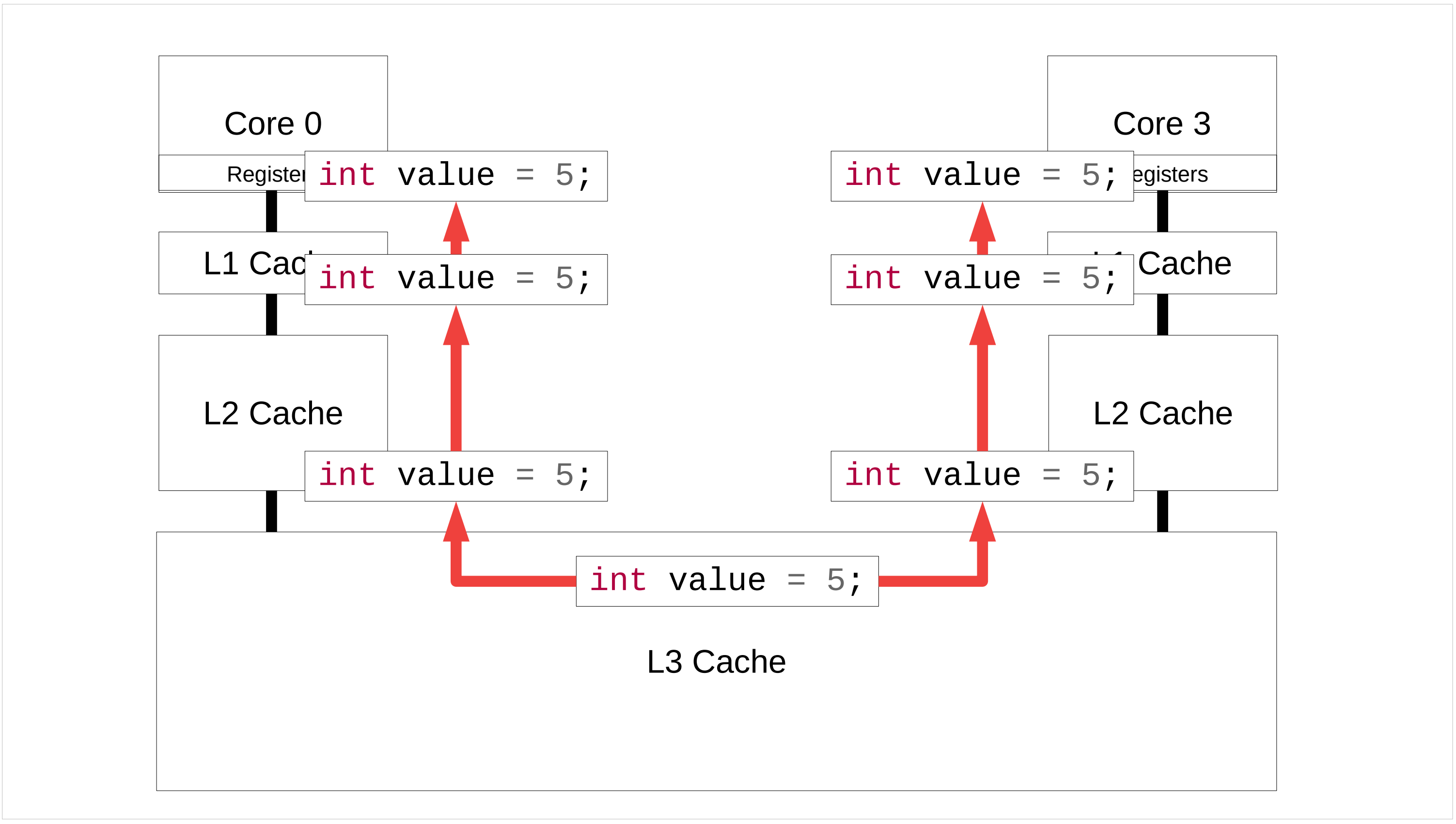
In the CPU, that usually means the cache, although there are other potential causes too.

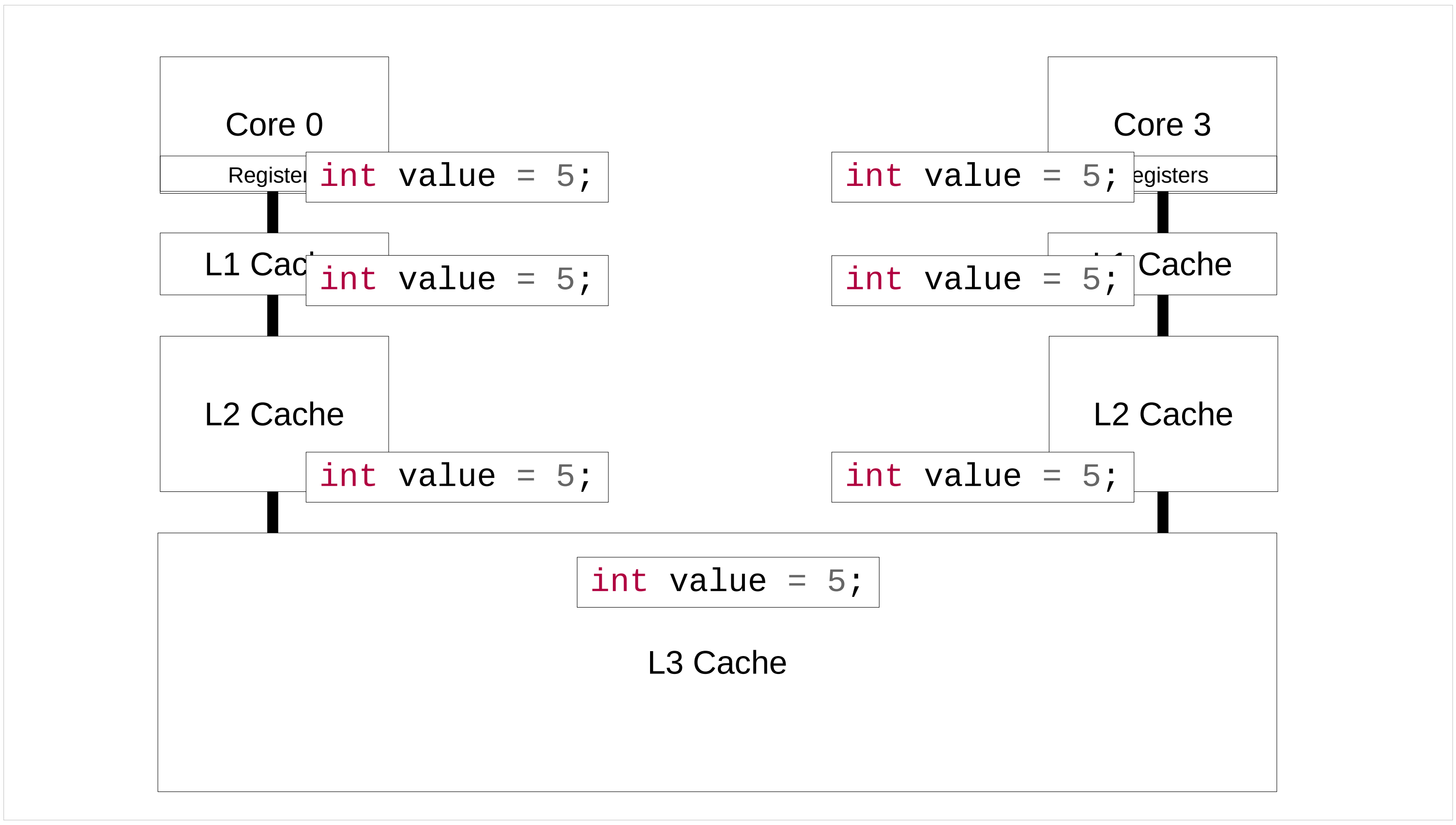


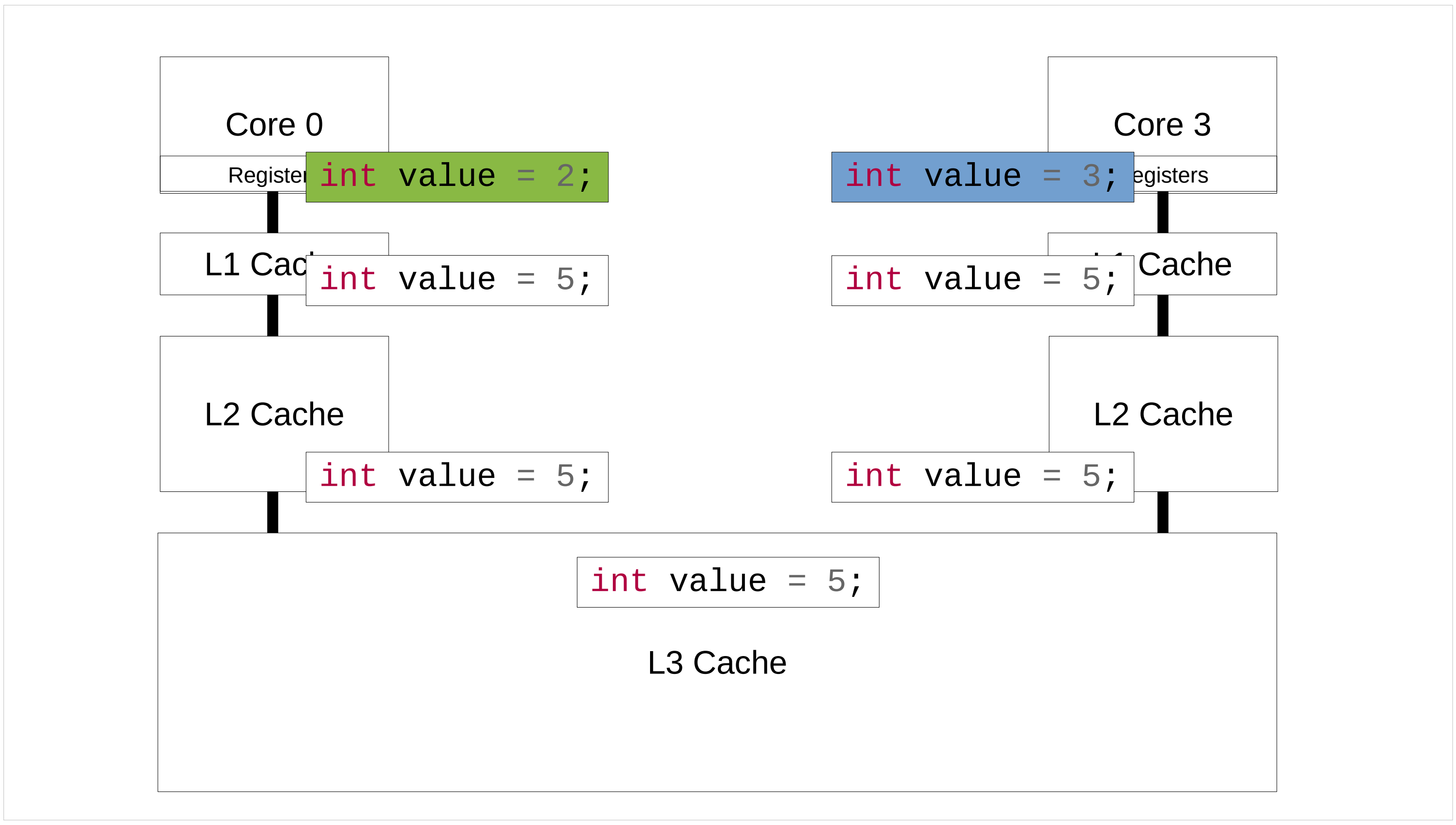
In this example, two threads will try to update a value that's currently in L3 cache.



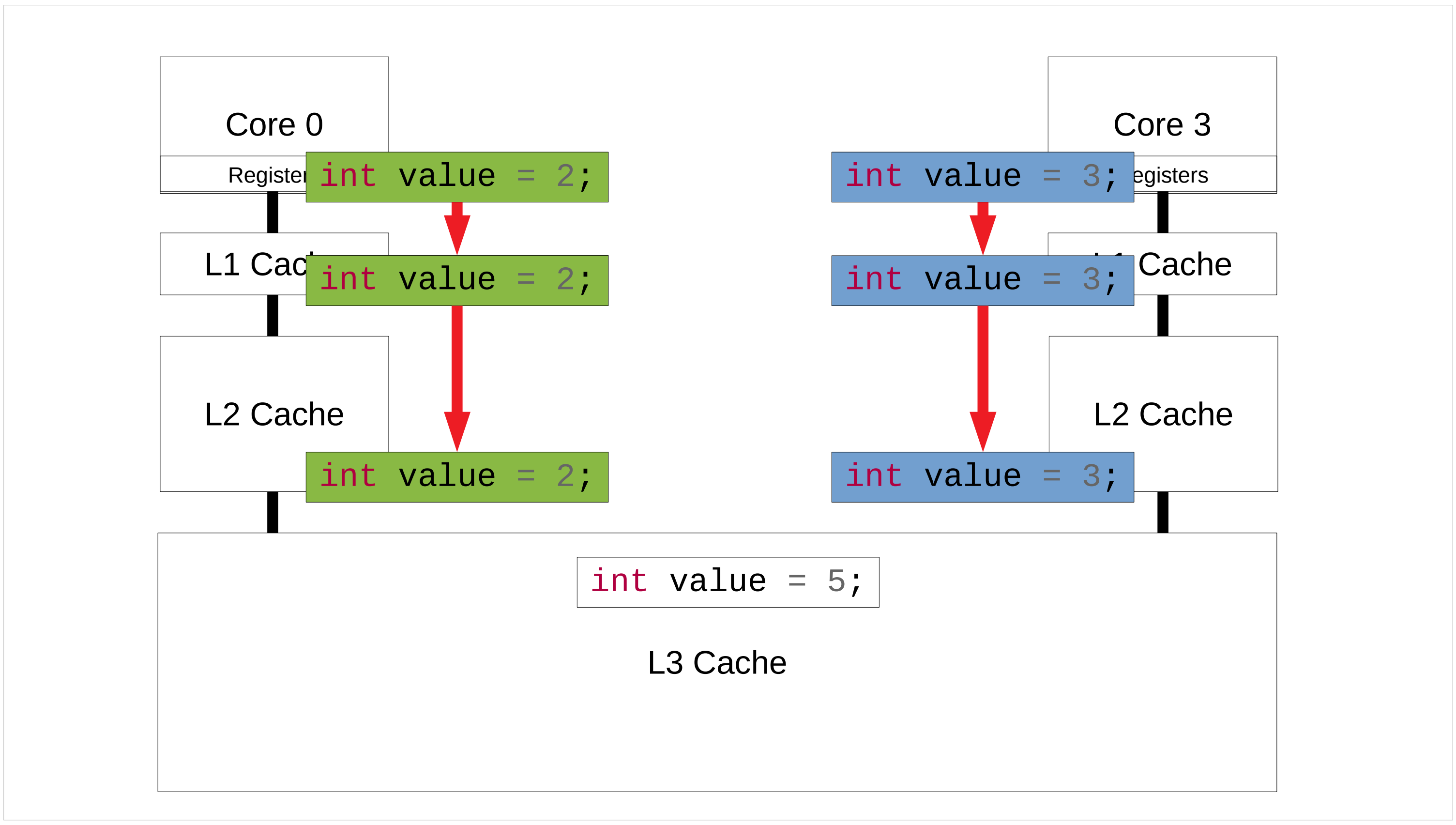
The CPUs request the value, which is then sent down the cache hierarchy into the core itself.


Core 0
Registers
L1 Cache
L2 Cache
Core 3
Registers
L1 Cache
L2 Cache
L3 Cache
int value = 5;

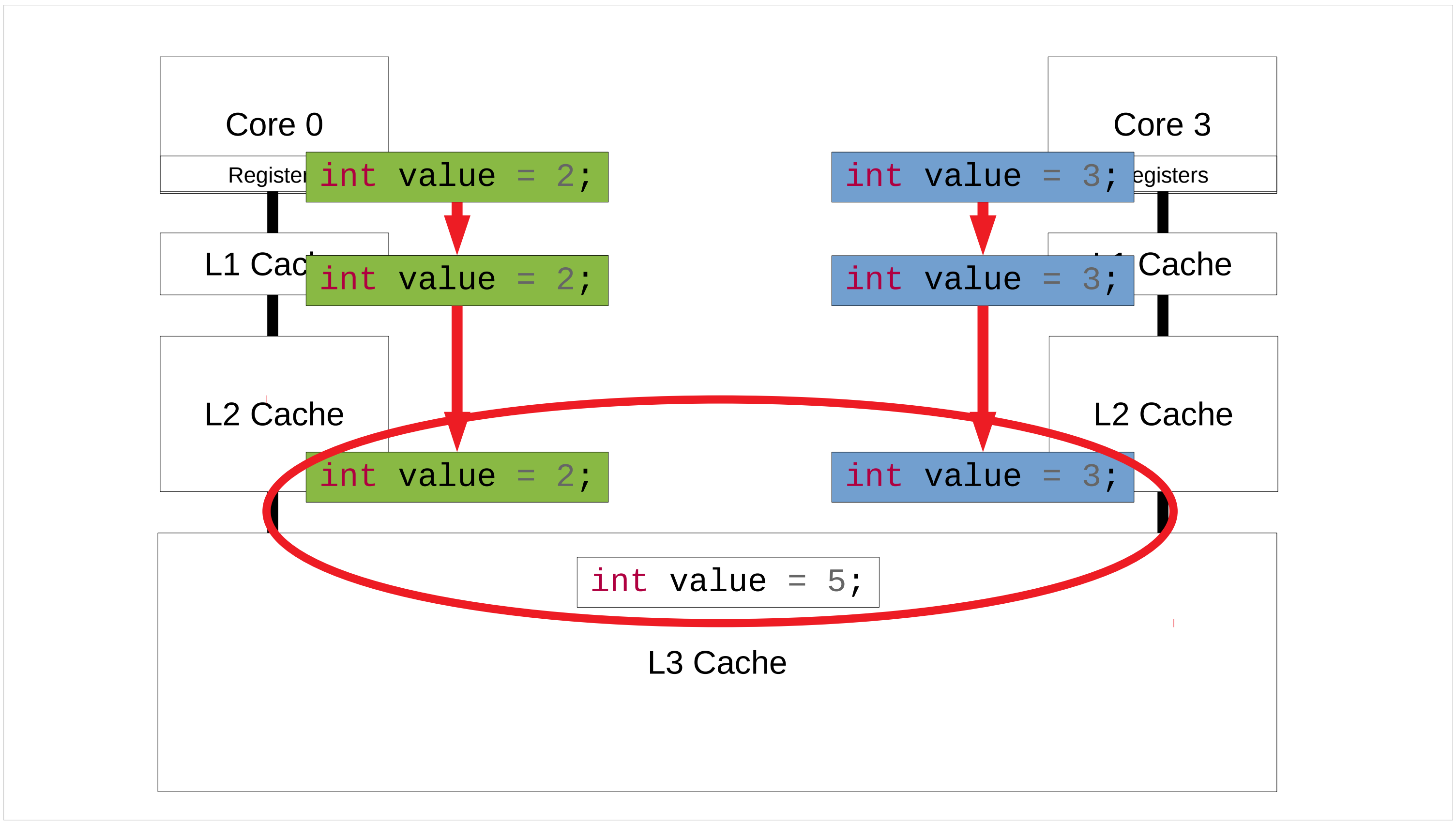

Core 0
Registers
int value = 5;
L1 Cache
int value = 5;
L2 Cache
int value = 5;
Core 3
int value = 5;
Registers
L1 Cache
int value = 5;
L2 Cache
int value = 5;
int value = 5;
L3 Cache



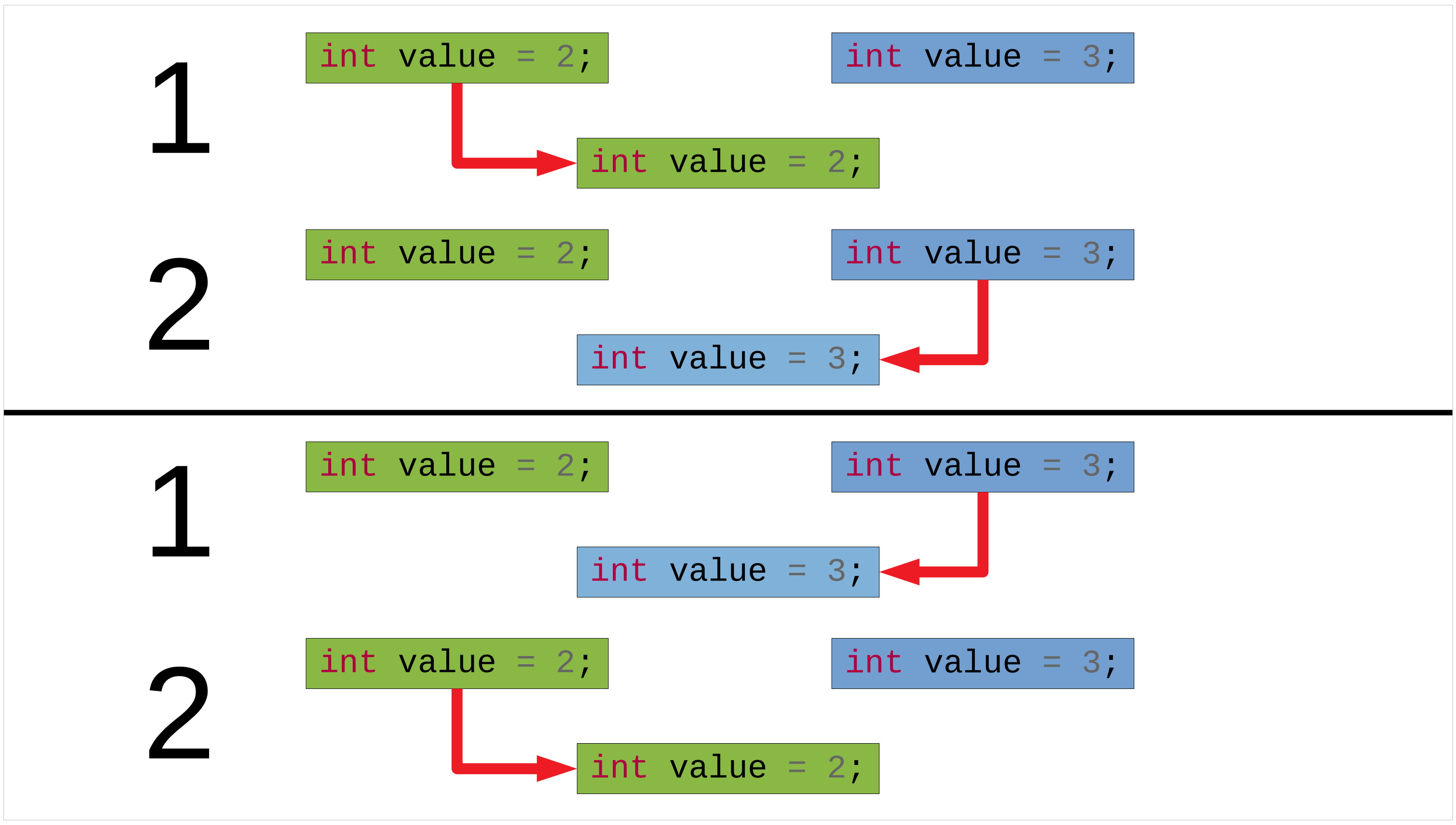
Here the program modifies its value, and both cores assign a different one.



Eventually, these values are written to higher cache levels.



This is where the race condition can occur. The values are going to be written to L3 cache from L2, but the order in which these writes occur affect the outcome.



There are two scenarios here. Both result in a different outcome. Hence this is a race condition.

Race conditions are not necessarily bad

Small note: race conditions are not always an issue, as long as they do not affect the outcome of the program.

Find the race condition!

```
#include <iostream>
#include <thread>
#include <atomic>

void playSpeedTicTacToe(char sign, std::atomic<char>* board) {
    for(int i = 0; i < 9; i++) {
        if(board[i] == ' ') {
            board[i] = sign;
        }
    }
}

int main() {
    std::atomic<char> board[9];
    for(int i = 0; i < 9; i++) { board[i] = ' '; }

    std::thread playerX(playSpeedTicTacToe, 'x', board);
    std::thread playerO(playSpeedTicTacToe, 'o', board);

    playerX.join(); playerO.join();

    std::cout << board[0] << "|" << board[1] << "|" << board[2] << std::endl;
    std::cout << board[3] << "|" << board[4] << "|" << board[5] << std::endl;
    std::cout << board[6] << "|" << board[7] << "|" << board[8] << std::endl;
    return 0;
}
```

If one thread performs the if(board[i] == ‘ ’) check and is interrupted temporarily, the other thread can have done the same check and written its own character to the empty board space. The other thread’s change will therefore be overwritten.

The board itself consists of atomic integers, so any change made by one of the threads *will* be seen by the other one. However, this does not prohibit the race condition from happening.

```cpp
#include <iostream>
#include <thread>
#include <atomic>

void playSpeedTicTacToe(char sign, std::atomic<char>* board) {
    for(int i = 0; i < 9; i++) {
        if(board[i] == ' ') {
            board[i] = sign;
        }
    }
}

int main() {
    std::atomic<char> board[9];
    for(int i = 0; i < 9; i++) { board[i] = ' '; }

    std::thread playerX(playSpeedTicTacToe, 'x', board);
    std::thread playerO(playSpeedTicTacToe, 'o', board);

    playerX.join(); playerO.join();

    std::cout << board[0] << "|" << board[1] << "|" << board[2] << std::endl;
    std::cout << board[3] << "|" << board[4] << "|" << board[5] << std::endl;
    std::cout << board[6] << "|" << board[7] << "|" << board[8] << std::endl;
    return 0;
}
```

```
x|x|x
x|x|x
x|x|x
```

In practice, the second thread takes so long to initialise that the x player consistently appears to win. Does not take away that once in a blue moon the race condition can occur, however! Unlikely race conditions are bound to happen at some point.

```
#include <iostream>
#include <thread>
#include <atomic>
#include <stdlib.h>
#include <time.h>

void playSpeedTicTacToe(char sign, std::atomic<char>* board) {
    for(int i = 0; i < 10000; i++) {
        int randomField = rand() % 9;
        board[randomField] = sign;
    }
}

int main() {
    srand (time(NULL));

    std::atomic<char> board[9];
    for(int i = 0; i < 9; i++) { board[i] = ' '; }

    std::thread playerX(playSpeedTicTacToe, 'x', board);
    std::thread playerO(playSpeedTicTacToe, 'o', board);

    playerX.join(); playerO.join();

    return 0;
}
```

This snippet shows the risk of using library functions inside multiple threads. If these functions are not guaranteed to be thread safe (and the vast majority you'll find is not), you risk running into race conditions inside of them.

In this case, the rand() function depends on internal state to compute the next random number. Having two threads present inside the function can therefore lead to race conditions.

```
#include <iostream>
#include <thread>
#include <mutex>

void doCount(std::mutex* mut, int* counter) {
    mut->lock();
    (*counter)++;
    mut→unlock();

    std::cout << "Counter is now: " << counter << std::endl;
}

int main() {
    std::thread threads[10];
    std::mutex mut;

    for(int i = 0; i < 10; i++) {
        threads[i] = std::thread(doCount, i, &mut, &counter);
    }

    for(int i = 0; i < 10; i++) {
        threads[i].join();
    }

    return 0;
}
```

Another example of using library functions. The counter variable is properly locked here, but std::cout is not thread safe.

```
#include <iostream>
#include <thread>
#include <condition_variable>

void wakeSleep(std::condition_variable* sleeper, std::mutex* mut) {
    std::unique_lock<std::mutex> lock(*mut);
    sleeper->notify_all();
    sleeper->wait(lock);
    sleeper->notify_all();
}

int main() {
    std::thread threads[10];
    std::mutex mut;
    std::condition_variable sleeper;

    for(int i = 0; i < 10; i++) {
        threads[i] = std::thread(wakeSleep, i, &sleeper, &mut);
    }
    for(int i = 0; i < 10; i++) {
        threads[i].join();
    }

    return 0;
}
```

This snippet shows a race condition involving a condition variable. In one scenario, all threads first call notify_all(), releasing any waiting threads. However, there are none at that point in time, so nothing happens. Next, all threads wait on the wait() call, resulting in a deadlock.

Final two are on a handout sheet!

If you have a computer:

https://gist.github.com/bartvbl

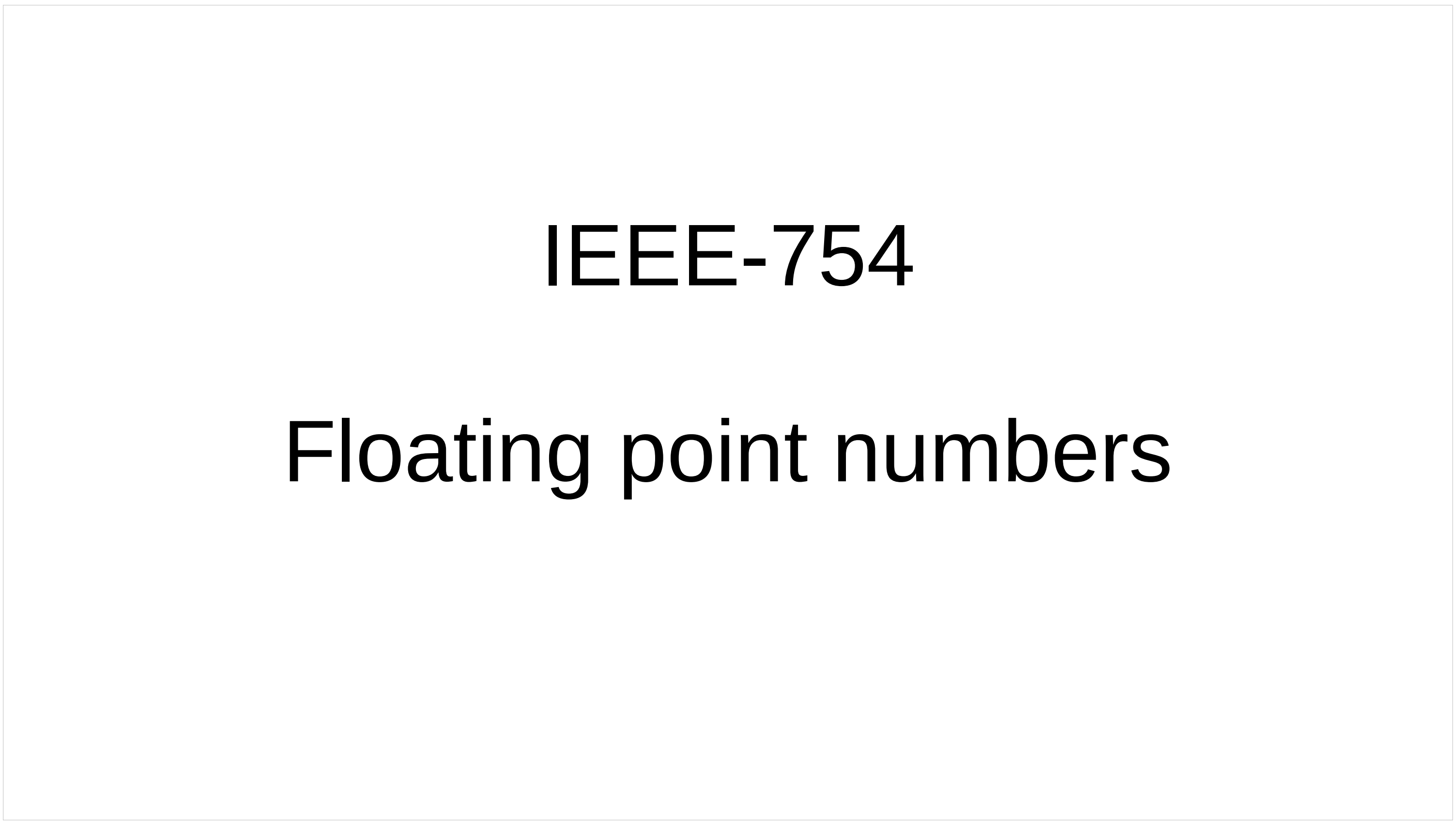
Front side: there are three lines which perform the actual juggling (call the different functions in Hand). Putting two mutexes around it is easiest, but it's worth noting that the left hand is only used in the first two lines, and the right hand only in the last two. So you can unlock the left hand a little earlier, and lock the right hand a little later.

The back side: I mentioned during the lecture that there is one race condition in the else clause of the if statement that handles the behaviour of each executing thread, between incrementing readyCount and trying to lock the first present. This is not the case, as the outcome of the program is not affected by whether the mutex was locked or unlocked by thread 0 at that point. However, the program will always deadlock, because threadCount does not equal the number of presents, which the code relies upon (evil, I know).

And now for something completely different..

IEEE-754

I wanted to talk a bit about floating point numbers. It was kind of rushed during the lecture, but there are a few things that are relevant to keep in mind when doing processing of large sets of data.



IEEE-754

Floating point numbers

First thing is accuracy. Floating point numbers cannot represent every real number out there, so they need to round to the nearest one they can represent.

Their number representation works on the basis of “buckets”. Each bucket covers the number between a power of 2. For instance, a few successive buckets would be 2-4, 4-8, 8-16, 16-32, and so on. Negative powers are also allowed, which covers numbers closer to zero (1/8 – ¼, ¼ – ½, ½ – 1). The bucket in which the number is located is stored in the so-called “exponent” bits of the floating point number. Within each bucket, numbers are interpolated linearly between each power of two. This value is stored in the “mantissa” bits of the number.

Subsequent floats:

$1024.0 \rightarrow 1024.0001$

$65536.0 \rightarrow 65536.01$

$1048576.0 \rightarrow 1048576.1$

The greater the powers of two are, the greater distance each successive mantissa value covers. Here are some examples of successive floating point values. Notice in the case of bottom one that only steps of 0.1 are representable by the IEEE-754 standard.

Each arithmetic operation increases the error of the final result.

Bigger ‘scale’ differences cause bigger errors.

Since not all real numbers can be represented, each arithmetic operation results in a rounding error between the “correct” answer and the value that ends up being stored in the floating point number.

The longer your “chain” of successive floating point computations is, the greater this error will be. For instance, summing an array of floats by iterating over the elements and adding it to an accumulator variable means your error will increase linearly with the number of values in the array. It may be better to split up the array into multiple parts, and computing a sum on each separately first if you run into accuracy problems.

Also worth noting is that adding big numbers to small ones is the best way of introducing accuracy errors. Performing arithmetic operations in a different order may be beneficial there.

Floating point arithmetic is not associative.
Order definitely matters!

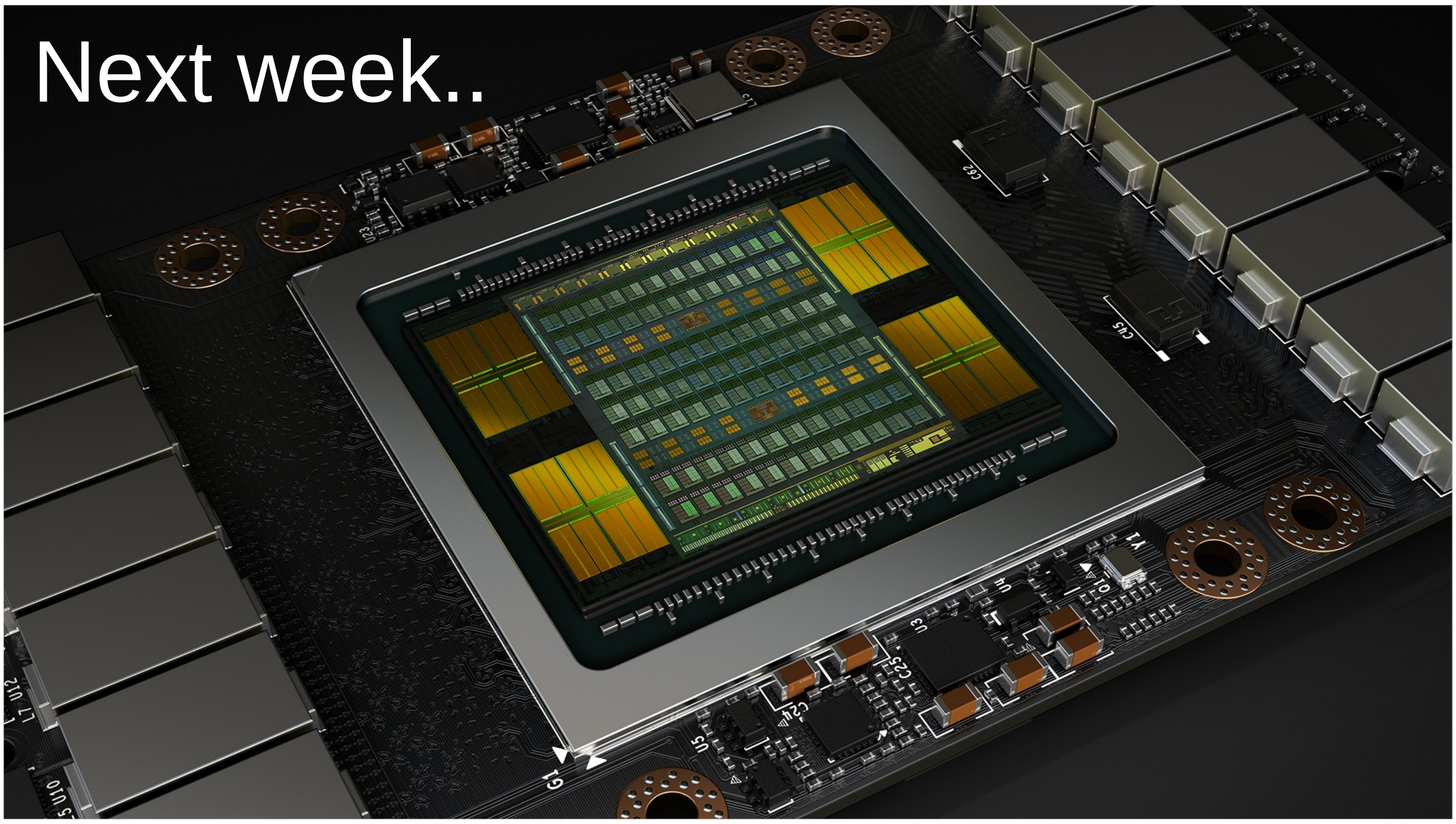
Half of their precision is between -1 and 1.

A couple of other things that may be relevant:

Due to the rounding behaviour, calculations are not associative (I said commutative in the lecture. Oops!).

Also, half of the values floating point numbers can represent are between -1 and 1. You may be able to exploit that if you need the precision.

Next week..



Next week we’ll talk about OpenMP, and FINALLY touch on GPUs (looking forward to that myself :))