It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages.
Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages.

And to sequential-trained programmers.
Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages.

And to sequential-trained programmers.

There is nothing in the C language itself to stop parallel stupidities as it was designed as a sequential language.
Concurrency Control
Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages.

And to sequential-trained programmers.

There is nothing in the C language itself to stop parallel stupidities as it was designed as a sequential language.

As were many other languages in popular use today.
Modern compilers and modern hardware both try their best to execute your code as fast as possible.
Modern compilers and modern hardware both try their best to execute your code as fast as possible

But in doing so, they can break parallel code
Modern compilers and modern hardware both try their best to execute your code as fast as possible.

But in doing so, they can break parallel code.

For example, some compiler optimisations can break parallel code.
Modern compilers and modern hardware both try their best to execute your code as fast as possible.

But in doing so, they can break parallel code.

For example, some compiler optimisations can break parallel code.

And some hardware optimisations can break parallel code.
Concurrency Control
Compiler Reordering

Modern compilers sometimes reorder code to make things more efficient. For example in

A
while (cont == 0) {/* nothing */} x = 42;
print x;

B
cont = 1;

where the intent was to have thread A to wait for thread B to set the cont flag before continuing to print 42
Modern compilers sometimes reorder code to make things more efficient. For example in

```plaintext
A
while (cont == 0) {/* nothing */} print x;
B
x = 42;
cont = 1;
```

where the intent was to have thread A to wait for thread B to set the `cont` flag before continuing to print 42.

A compiler only seeing the code for B may conclude that the variables `cont` and `x` are independent and so (perhaps for some efficiency reason) it can rearrange the code as

```plaintext
cont = 1;
x = 42;
```
Similarly for A: it is possible that the print can be reordered to before the loop
Similarly for A: it is possible that the print can be reordered to before the loop

Note: *never* write code like this in the hope that it might work: it is simply buggy code! Use a semaphore or equivalent
Similarly for A: it is possible that the print can be reordered to before the loop

Note: *never* write code like this in the hope that it might work: it is simply buggy code! Use a semaphore or equivalent

The problem is that there is a hidden relationship between the variables \( x \) and \( \text{cont} \) that is in the mind of the programmer, but is not expressed in the code
Example. Consider the code:

```c
int a = 0;
int b = 0;

A
a = 42;
printf("%d\n", b);

B
b = 42;
printf("%d\n", a);
```

Explain how it might print 0 twice, even though it appears we always print after an update.
Concurrency Control

Compiler Reordering

Thus, to be correct, the programmer needs to inform the compiler not to do these kinds of “optimisations”
Concurrency Control

Compiler Reordering

Thus, to be correct, the programmer needs to inform the compiler not to do these kinds of “optimisations”

Languages like C and Java have a `volatile` keyword

```c
volatile int cont;
```

tells the compiler not to mess around with such variables and assume that external operations might change their value
Concurrency Control
Compiler Reordering

Thus, to be correct, the programmer needs to inform the compiler not to do these kinds of “optimisations”

Languages like C and Java have a volatile keyword

volatile int cont;

tells the compiler not to mess around with such variables and assume that external operations might change their value

But volatile was introduced for hardware/peripheral-related reasons and is not recommended as a way of fixing concurrency issues as they don’t solve the whole problem, as the hardware needs telling, too
Concurrency Control

Compiler Reordering

Thus, to be correct, the programmer needs to inform the compiler not to do these kinds of “optimisations”

Languages like C and Java have a volatile keyword

volatile int cont;

tells the compiler not to mess around with such variables and assume that external operations might change their value

But volatile was introduced for hardware/peripheral-related reasons and is not recommended as a way of fixing concurrency issues as they don’t solve the whole problem, as the hardware needs telling, too

Summary: don’t use volatile to try to solve parallelism problems
We have already seen (a while ago) how modern CPUs use out of order execution on machine instructions to improve efficiency in superscalar architectures, where the processor can reorder instructions as it sees fit.
We have already seen (a while ago) how modern CPUs use *out of order execution* on machine instructions to improve efficiency in superscalar architectures, where the *processor* can reorder instructions as it sees fit.

So, even given un-reordered code (or machine code equivalent loading registers):

```plaintext
cont = 1;  load $r1, 1
x = 42;    load $r2, 42
```

the CPU might *while running* decide the loads are independent and load *x* ($r2) first.
Concurrency Control

Hardware Reordering

We have already seen (a while ago) how modern CPUs use *out of order execution* on machine instructions to improve efficiency in superscalar architectures, where the *processor* can reorder instructions as it sees fit.

So, even given un-reordered code (or machine code equivalent loading registers):

```plaintext
cont = 1;
load $r1, 1
x = 42;
load $r2, 42
```

the CPU might *while running* decide the loads are independent and load $x$ ($r2$) first.

Out of order execution is common in modern architectures.
Thus we also need special machine instructions like

```c
while (cont == 0) { /* nothing */ }  
ex = 42;  
memory_fence();  
memory_fence();  
print x;  
cont = 1;
```

(details vary according to language and compiler) that tell the processor and compiler not to reorder things.
Thus we also need special machine instructions like

```java
while (cont == 0) {/* nothing */} x = 42;
memory_fence(); memory_fence();
print x;
cont = 1;
```

(details vary according to language and compiler) that tell the processor and compiler not to reorder things

Then `memory_fence()` would compile to a specific special machine instruction that tells the out of order mechanism not to move things across this boundary.
In fact, you *must* use some primitive like a fence, or something that uses a fence (e.g., a semaphore), to ensure the intended behaviour.
Concurrency Control
Memory Consistency

In fact, you *must* use some primitive like a fence, or something that uses a fence (e.g., a semaphore), to ensure the intended behaviour.

Memory fences work, but are a synchronisation point (so, bad by Amdahl, etc.), thus more subtle mechanisms are also used.
The specification for a parallel language needs a *memory model* to describe how memory reads and writes are visible to multiple processors.
Concurrency Control
Memory Consistency

The specification for a parallel language needs a *memory model* to describe how memory reads and writes are visible to multiple processors.

This involves the use of special language constructs and special to inform the compiler and hardware about what kinds of reordering are allowable and what kinds of *memory consistency* across processors are needed.
The specification for a parallel language needs a memory model to describe how memory reads and writes are visible to multiple processors.

This involves the use of special language constructs and special to inform the compiler and hardware about what kinds of reordering are allowable and what kinds of memory consistency across processors are needed.

So, for example, the programmer may decide that some reads or some writes may be reordered, while others should not.
Concurrenty Control
Memory Consistency

Generally, the programmer must understand the issues involved and use the right constructs in the right places.
Concurrency Control
Memory Consistency

Generally, the programmer must understand the issues involved and use the right constructs in the right places

Allowing just enough flexibility for the compiler/hardware to be efficient, while still correct code
Concurrency Control
Memory Consistency

Generally, the programmer must understand the issues involved and use the right constructs in the right places.

Allowing just enough flexibility for the compiler/hardware to be efficient, while still correct code.

Thus allowing the system to reduce synchronisation and increase parallelism.
Concurrency Control

Memory Consistency

Fortunately for of us, if we use primitives (locks, semaphores, and so on) and higher-level constructs they will look after the details for us
Concurrency Control
Memory Consistency

Fortunately for of us, if we use primitives (locks, semaphores, and so on) and higher-level constructs they will look after the details for us

As long as we use them!
Fortunately for us, if we use primitives (locks, semaphores, and so on) and higher-level constructs, they will look after the details for us.

As long as we use them!

So: if you have a cross-thread relationship, use a parallelism mechanism, don’t just wing it.
Concurrency Control
Memory Consistency

Fortunately for us, if we use primitives (locks, semaphores, and so on) and higher-level constructs they will look after the details for us.

As long as we use them!

So: if you have a cross-thread relationship, use a parallelism mechanism, don’t just wing it.

From the Go website:

Share memory by communicating; don’t communicate by sharing memory.
Concurrency Control
Memory Consistency

Exercise. Read about memory consistency. Including: memory fences, *strict consistency*, *strong consistency*, *causal consistency*, *weak consistency*, *sequentially consistent*, *acquire-release*, *relaxed*, *consume*, etc.

Exercise. Read about the difference between Java’s memory model and C/C++’s model (and what volatile does in each)

Exercise. Read about the difference between the Intel (x86) memory model and the Arm memory model