
Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
Are There Time Bombs in Your Real-Time Code?
Mike Lindner
More – in my bilingual blog: https://sw-arch.blog/
Architect in Slippers / ארכיטקט בכפכפים

Intro

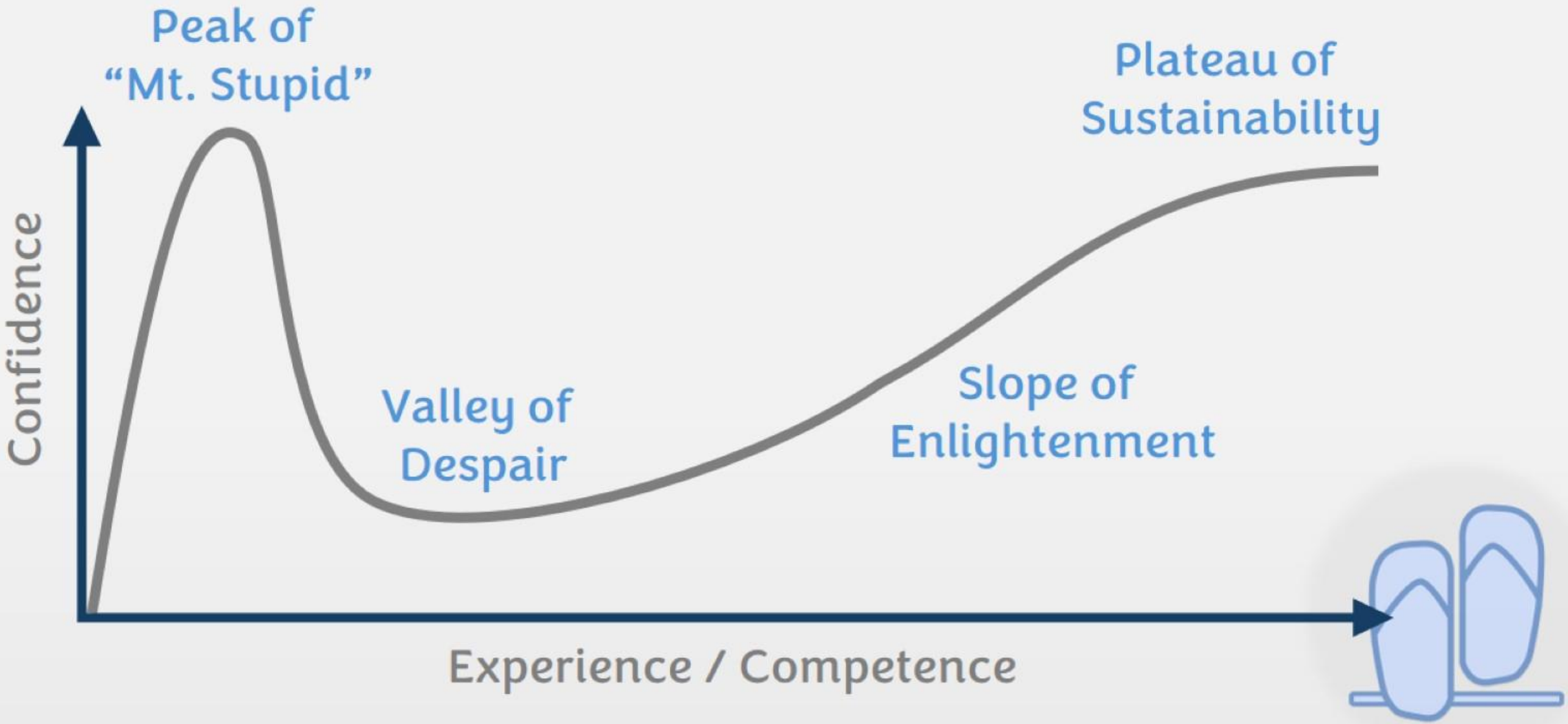
Intro

- Who I am
- Who's the audience
- What this lecture is about

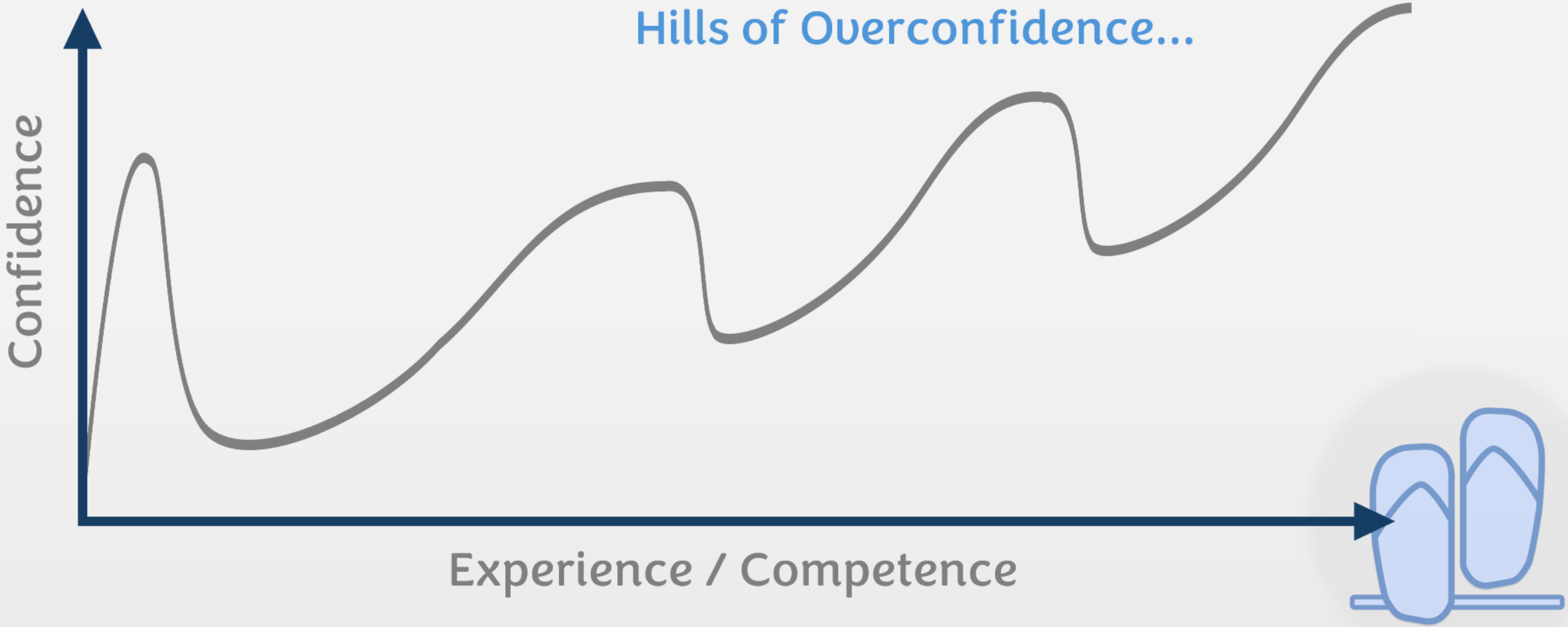
Rambam

Hear
the truth
from
whoever
utters it

Dunning Krugger Effect



Maybe It's a Fractal?



C, C++, RT/Embedded Programmers

We have a much better grasp of the internals:

- Bits and bytes.
- Compilers and their shenanigans.
- Processors and their architectures.

But those are also very challenging domains:

- Highly complicated.
- Rapidly changing.
- Dramatically different on each platform.

So, we must pay attention not to fall off those hills.

Socrates

I know
one thing –
That
I know
nothing

“Severity” of Issues

Found in Dev

- Break compilation
- Break linkage
- Immediate crash
- Wrong logic/math

Found Too Late

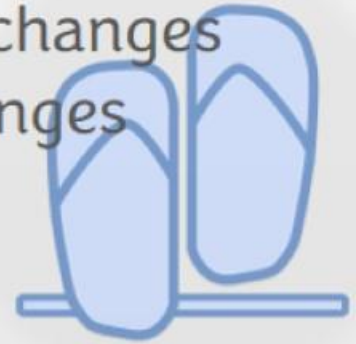
- Found in prod
- *Not* found in prod

Found in Lab

- Occur sometimes
- Occur rarely

Waiting in the Shadows...

- Exist only in specific compilations
- Might occur if (other) code changes
- Might occur if platform changes



A Bug Found in Production

Like the possibility to create 184,467,440,737 BTC in 2010

A Bug that is NOT Found in Production

Like the one
that made
TWO Boeing-737
crashes
in 2018–2019

The Airworthiness Group comprised of members from Ethiopian CAA, Boeing, and NTSB convened at accident site, located near Ejere, Ethiopia, on March 12, 2019 to examine the Airplane wreckage with a specific focus on flight controls and the air data system components.

FIGURE 1: RECOVERED WRECKAGE PILE ON THE SITE

1.12.5 Recovered Wreckage Examination

A Piece of Code that Perfectly Works

Better Not to Appear in that List...

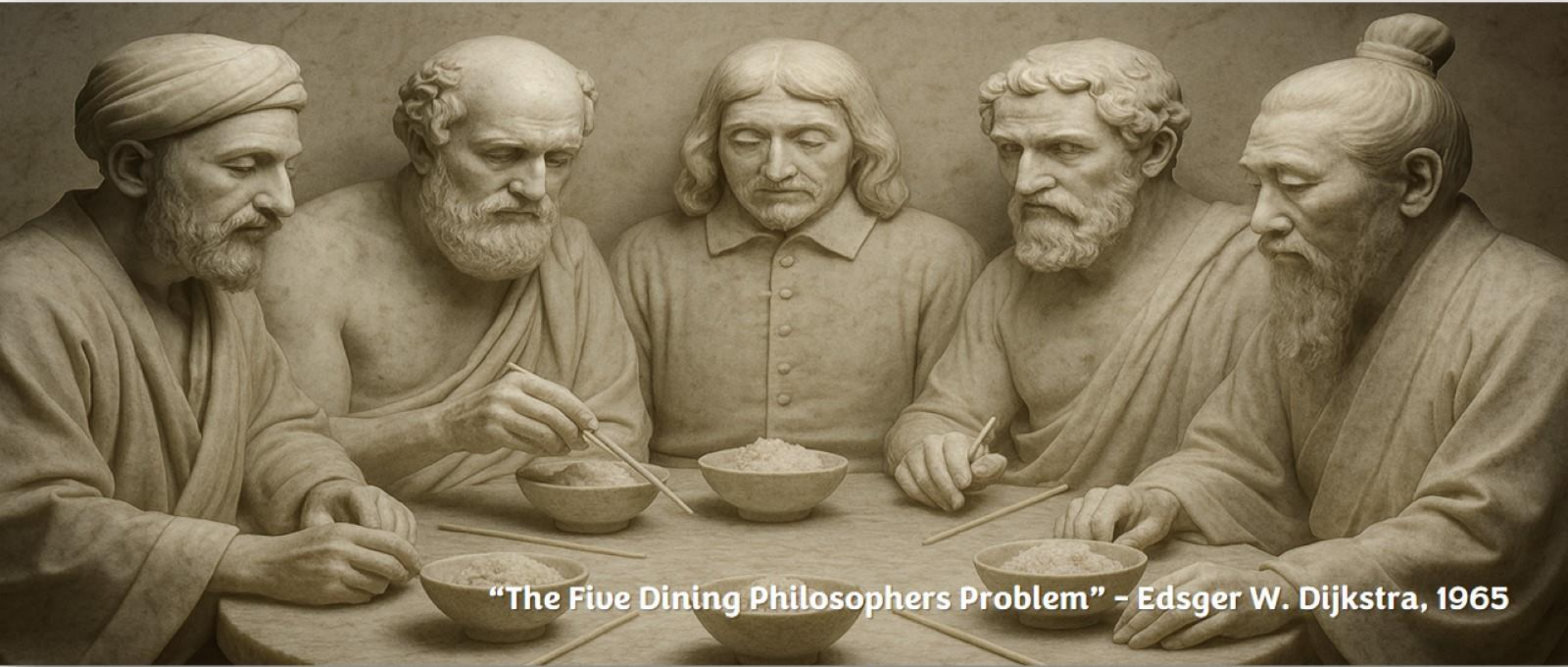
So, let's focus on a specific issue.

One that makes RT programmers give their best.

Race Condition

And creative ways to avoid it.

The Problem is Known.



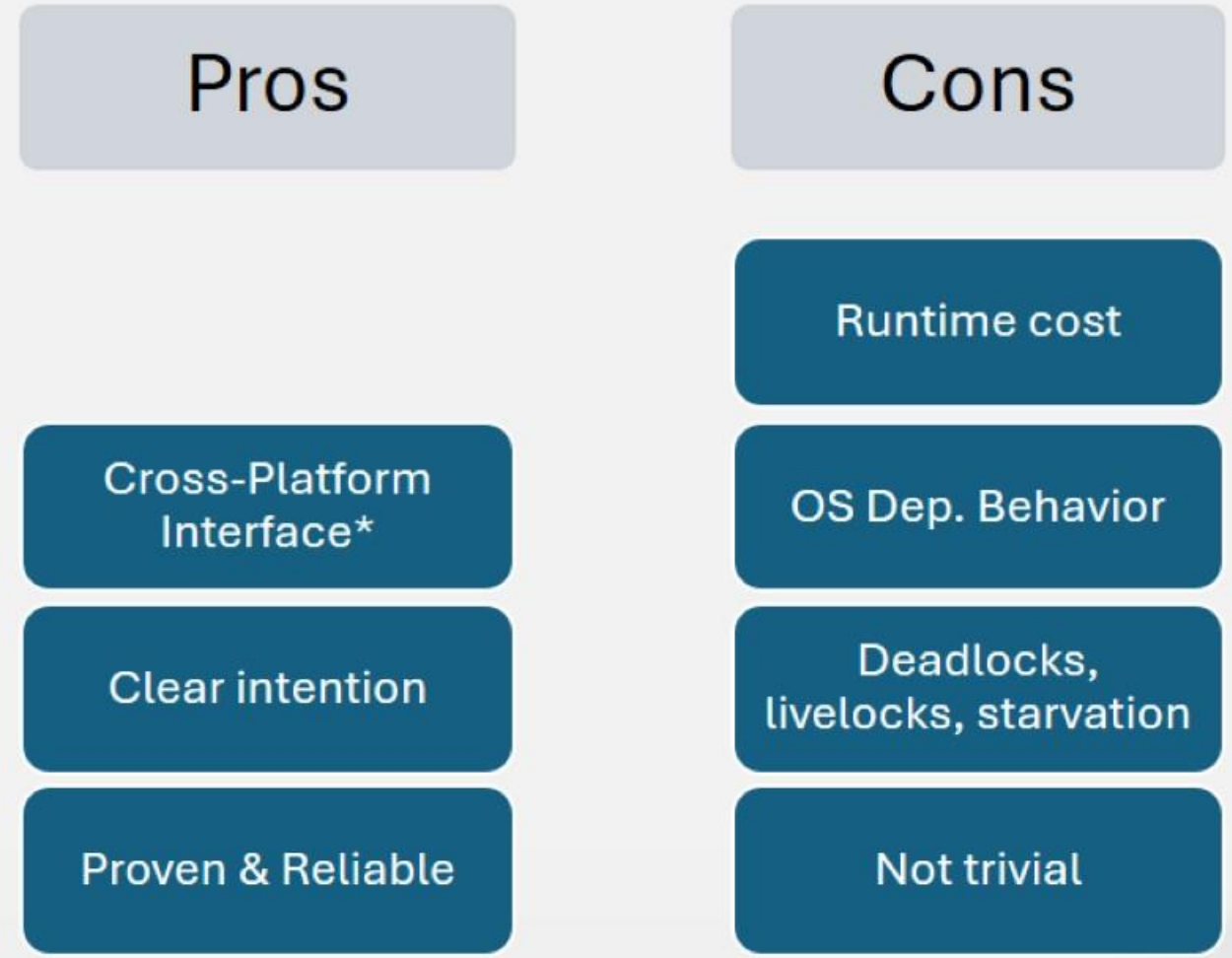
So are its Common Solutions:

- Mutex
- Semaphore
- ...

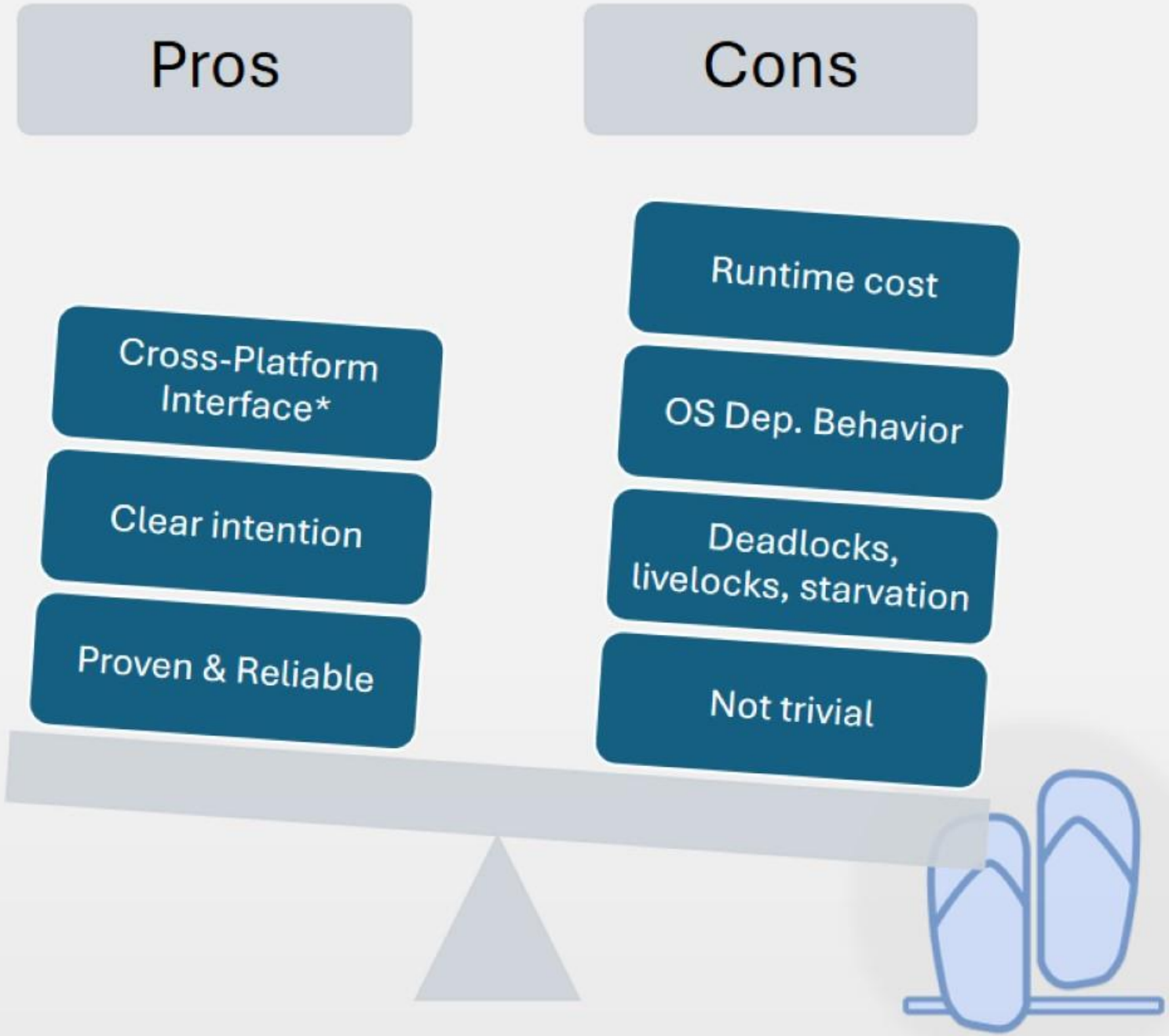
Classic Sync Objects

Pros	Cons
Cross-Platform Interface*	OS Dep. Behavior
Clear intention	Deadlocks, livelocks, starvation
Proven & Reliable	Not trivial

Classic Sync Objects



Classic Sync Objects



Which is why...

Many RT/Embedded programmers try to avoid them.

For a good reason.

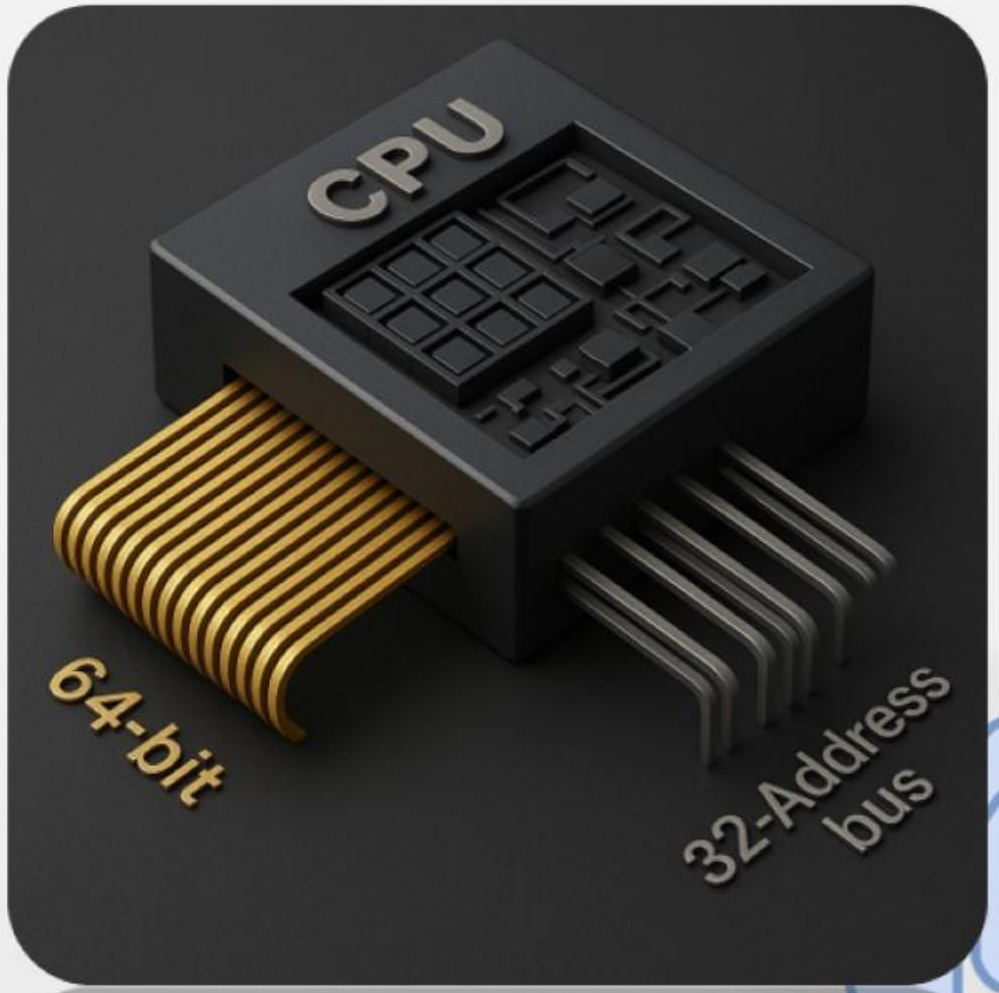
With numerous other creative solutions.


CPU I/O Size “Atomicity”
Rely on the CPU ability to store/load in a single instruction

Atomicity of a Processor I/O Size

We know the chip architecture.

Why not use it?



It works like this:

CPU I/O width is known ✓

Use a variable that long ✓

CPU will access it in a single instruction ✓

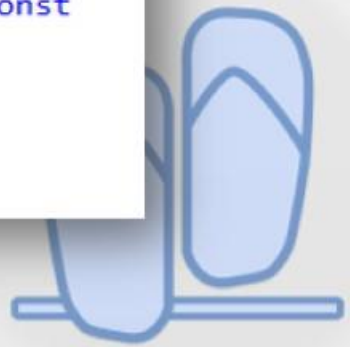
No context-switch can interrupt ✓

Hooray! It's thread-safe! ?

Consider this simple class:

```
1   #include <cstdint>
2
3   using measure_t = int64_t;
4
5   class Measures
6   {
7   private:
8       measure_t voltage = 0;
9       measure_t current = 0;
10  protected:
11      void setVoltage(measure_t v);
12      void setCurrent(measure_t p);
13  public:
14      measure_t getVoltage() const;
15      measure_t getCurrent() const;
16  };
17
```

```
18  void Measures::setVoltage(measure_t v)
19  {
20      voltage = v;
21  }
22
23  void Measures::setCurrent(measure_t p)
24  {
25      current = p;
26  }
27
28  measure_t Measures::getVoltage() const
29  {
30      return voltage;
31  }
32
33  measure_t Measures::getCurrent() const
34  {
35      return current;
36  }
37
```



… and its compiled assembly:

```
1  Measures::setVoltage(long long):
2          stm     r0, {r2-r3}
3          bx      lr
4  Measures::getVoltage() const:
5          ldmia   r0, {r0-r1}
6          bx      lr
```

Set/Get `voltage` use single instructions.

QED!

Yes, but...

```
 7   Measures::setCurrent(long long):
 8           str     r2, [r0, #8]
 9           str     r3, [r0, #12]
10           bx      lr
11   Measures::getCurrent() const:
12           add     r1, r0, #8
13           ldmia   r1, {r0-r1}
14           bx      lr
```

Turns out the compiler might surprise us.

Lessons

- The compiler **can** use a single instruction ≠ it always does

And What If We Ensure That?

- So, it *might* not use a single instruction.
- But what if we make sure it does?
- E.g., we verify the assembly or write it directly, so we have

```
1  Measures::setVoltage(long long):
2          stm     r0, {r2-r3}
3          bx      lr
4  Measures::getVoltage() const:
5          ldmia   r0, {r0-r1}
6          bx      lr
```

- Then what?

It Might Still Not Work.

For example, when using –

- Multi-core
- DMA
- Memory-mapped IO
- Non-cached memory

"Single instruction" is *not* enough.

Others may get it in a few phases,

and may read a *partial* update.

Or... It May Be a Time Bomb

Work perfectly in the given settings.

Never show a problem, under millions of tests.

Not providing any clue in the code that it is very fragile.

And then,
one day,
due to some change in the architecture,
the BSP settings,
or somewhere else in the code –
go off.

Confucius

He who fails
to plan
for the long term
will soon
face
immediate troubles.

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break

Using a Native Word

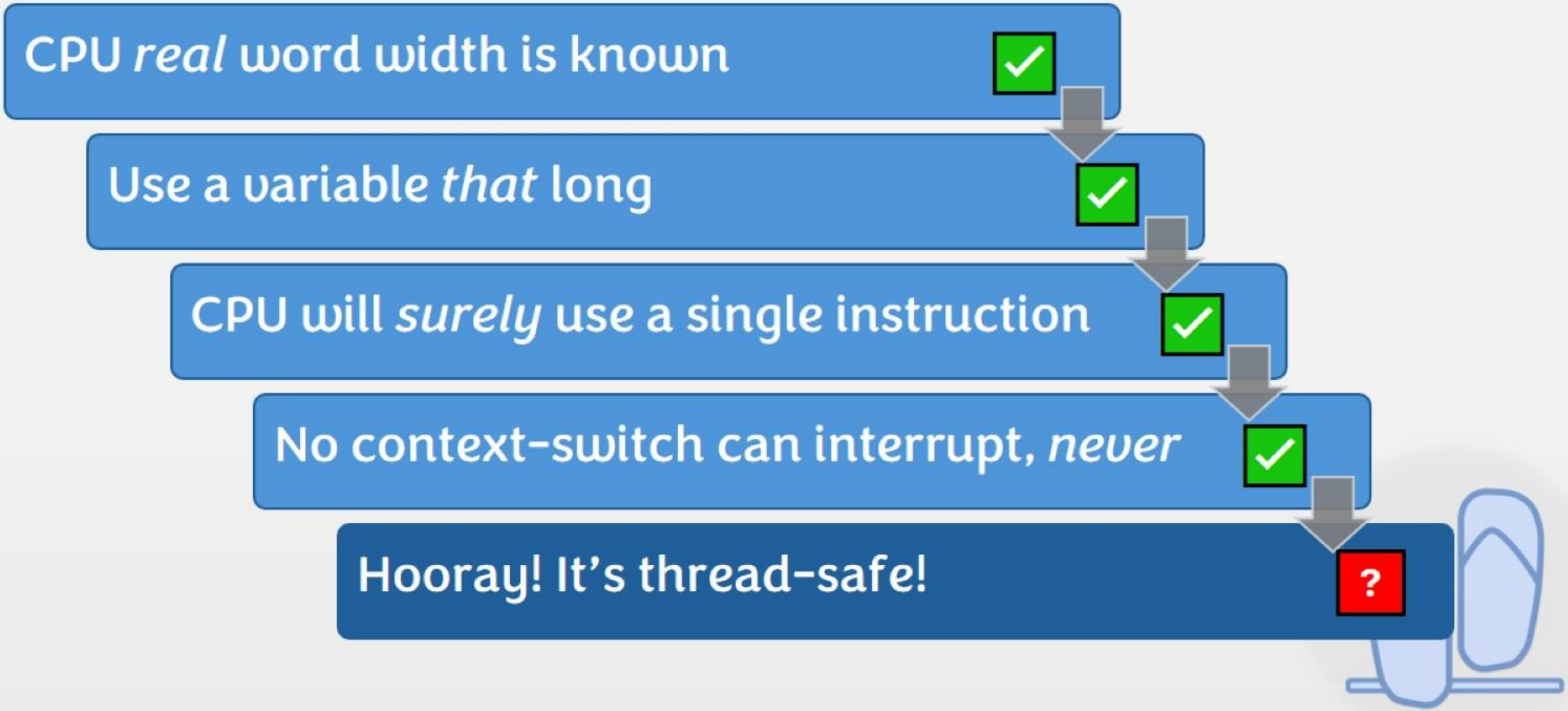
Rely on the CPU true native word size

Atomicity of a Really Native Word

So the CPU I/O is misleading.

What about its real internals?
Its native, register size?

It works like this:



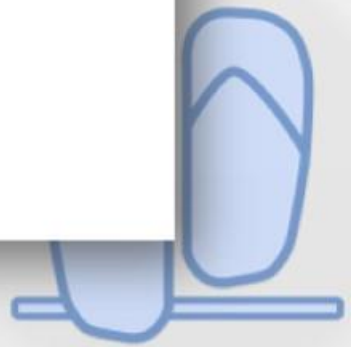
For Example, 32bit on a 32bit Arch:

Changing type to 32bit:

```
1   #include <cstdint>
2
3   using measure_t = int32_t;
4
```

Produces:

```
1   Measures::setVoltage(long):
2           str     r1, [r0]
3           bx      lr
4   Measures::setPower(long):
5           str     r1, [r0, #4]
6           bx      lr
7   Measures::getVoltage() const:
8           ldr     r0, [r0]
9           bx      lr
10  Measures::getPower() const:
11          ldr     r0, [r0, #4]
12          bx      lr
```



Looks All Right Now!

A single Store instruction.

A single Load instruction.

Is there any reason for the compiler *not* to use one instruction?

Well, At Least One: Alignment

That single, native element must be properly aligned.

Why wouldn't it?

It is not rare in RT-Embedded systems:

- To match a specific communication protocol
- To allow different machines share the same structures
- To match legacy code from an older architecture
- ...

This is when natively aligned:

```
1   #include <cstdint>
2
3   using volatge_t = int16_t;
4   using power_t = int32_t;
5
6
7   class Measures
8   {
9   private:
10      volatge_t voltage = 0;
11      power_t power = 0;
12  protected:
13      void setVoltage(volatge_t v);
14      void setPower(power_t p);
15  public:
16      volatge_t getVoltage() const;
17      power_t getPower() const;
18  };
```

```
1   Measures::setVoltage(short):
2           strh    r1, [r0]        @ movhi
3           bx      lr
4   Measures::setPower(long):
5           str     r1, [r0, #4]
6           bx      lr
7   Measures::getVoltage() const:
8           ldrsh   r0, [r0]
9           bx      lr
10  Measures::getPower() const:
11          ldr     r0, [r0, #4]
12          bx      lr
```

And that's what happens when it's not:

```cpp
#include <cstdint>

using volatge_t = int16_t;
using power_t = int32_t;

#pragma pack(2)
class Measures
{
private:
    volatge_t voltage = 0;
    power_t power = 0;
protected:
    void setVoltage(volatge_t v);
    void setPower(power_t p);
public:
    volatge_t getVoltage() const;
    power_t getPower() const;
};
```

```asm
Measures::setVoltage(short):
        strh    r1, [r0]        @ movhi
        bx      lr
Measures::setPower(long):
        lsr     r3, r1, #16
        strh    r1, [r0, #2]    @ movhi
        strh    r3, [r0, #4]    @ movhi
        bx      lr
Measures::getVoltage() const:
        ldrsh   r0, [r0]
        bx      lr
Measures::getPower() const:
        ldrh    r3, [r0, #2]
        ldrh    r0, [r0, #4]
        orr     r0, r3, r0, lsl #16
        bx      lr
```

The logic itself did not change a bit!

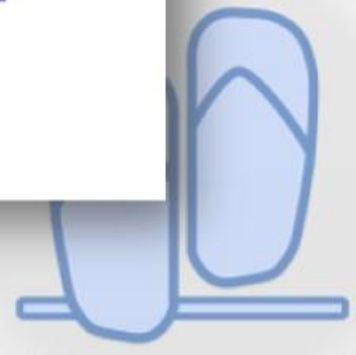
The only change is elsewhere.

The logic is exactly the same.
Simple, clear, innocent.
Fully operational.

Unless –
you know it must be atomic.

Can you?

```
20  void Measures::setVoltage(volatge_t v)
21  {
22      voltage = v;
23  }
24
25  void Measures::setPower(power_t p)
26  {
27      power = p;
28  }
29
30  volatge_t Measures::getVoltage() const
31  {
32      return voltage;
33  }
34
35  power_t Measures::getPower() const
36  {
37      return power;
38  }
39
```



Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- **Non-native alignment can break atomicity**

Meaning of “Atomicity”

Is that what we actually need?

What is Probably Guaranteed?

- Can a context-switch leave a torn value in `voltage`?
  Usually... No,
  assuming that cache [same core] is used.

- Can another core see a torn value in `voltage`?
  Probably No,
  assuming that coherent cache [same CPU] is used (subject to processor spec).

- The above is true, for example, for ARM spec.
  Exotic processors might have special behavior.

What is *Not* Guaranteed?

- Can *another component* see a torn value in `voltage`?
  Maybe.
  It depends on peripherals – e.g. DMA, buses...

- Can a *non-cached* access see a torn value in `voltage`?
  Maybe.
  It depends on the architecture of RAM, controllers etc.

Must Values be Written/Read?

Sometimes, sync objects are used to verify no value is lost.

Atomicity alone *does not* guarantee that.
Usually that's enough.
Sometimes not.

Say that we want to set `voltage` in gradient phases:

```
39  void Measures::setVoltageProfile(Measures& m, measure_t start, measure_t add)
40  {
41      for (int i = 0; i <= 1024; ++i)
42          m.setVoltage(start + i * add);
43  }
44
```

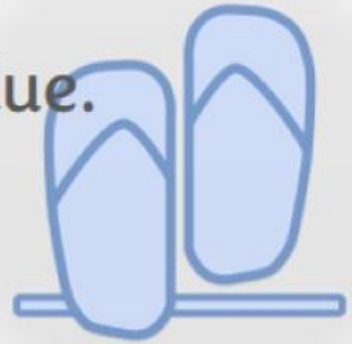
The Compiler is Smarter than Us...

Why work hard when the final value of `voltage` is known?

This is what the compiler (-O3) produces:

```
13    Measures::setVoltageProfile(Measures&, long, long):
14            add     r2, r2, r3, lsl #10
15            str     r2, [r1]
16            bx      lr
```

Polling on `voltage` will only show the final `start+add*1024` value.



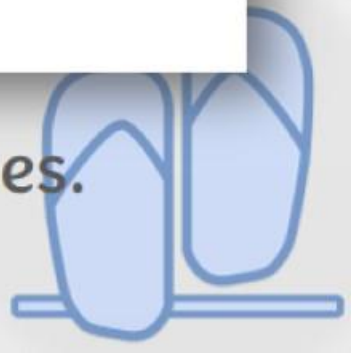
Using `volatile` Qualification

To make sure every call to `setVoltage()` actually updates it –
we may try to use `volatile`:

```
5   class Measures
6   {
7   private:
8       volatile measure_t voltage = 0;
9       volatile measure_t power = 0;
```

```
13  Measures::setVoltageProfile(Measures&, long, long):
14          ldr     r0, .L10
15  .L7:
16          subs    r0, r0, #1
17          str     r2, [r1]
18          add     r2, r2, r3
19          bne     .L7
20          bx      lr
21  .L10:
22          .word   1025
```

Still –
no guarantee that all values will propagate to other cores/devices.



Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- **Atomicity ≠ actual writing/reading**

Get. Set. What about Change?

One day, someone might change the interface a bit.
For example, by adding this little function:

```
40   void Measures::increasePower(measure_t add)
41   {
42       power += add;
43   }
44
```

Small and innocent change, but no more atomic power.

No More Single Instruction

Our new `increasePower()` now reads, then writes.

```
13    Measures::increasePower(long):
14            ldr     r3, [r0, #4]
15            add     r3, r3, r1
16            str     r3, [r0, #4]
17            bx      lr
```

Race condition is back.

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- **Atomicity of write and read ≠ atomicity of “change”**

What Atomicity Level We Need?

Atomicity of native types is only relevant to a *single* datum.

Many times, atomicity requires more than that.

```
43   void Measures::setBoth(measure_t v, measure_t p)
44   {
45       voltage = v;
46       power = p;
47   }
48
49   std::pair<measure_t, measure_t> Measures::getBoth() const
50   {
51       return { voltage, power };
52   }
53
```

What Might Happen Now?

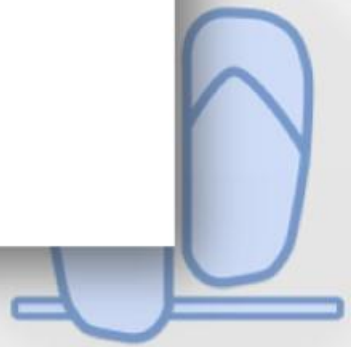
Writing/reading the pair is no more atomic:

- One might read the new `voltage` with the old `power`.
- Or, in some cases – even new `power` with the old `voltage`.

Maybe a Flag Will Help?

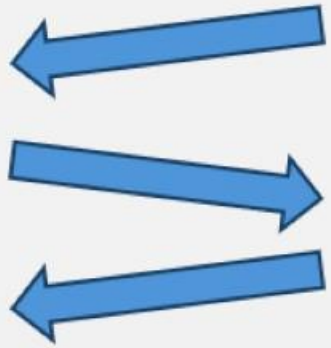
```
 8   private:
 9       volatile measure_t voltage = 0;
10       volatile measure_t power = 0;
11       volatile bool valid = true;
```

```
57   void Measures::setBothWithFlag(measure_t v, measure_t p)
58   {
59       valid = false;
60       voltage = v;
61       power = p;
62       valid = true;
63   }
64
65   std::pair<measure_t, measure_t> Measures::getBothWithFlag() const
66   {
67       if (valid)
68           return { voltage, power };
69       else
70           return { 0, 0 };
71   }
```



Let's See:

```
Void Measures::
  setBothWithFlag
  (measure_t v, measure_t p)
{

    valid = false;
    voltage = v;


    power = p;
    valid = true;
}
```



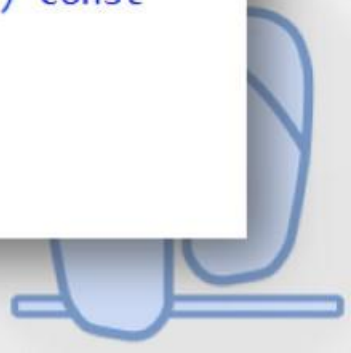
```
std::pair <measure_t, measure_t>
  Measures::
  getBothWithFlag() const
{
    if (valid) // true


        return { voltage, power };


    else
        return { 0, 0 };
}
```

What about Double Buffer?

- Use 2 sets
- Add an “index”
- Update index after change

```
 8   private:
 9       volatile measure_t voltage[2] = { 0, 0 };
10       volatile measure_t power[2] = { 0, 0 };
11       volatile int index = 0;
```

```
26   void Measures::setBothDoubleBuff(measure_t v, measure_t p)
27   {
28       int i = !index;
29       voltage[i] = v;
30       power[i] = p;
31       index = i;
32   }
33
34   std::pair<measure_t, measure_t> Measures::getBothDoubleBuff() const
35   {
36       int i = index;
37       return { voltage[i], power[i] };
38   }
```



Will it Work?

If calls to `setBothDoubleBuff()` might occur simultaneously:
➔ Race condition can still occur

What if it is guaranteed never to happen?

In this case, `getBothDoubleBuff()` should read a valid pair – either a former one or the new one.
Isn't it?

Read Seems to be Reliable:

```
void Measures::
  setBothDoubleBuff
  (measure_t v, measure_t p)
{
   int i = !index;
   voltage[i] = v;
   power[i] = p;
   index = i;
}
```

```
std::pair
  <measure_t, measure_t>
  Measures::
  getBothDoubleBuff() const
{
   int i = index;
   return
     {voltage[i], power[i]};
}
```

… But It's Not Necessarily So!

If it is –

- on a single-core,
- using L1 cache,
- with atomicity of `int`,
- that is properly aligned,
- and qualified as `volatile`

➔ it might be safe.

For a multi-core system – this is not necessarily true.

This is What Happens:

This is the current state –
active buffer is 1, values are 100, 200.

	0	1
voltage	10	100
power	20	200
index	1	

Then, this is called:

```
setBothDoubleBuff(1000, 2000)
```

	0	1
voltage	1000	100
power	20	200
index	1	

	0	1
voltage	1000	100
power	2000	200
index	1	

	0	1
voltage	1000	100
power	2000	200
index	0	

• • •

	0	1
voltage	1000	100
power	20	200
index	1	

	0	1
voltage	1000	100
power	2000	200
index	1	

	0	1
voltage	1000	100
power	2000	200
index	0	

Still, another core might not see the changes in that order; therefore, **`getBothDoubleBuff()`** might see:

	0	1
voltage	1000	100
power	20	200
index	0	

which will return the corrupted pair

```
{ 1000, 20 }
```

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- **Atomicity of individual actions ≠ keeping their order**

Priority Preemptive Writer

Rely on writing in higher priority task/thread only

Common OS Scheduling (in a very general way)

The OS provides CPU time to each thread.

Higher priority – more/longer slices;

Lower priority – less/shorter slices.

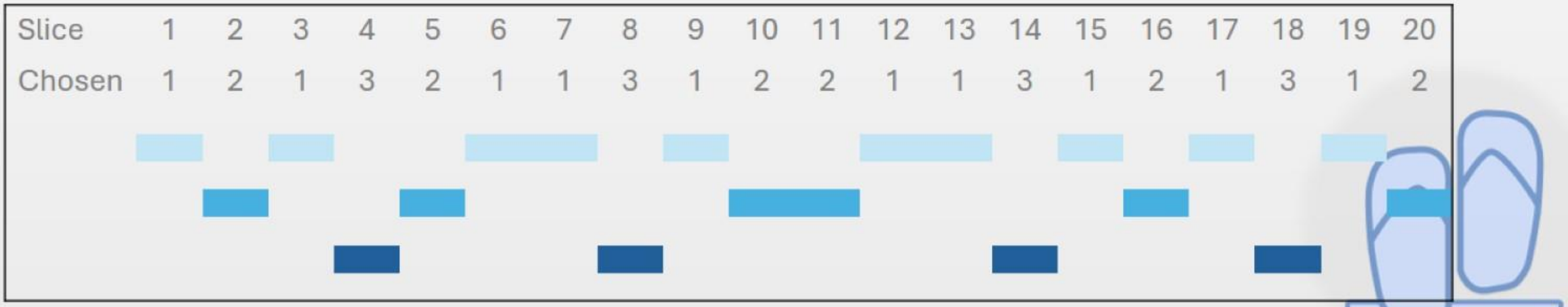
No starvation to any thread.

T1 (High Priority)

T2 (Mid Priority)

T3 (Low Priority)

Slice	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Chosen	1	2	1	3	2	1	1	3	1	2	2	1	1	3	1	2	1	3	1	2



Hard-Real-time Requirements

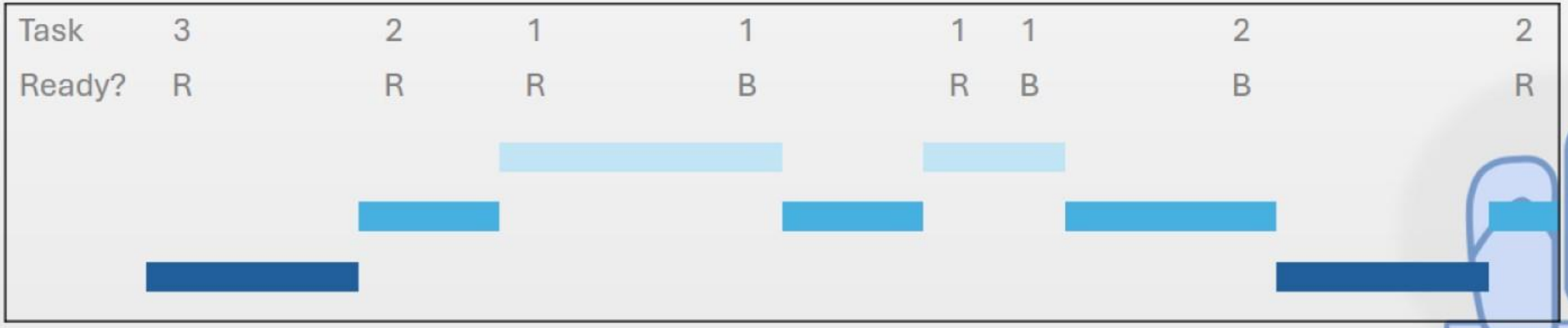
In hard-real-time systems, some actions are “time bound”:
They *must* complete within a given period.

This cannot be guaranteed under common OS scheduling.
Therefore, RTOSs use ***preemptive scheduling***.

Preemptive Scheduling (in a very general way)

Higher priority is absolute:
When a higher priority task is ***ready*** –
All lower priority tasks are paused.
Only when it is ***blocked***, will they resume.

T1 (High Priority)
T2 (Mid Priority)
T3 (Low Priority)



Let's Look Again at `set/getBoth()`:

What if –

- `setBoth()` is only being called from priority *P*
- `getBoth()` is only being called from priorities *lower* than *P*

```
43  void Measures::setBoth(measure_t v, measure_t p)
44  {
45      voltage = v;
46      power = p;
47  }
48
49  std::pair<measure_t, measure_t> Measures::getBoth() const
50  {
51      return { voltage, power };
52  }
53
```

It works like this:

Write only in a single task of priority P ✓

Read only from lower priorities ✓

Read will never stop **write** in the middle ✓

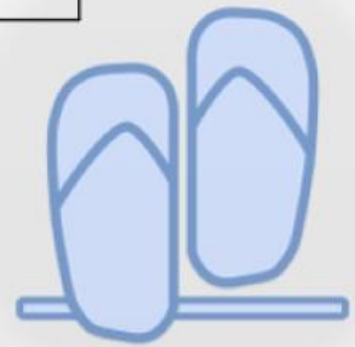
All **writes** are atomic related to **reads** ✓

Hooray! It's thread-safe! ?

What About Multi-Core?

Of course, preemption is only relevant in single-core.

On multi-core, few tasks run in parallel all the time.



And on a Single-Core?

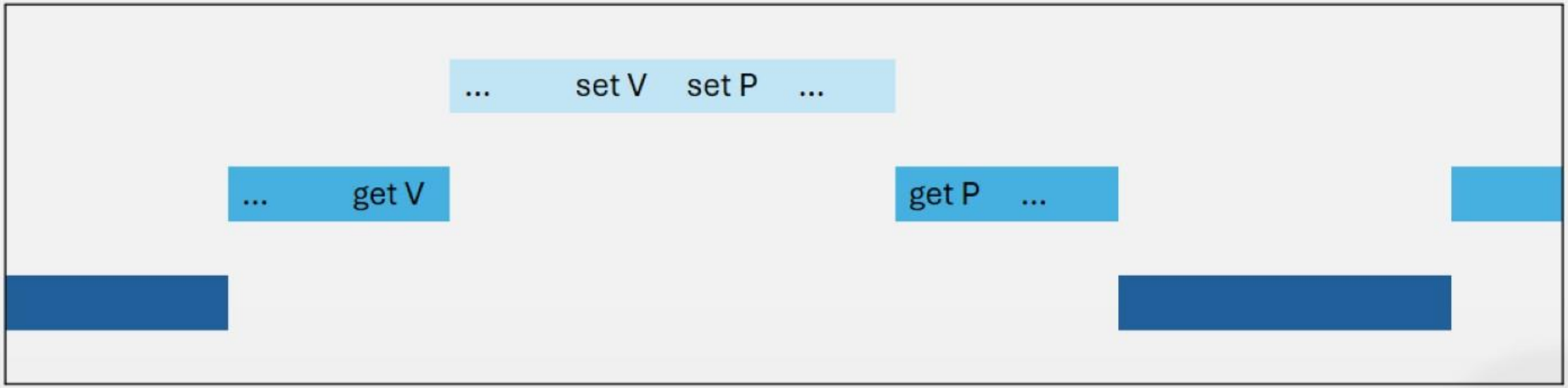
Let's take a second look at that:

Read will never stop **write** in the middle ✅

But what about the opposite?

Write During Read

Write (`setBoth()`) can happen during **read** (`getBoth()`):



So that `getBoth()` returns the *old* `voltage` with the *new* `power`.

And More...

Let's take a *third* look at that:

Read will never stop **write** in the middle ?

Is that true at all?

Preemptive Scheduling – Deeper Look

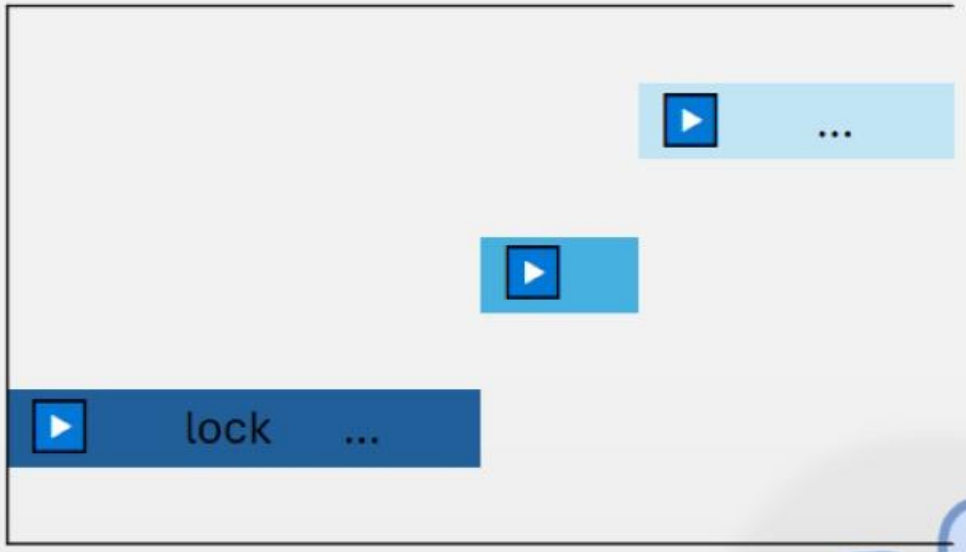
Preemptive scheduling is simple.
It allows higher-priority tasks to finish as fast as possible.
It provides absolute control about the priorities.

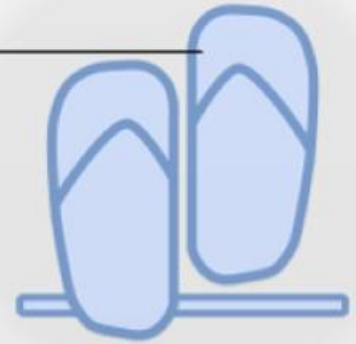
But it comes with costs and anomalies.

Priority Inversion

Consider tasks L, M, H (low, medium, high priority accordingly).

- L runs first; it locks a mutex.
- Then M becomes ready and preempts it.
- Then H becomes ready and preempts it.
- H tries locking the mutex held by L.
- H then blocks;
  M continues running instead.
- When M finishes, L continues running – releasing the mutex.
- Then H continues.

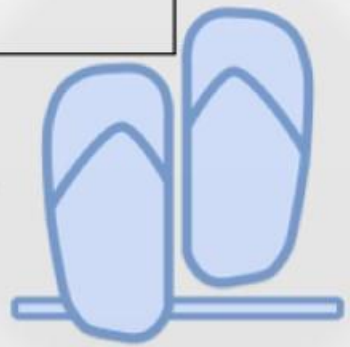




So…

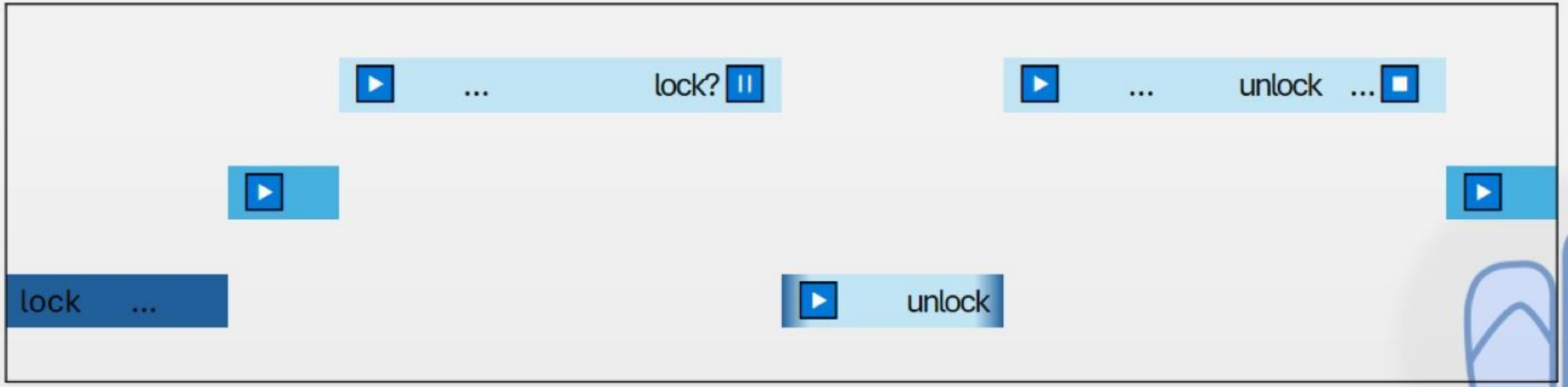
H is being starved while **M** runs freely –
because **M** prevents **L** from unlocking the mutex.

This is called ***Priority Inversion***: M is de-facto “higher” than H.



Priority Inheritance

To prevent this, many RTOSs use ***priority inheritance***:
If a high-priority task is blocked by a lower one –
the blocker temporarily *inherits* the higher priority.



Therefore –

This former observation

Read will never stop **write** in the middle ✗

might be totally wrong.

Again:

It is possible that in

your current code,

your current design,

your current CPU architecture,

your current OS,

your current BSP definitions –

both read and write are 100% atomic related to each other.

But without a clear notion of race-condition protection –

this is a time-bomb, waiting for the slightest change.

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- Atomicity requires both write and read to atomic
- Even in preemptive-scheduling, priority may change

Pausing Interrupts

The popular RT “Critical Section” solution

Pausing Interrupts

This solution is very popular in RT systems.

For example, FreeRTOS provides

`taskENTER_CRITICAL()`, `taskEXIT_CRITICAL()`,

and a few more.

Even the OS itself uses that mechanism.

It works like this:

Pause reaction to any interrupt ✓

No ISR will interrupt during the section ✓

No context-switch will happen ✓

Whatever is in the section is atomic ✓

Hooray! It's thread-safe! ?

Handle With Care!

In some cases, this is a good solution.

In some cases, this is a not-so-good solution.

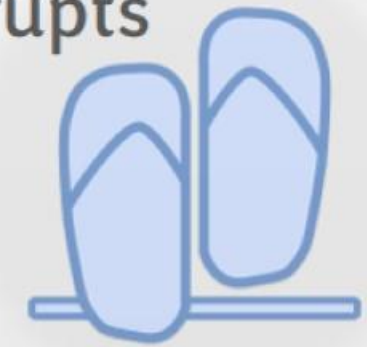
And there are cases in which it is not a solution at all.

Cases Where It's Irrelevant

- **On multi-core:**
  Other threads continue running on the other cores

- **When racing with other data sources:**
  The DMA works independently

- **When higher-priority ISRs are involved:**
  Some mechanisms might block only lower-priority interrupts



Cases Where It's Risky

- **Too many, or too long such sections**
  Interrupts/high-priority tasks get starved – priority inversion

- **Blocking operations**
  May result in deadlock – no one will ever release them

- **When no ISR is involved**
  Prefer task-level solution, e.g. prevent context-switch instead

When It's Fine

When –

- on a single core
- no DMA access from other devices
- the critical section involves ISRs and tasks
- but no too-high interrupt level... (that will not stop)
- section is very short
- no blocking or time-unbound calls

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- Atomicity requires both write and read to atomic
- Even in preemptive-scheduling, priority may change
- Pausing interrupts cannot protect from other cores/devices
- Blocking while interrupts are paused will cause a dead-lock
- Involving entities not related to the critical section harms priorities

Using Delays

The craftsmanship of fixing without solving

Just Make Them Not Collide!

We have all been there.

Strange problems with the new feature...

Something else is suddenly break.

Maybe it's a race condition.

Maybe something darker.

To really understand it will take time.

The delivery is in 3 hours.

So... Let's just make sure those two do not collide!

It works like this:

I have no idea what the problem is...

Tried delay of 250µsec; didn't work.

Tried delay of 500µsec; New problem.

Tried delay of 317µsec; It works.

Hooray! Fixed it!

It comes in flavors:

Using a timer

Using sleep()

Using busy-wait

Don’t.

No need to explain that, right?

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- Atomicity requires both write and read to atomic
- Even in preemptive-scheduling, priority may change
- Pausing interrupts cannot protect from other cores/devices
- Blocking while interrupts are paused will cause a dead-lock
- Involving entities not related to the critical section harms priorities
- Timer ≠ sync. Starting a delay → count-down to detonation

Get Modern

Use atomic operations, especially with std::atomic

Atomic Operations

Modern processors provide modern sync solutions.

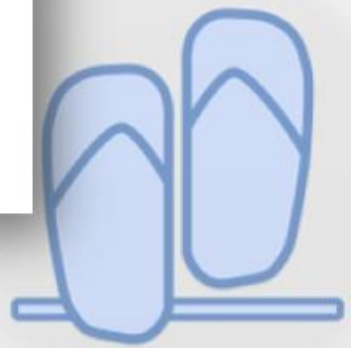
Modern C++ provides unified and easy interface for that:

`std::atomic`.

The Same Code, with Atomics:

```cpp
#include <atomic>

using measure_t = int32_t;
using volatge_t = measure_t;
using power_t = measure_t;

class Measures
{
private:
    std::atomic<volatge_t> voltage = 0;
    std::atomic<power_t> power = 0;
protected:
    void setVoltage(volatge_t v);
    void setPower(power_t p);
    void increasePower();
public:
    volatge_t getVoltage() const;
    power_t getPower() const;
};
```

```cpp
void Measures::setVoltage(volatge_t v)
{
    voltage = v;
}

void Measures::setPower(power_t p)
{
    power = p;
}

void Measures::increasePower()
{
    ++power;
}

volatge_t Measures::getVoltage() const
{
    return voltage;
}

power_t Measures::getPower() const
{
    return power;
}
```



Atomic: The Good, The Bad, The Ugly

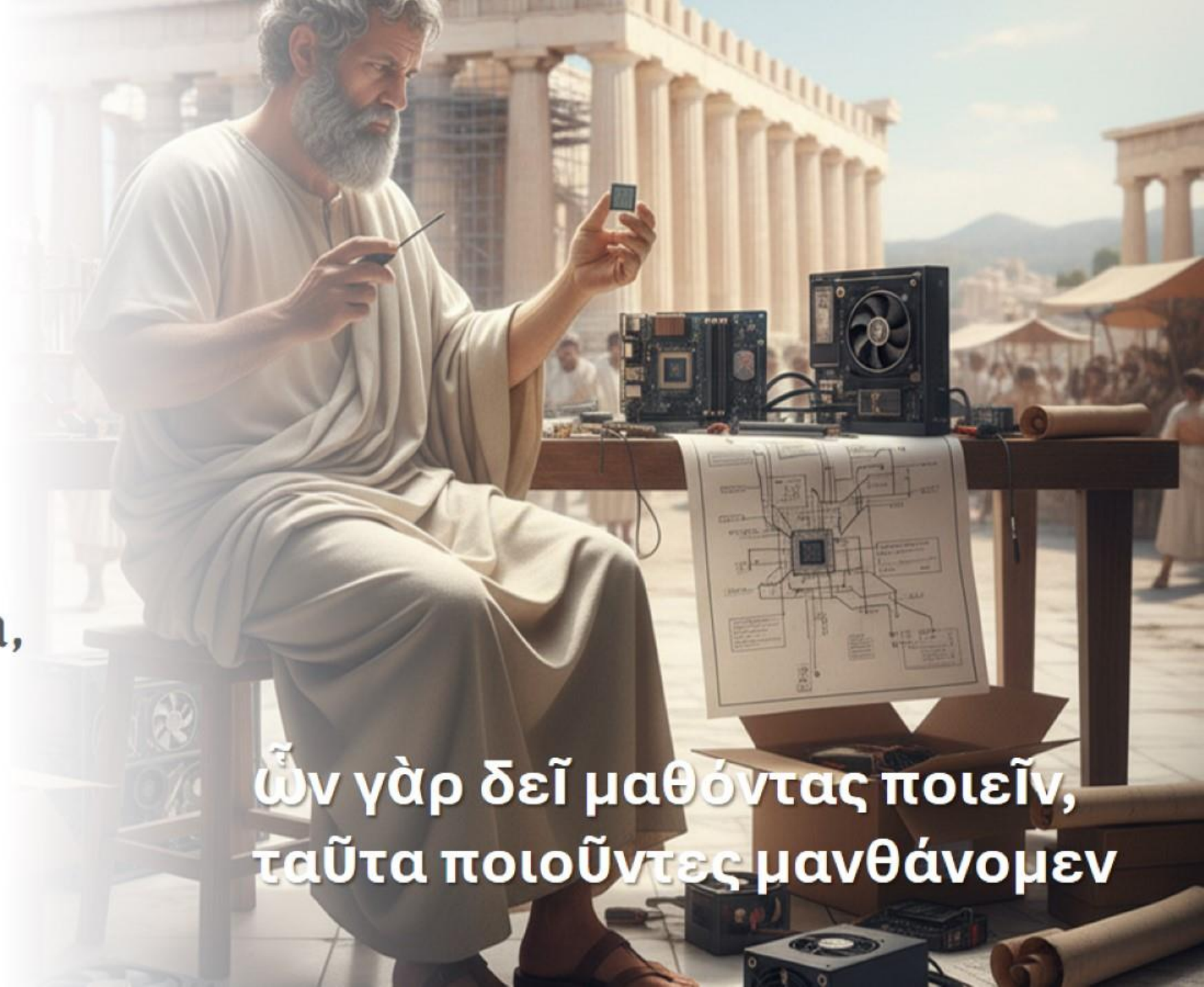
The Good:
- Unified, clear, cross-platform interface
- Very intuitive for naïve use

The Bad:
- Might have *very different* implementation on each platform
- Naïve use might result in sub-optimal performance

The Ugly:
- Precise control is possible, but severely compromises simplicity
- Use this trade-off wisely

Aristotle

For the things
we must learn
before
we can do them,
we learn by
doing them.



Example: Atomic Write (uint32_t)

Some ARM Cortex A55

```
1   Measures::setVoltage(int):
2           stlr    w1, [x0]
3           ret
4   Measures::setPower(int):
5           add     x0, x0, 4
6           stlr    w1, [x0]
7           ret
```

Some ARM Cortex A9

```
1   Measures::setVoltage(long):
2           dmb     ish
3           str     r1, [r0]
4           dmb     ish
5           bx      lr
6   Measures::setPower(long):
7           dmb     ish
8           str     r1, [r0, #4]
9           dmb     ish
10          bx      lr
```

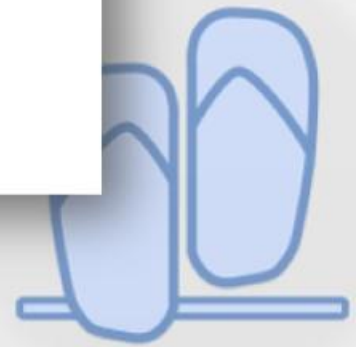
Example: Atomic Increase (uint32_t)

Some ARM Cortex A55

```
 8   Measures::increasePower():
 9           add     x0, x0, 4
10           mov     w1, 1
11           ldaddal w1, w1, [x0]
12           ret
```

Some ARM Cortex A9

```
11   Measures::increasePower():
12           adds    r0, r0, #4
13           dmb     ish
14   .L5:
15           ldrex   r3, [r0]
16           adds    r3, r3, #1
17           strex   r2, r3, [r0]
18           cmp     r2, #0
19           bne     .L5
20           dmb     ish
21           bx      lr
```



… Atomic Increase (uint32_t)

Some ARM Cortex A9

```
11  Measures::increasePower():
12          adds    r0, r0, #4
13          dmb     ish
14  .L5:
15          ldrex   r3, [r0]
16          adds    r3, r3, #1
17          strex   r2, r3, [r0]
18          cmp     r2, #0
19          bne     .L5
20          dmb     ish
21          bx      lr
```

We have a *loop*.

➔This is no more time-bound!

If there are *a lot* of accesses,
at the *same time*,
from *different cores* –
it might take longer.

Example: Atomic Write (32/64 bit)

A55: 32/64 bit

```
1   Measures::setVoltage(int):
2           stlr    w1, [x0]
3           ret
```

```
1   Measures::setVoltage(long):
2           stlr    x1, [x0]
3           ret
```

A9: 32/64 bit

```
1   Measures::setVoltage(long):
2           dmb     ish
3           str     r1, [r0]
4           dmb     ish
5           bx      lr
```

```
1   Measures::setVoltage(long long):
2           push    {fp}
3           dmb     ish
4   .L2:
5           ldrexd  fp, ip, [r0]
6           strexd  r1, r2, r3, [r0]
7           cmp     r1, #0
8           bne     .L2
9           dmb     ish
10          ldr     fp, [sp], #4
11          bx      lr
```

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- Atomicity requires both write and read to atomic
- Even in preemptive-scheduling, priority may change
- Pausing interrupts cannot protect from other cores/devices
- Blocking while interrupts are paused will cause a dead-lock
- Involving entities not related to the critical section harms priorities
- Timer ≠ sync. Start a delay ➔ count-down to detonation
- std::atomic implementation may dramatically vary across platforms

Summary

The things we don’t know that we don’t know

Never Assume.

- Just because it works,
  does not mean it's bug-free.
- Just because it compiled without bugs,
  does not mean the next compilation will.
- Just because your code isn't subject to compiler changes now,
  does not mean maintainers will keep it that way
- Just because it works here
  does not mean it will work there

So:

Make sure that
you
know what you are doing.

And not less important:

Make sure that
future code maintainers
know what you were doing.

Descartes

I code,
therefore
I bug.

The End | Q & A

Are There Time Bombs in Your Real-Time Code?

Mike Lindner

mike.Lindner@gmail.com

https://linkedin.com/in/mike-lindner

https://sw-arch.blog/
Architect in Slippers | ארכיטקט בכפכפים

Lessons

- The compiler **can** use a single instruction ≠ it always does
- Single instruction store/load ≠ other cores see it as one
- CPU I/O width ≠ RAM, DMA or other buses width
- Code that not expressly define “atomicity” – might break
- Non-native alignment can break atomicity
- Atomicity ≠ actual writing/reading
- Atomicity of write and read ≠ atomicity of “change”
- Atomicity requires both write and read to atomic
- Even in preemptive-scheduling, priority may change
- Pausing interrupts cannot protect from other cores/devices
- Blocking while interrupts are paused will cause a dead-lock
- Involving entities not related to the critical section harms priorities
- Timer ≠ sync. Start a delay ➔ count-down to detonation
- std::atomic implementation may dramatically vary across platforms