!concurrent,worldHello
or
concurrency in the eyes of VM engineer

Alexander Filatov
filatov.alexandr@huawei.com

Bio: Alexander Filatov

VM engineer with 10+ years experience.

- 2014 - 2019, Excelsior JVM with AOT compilation[1]
- 2019 - now, Huawei, Languages and Compilers lab

[1] https://dl.acm.org/doi/10.1145/584369.584387

Bio: Alexander Filatov

VM engineer with 10+ years experience.

- 2014 - 2019, Excelsior JVM with AOT compilation[1]
- 2019 - now, Huawei, Languages and Compilers lab

Scope of my daily work: run-time of Virtual Machine
Deep specialization: Garbage Collection

[1] https://dl.acm.org/doi/10.1145/584369.584387

Bio: Alexander Filatov

Areas of interest:

- automatic memory management

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism
- weak memory models

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism
- weak memory models
- correctness of thread-safe data structures

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism
- weak memory models
- correctness of thread-safe data structures

Personal experience: debugged a lot of

- my own parallel/concurrent code

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism
- weak memory models
- correctness of thread-safe data structures

Personal experience: debugged a lot of

- my own parallel/concurrent code
- 3rd-party concurrent solutions

Bio: Alexander Filatov

Areas of interest:

- automatic memory management
- concurrency and parallelism
- weak memory models
- correctness of thread-safe data structures

Personal experience: debugged a lot of

- my own parallel/concurrent code
- 3rd-party concurrent solutions
- AOT/JIT compiler

Name of this talk

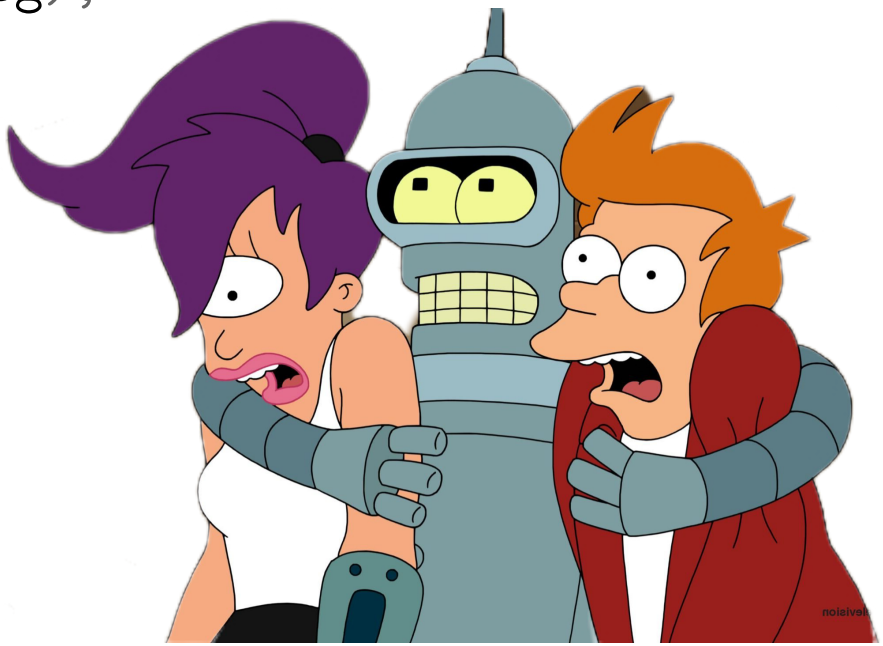
!concurrent,worldHello

```
for (String msg : new String[] {
        "Hello", ",", "concurrent", "world", "!"
    }) {
    new Thread() {
        public void run() {
            System.out.println(msg);
        }
    }.start();
}
```

Name of this talk

!concurrent,worldHello

```
for (String msg : new String[] {
        "Hello", ",", "concurrent", "world", "!"
    }) {
    new Thread() {
        public void run() {
            System.out.println(msg);
        }
    }.start();
}
```



Plan

We are here

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Time passes ...

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Time passes ...

- Everything became slower

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Time passes ...

- Everything became slower
- Application randomly hangs

Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Time passes ...

- Everything became slower
- Application randomly hangs
- Source code is hard to maintain

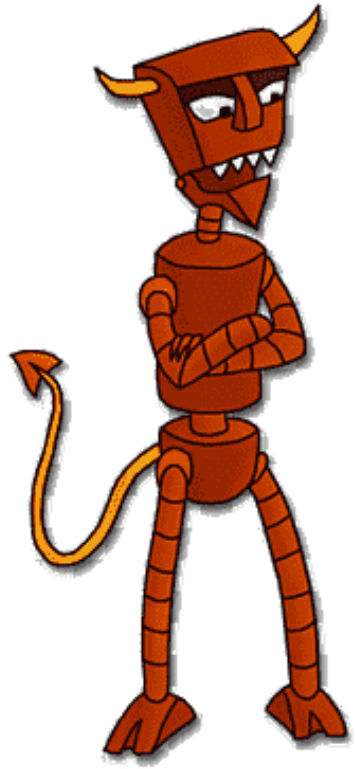
Idea №1: let's use parallelism!

Common scenario

Colleague suggests to rewrite part of a project to enable concurrency

Time passes ...

- Everything became slower
- Application randomly hangs
- Source code is hard to maintain



Idea №1: let's use parallelism!

Why did this happen?

Idea №1: let's use parallelism!

Why did this happen?

```
synchronized(objectA) {
 synchronized(objectB) {
   computeStuff();
 }
}
```

```
synchronized(objectB) {
 synchronized(objectA) {
   computeStuff();
 }
}
```

Idea №1: let's use parallelism!

Why did this happen?

```
synchronized(objectA) {
 synchronized(objectB) {
   computeStuff();
 }
}
```

```
synchronized(objectB) {
 synchronized(objectA) {
   computeStuff();
 }
}
```

It is easy to break concurrent system with single-line patch.

Idea №1: let's use parallelism!

Why did this happen?

```
synchronized(objectA) {
 synchronized(objectB) {
   computeStuff();
 }
}
```

```
synchronized(objectB) {
 synchronized(objectA) {
   computeStuff();
 }
}
```

It is easy to break concurrent system with single-line patch.
Problem ALWAYS happens after original code was developed and tested.

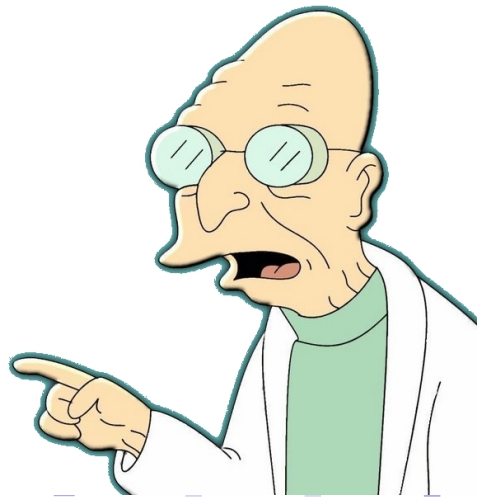
Idea №1: let's use parallelism!

Why did this happen?

```
synchronized(objectA) {
 synchronized(objectB) {
   computeStuff();
 }
}
```

```
synchronized(objectB) {
 synchronized(objectA) {
   computeStuff();
 }
}
```

It is easy to break concurrent system with single-line patch.
Problem ALWAYS happens after original code was developed and tested.



Idea №1: let's use parallelism!

Conclusion

Parallel execution is useful but concurrency:

- Adds risk (hard to find&fix bugs)
- Adds cost (hard to maintain codebase)

Idea №1: let's use parallelism!

Conclusion

Parallel execution is useful but concurrency:

- Adds risk (hard to find&fix bugs)
- Adds cost (hard to maintain codebase)

VM engineer point of view: stop! It is really hard to debug code written years ago that starts to break today after a series of unrelated commits.

Idea №1: let's use parallelism!

Conclusion

Parallel execution is useful but concurrency:

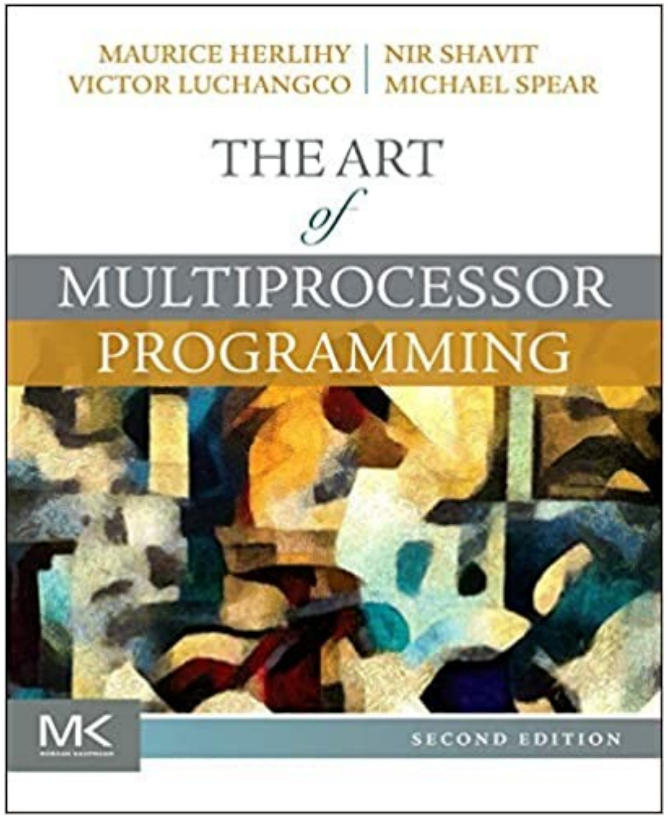
- Adds risk (hard to find&fix bugs)
- Adds cost (hard to maintain codebase)

VM engineer point of view: stop! It is really hard to debug code written years ago that starts to break today after a series of unrelated commits.
Wise person reaction: first of all, estimate pros and cons of using concurrency in YOUR scenario

Idea №2: let's use algorithm from the textbook!

VM engineer point of view

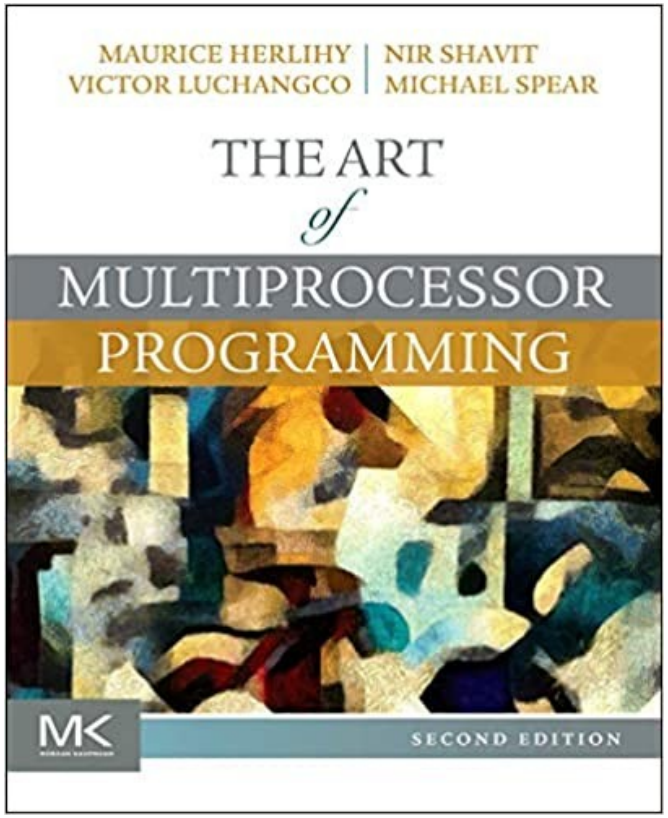
Reality



Idea №2: let's use algorithm from the textbook!

VM engineer point of view

Reality



My feelings

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language
- algorithm formalized on Promela language and verified using SPIN model checking tool

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language
- algorithm formalized on Promela language and verified using SPIN model checking tool

"Using Hardware Transactional Memory to Correct and Simplify and Readers-writer Lock Algorithm", 2013

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language
- algorithm formalized on Promela language and verified using SPIN model checking tool

"Using Hardware Transactional Memory to Correct and Simplify and Readers-writer Lock Algorithm", 2013

- 20 years later!

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language
- algorithm formalized on Promela language and verified using SPIN model checking tool

"Using Hardware Transactional Memory to Correct and Simplify and Readers-writer Lock Algorithm", 2013

- 20 years later!
- A bug was found in this algorithm, it was related to memory management. Crashed on real hardware.

Idea №2: let's use algorithm from the textbook!

KSUH lock

"A Fair Fast Scalable Reader-Writer Lock ", 1993

- Orran **K**rieger, Michael **S**tumm, Ron **U**nrau, Jonathan **H**anna
- In Proc. of the International Conference on Parallel Processing
- implemented in C language
- algorithm formalized on Promela language and **verified**[2] using SPIN model checking tool

"Using Hardware Transactional Memory to Correct and Simplify and Readers-writer Lock Algorithm", 2013

- 20 years later!
- A bug was found in this algorithm, it was related to memory management. Crashed on real hardware.

[2] partially

Idea №2: let's use algorithm from the textbook!

Conclusion

Complicated concurrent algorithms are useful but

- Add risk (hard to find&fix bugs)
- Add cost (hard to maintain codebase)

Idea №2: let's use algorithm from the textbook!

Conclusion

Complicated concurrent algorithms are useful but

- Add risk (hard to find&fix bugs)
- Add cost (hard to maintain codebase)

VM engineer point of view: do you really understand the algorithm in textbook and could prove it will work properly in your scenario?

Idea №2: let's use algorithm from the textbook!

Conclusion

Complicated concurrent algorithms are useful but

- Add risk (hard to find&fix bugs)
- Add cost (hard to maintain codebase)

VM engineer point of view: do you really understand the algorithm in textbook and could prove it will work properly in your scenario?

Wise person reaction: it is inconvenient when code maintenance could be done only by person with PhD in distributed systems

Idea №2: let's use algorithm from the textbook!

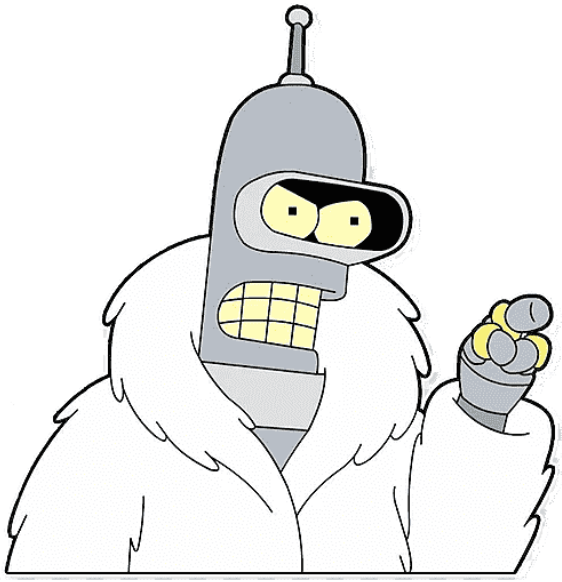
Conclusion

Complicated concurrent algorithms are useful but

- Add risk (hard to find&fix bugs)
- Add cost (hard to maintain codebase)

VM engineer point of view: do you really understand the algorithm in textbook and could prove it will work properly in your scenario?

Wise person reaction: it is inconvenient when code maintenance could be done only by person with PhD in distributed systems



Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.
It is very smart thing, that makes your synchronized blocks very fast[3].

```
while (0 != count--) {
  synchronized (jvmLock) {
    ++counter;
  }
}
```

[3] https://mechanical-sympathy.blogspot.com/2011/11/biased-locking-osr-and-benchmarking-fun.html

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.
It is very smart thing, that makes your synchronized blocks very fast[3].

```
while (0 != count--) {
  synchronized (jvmLock) {
    ++counter;
  }
}
```

Sandy Bridge 2.0GHz - Ops/Sec			
Threads	-UseBiasedLocking	+UseBiasedLocking	ReentrantLock
1	34,500,407	**396,511,324**	43,148,808

[3] https://mechanical-sympathy.blogspot.com/2011/11/biased-locking-osr-and-benchmarking-fun.html

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

- Published in 2006[4]

[4] https://dl.acm.org/doi/10.1145/1167473.1167496

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

- Published in 2006[4]
- Worked well in high-load production systems for many years

[4] https://dl.acm.org/doi/10.1145/1167473.1167496

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

- Published in 2006[4]
- Worked well in high-load production systems for many years
- "Costly to maintain"

[4] https://dl.acm.org/doi/10.1145/1167473.1167496

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

- Published in 2006[4]
- Worked well in high-load production systems for many years
- "Costly to maintain"
- Deprecated since Java 15 (JEP 374, 2019)

[4] https://dl.acm.org/doi/10.1145/1167473.1167496

Idea №2: let's use algorithm from the textbook!

Experts could use textbooks, right?

State-of-the-art production systems are fast because of smart tricks.
Example from the JVM world: biased locking.

- Published in 2006[4]
- Worked well in high-load production systems for many years
- "Costly to maintain"
- Deprecated since Java 15 (JEP 374, 2

[4] https://dl.acm.org/doi/10.1145/1167473.1167496

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Linux futex_wait() bug[5]

[5] https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64/m/BonaHiVbEmsJ

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Linux futex_wait() bug[5]

Gil Tene (CEO of Azul, co-author of C4 GC):

> *We had this one bite us hard and scare the %$ˆ out of us, so I figured I'd share the fear.*

[5] https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64/m/BonaHiVbEmsJ

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Linux futex_wait() bug[5]

Gil Tene (CEO of Azul, co-author of C4 GC):

> *We had this one bite us hard and scare the %$ˆ out of us, so I figured I'd share the fear.*
>
> *The linux futex_wait call has been broken for about a year.*

[5] https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64/m/BonaHiVbEmsJ

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Linux futex_wait() bug[5]

Gil Tene (CEO of Azul, co-author of C4 GC):

> *We had this one bite us hard and scare the %$ˆ out of us, so I figured I'd share the fear.*
>
> *The linux futex_wait call has been broken for about a year.*
>
> *The impact of this kernel bug is very simple: user processes can deadlock and hang in seemingly impossible situations. Thread.park() in Java may stay parked.*

[5] https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64/m/BonaHiVbEmsJ

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Linux futex_wait() bug[5]

Gil Tene (CEO of Azul, co-author of C4 GC):

> *We had this one bite us hard and scare the %$ˆ out of us, so I figured I'd share the fear.*
>
> *The linux futex_wait call has been broken for about a year.*
>
> *The impact of this kernel bug is very simple: user processes can deadlock and hang in seemingly impossible situations. Thread.park() in Java may stay parked.*
>
> *You'll spend a couple of months of someone's time trying to find the fault in your code, when there is nothing there to find.*

[5] https://groups.google.com/g/mechanical-sympathy/c/QbmpZxp6C64/m/BonaHiVbEmsJ

Idea №3: let's take existing solution!

Somebody on Earth could implement complicated system without bugs, right?

Idea №3: let's take existing solution!

Conclusion

- Always choose mature solutions

Idea №3: let's take existing solution!

Conclusion

- Always choose mature solutions
- Do not forget to update in time

Idea №3: let's take existing solution!

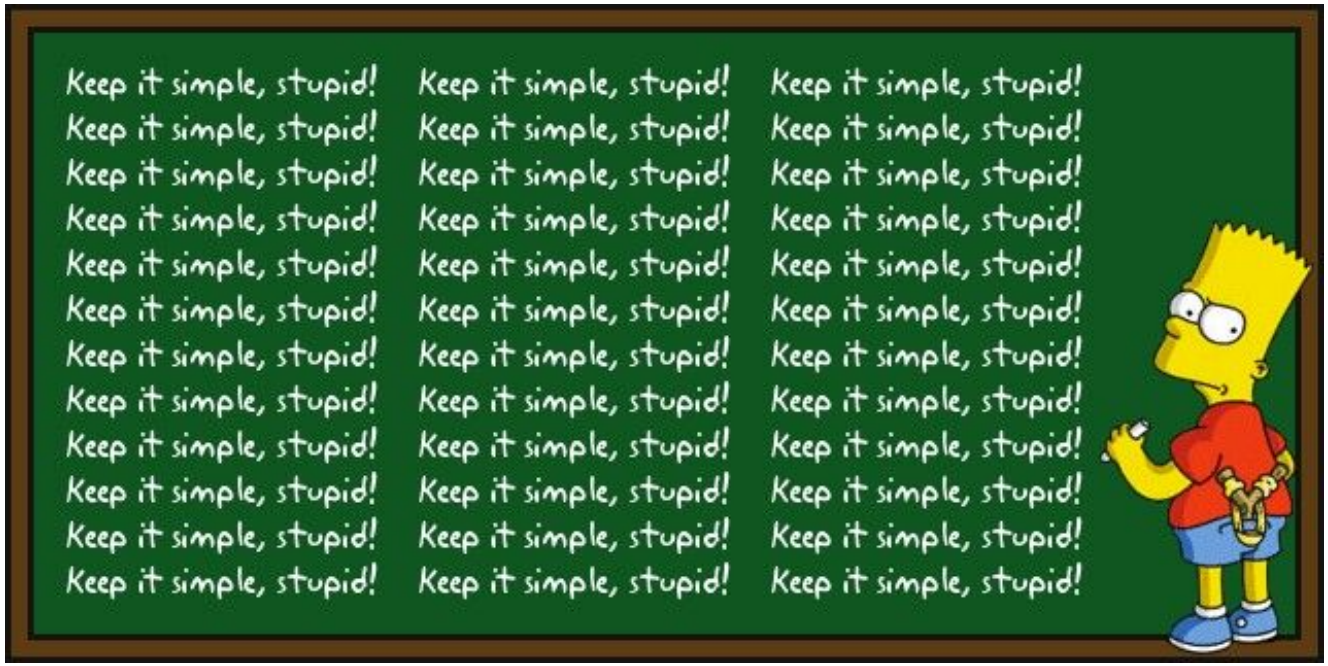
Conclusion

- Always choose mature solutions
- Do not forget to update in time
- If you could avoid tricky code – do this

Idea №3: let's take existing solution!

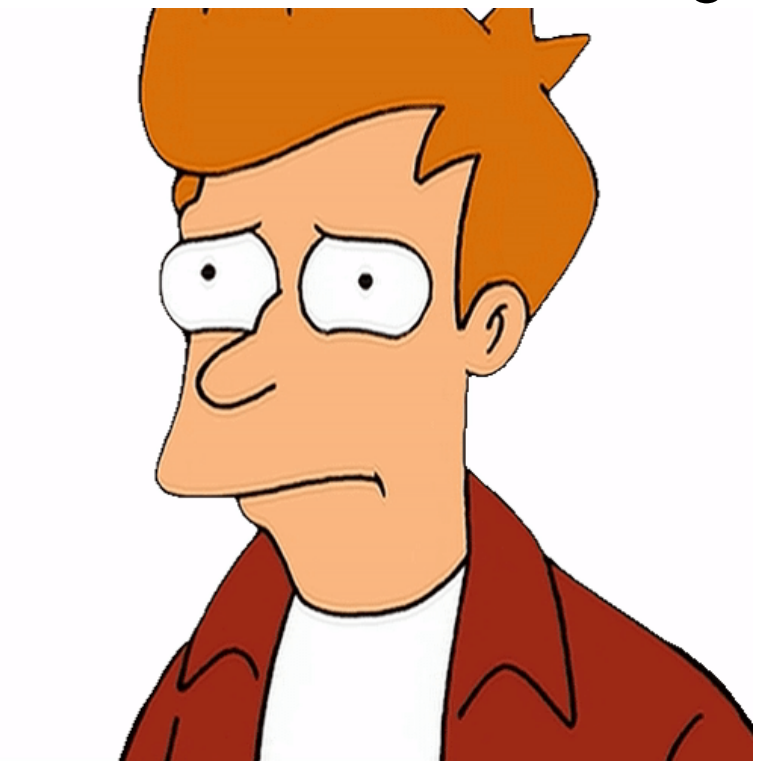
Conclusion

- Always choose mature solutions
- Do not forget to update in time
- If you could avoid tricky code – do this



Concurrency is hard

Person after a talk with VM engineer



Why this all is so complicated?

Why concurrency is hard?

Personal experience

Human brain was not designed for concurrency programming.

Why concurrency is hard?

Personal experience

Human brain was not designed for concurrency programming.
It is really hard to imagine several concurrently running threads and all possible execution inter-leavings.

Why concurrency is hard?

Personal experience

Human brain was not designed for concurrency programming.
It is really hard to imagine several concurrently running threads and all possible execution inter-leavings.

General-purpose advices

Use:

- Straightforward solutions

General-purpose advices

Use:

- Straightforward solutions
- Well-tested libraries for complicated standard tasks

General-purpose advices

Use:

- Straightforward solutions
- Well-tested libraries for complicated standard tasks
- Patterns for building concurrent systems

General-purpose advices

Use:

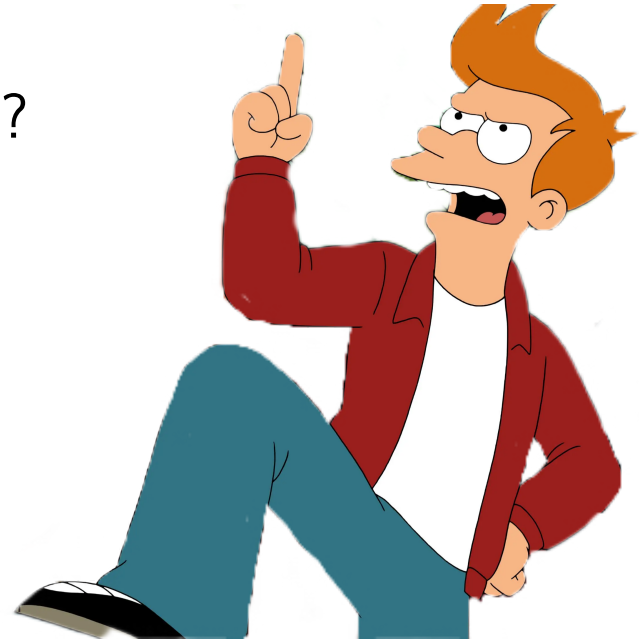
- Straightforward solutions
- Well-tested libraries for complicated standard tasks
- Patterns for building concurrent systems
- Domain-specific languages

General-purpose advices

Use:

- Straightforward solutions
- Well-tested libraries for complicated standard tasks
- Patterns for building concurrent systems
- Domain-specific languages

But what if I **really** need to write concurrent code?



We are here

Classic optimization of single-threaded code

Removing reads

```
static int a;
void foo_1() {
  while (true) {
    int tmp = a;
    if (tmp == 0) break;
    do_something_with(tmp);
  }
}
```

Classic optimization of single-threaded code

Removing reads

```
static int a;
void foo_1() {
  while (true) {
    int tmp = a;
    if (tmp == 0) break;
    do_something_with(tmp);
  }
}
```

```
// "optimized" version
void foo_2() {
  int tmp = a;
  if (tmp != 0)
    while (true) {
      do_something_with(tmp);
    }
}
```

Could compiler minimize number of memory loads and rewrite the function?

Classic optimization of single-threaded code

Godbolt

```
static int a;
void foo_1() {
  while (true) {
    int tmp = a;
    if (tmp == 0) break;
    do_something_with(tmp);
  }
}
```

Classic optimization of single-threaded code

Godbolt

```
static int a;
void foo_1() {
  while (true) {
    int tmp = a;
    if (tmp == 0) break;
    do_something_with(tmp);
  }
}
```

x86-64 clang 16.0.0 -O2[6]

x86-64 gcc 13.1 -O2[7]

```
foo_1:
        push    rbx
        mov     ebx, [a]
        test    ebx, ebx
        je      .LBB1_2
.LBB1_1:                                #<-/
        mov     edi, ebx                #   /
        call    do_something_with       #   /
        jmp     .LBB1_1                 #--/
.LBB1_2:
        pop     rbx
        ret
```

[6] https://godbolt.org/z/99j3erzaE

[7] https://godbolt.org/z/fxzGEo1qf

Classic optimization of single-threaded code

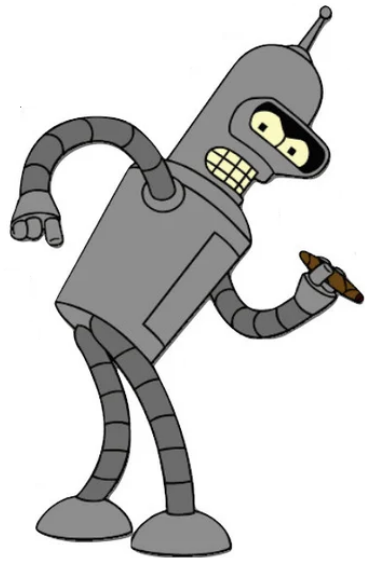
Godbolt

```
static int a;
void foo_1() {
  while (true) {
    int tmp = a;
    if (tmp == 0) break;
    do_something_with(tmp);
  }
}
```

x86-64 clang 16.0.0 -O2[6]

x86-64 gcc 13.1 -O2[7]

```
foo_1:
        push    rbx
        mov     ebx, [a]
        test    ebx, ebx
        je      .LBB1_2
.LBB1_1:                                #<-/
        mov     edi, ebx                #  /
        call    do_some                 /
        jmp     .LBB1_1                 --/
.LBB1_2:
        pop     rbx
        ret
```



[6] https://godbolt.org/z/99j3erzaE

[7] https://godbolt.org/z/fxzGEo1qf

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.
To catch and fix bugs, you will use:

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

To catch and fix bugs, you will use:

- compiler barriers[8]

[8] https://preshing.com/20120625/memory-ordering-at-compile-time/

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

To catch and fix bugs, you will use:

- compiler barriers[8]
- memory orderings (volatile, seq-cst, acquire-release...)[9]

[8] https://preshing.com/20120625/memory-ordering-at-compile-time/

[9] https://www.kernel.org/doc/html/latest/core-api/wrappers/memory-barriers.html

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

To catch and fix bugs, you will use:

- compiler barriers[8]
- memory orderings (volatile, seq-cst, acquire-release...)[9]
- concurrency primitives and their integration with programming language[10]

[8] https://preshing.com/20120625/memory-ordering-at-compile-time/

[9] https://www.kernel.org/doc/html/latest/core-api/wrappers/memory-barriers.html

[10] Hans-J. Boehm, Threads cannot be implemented as a library

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

To catch and fix bugs, you will use:

- compiler barriers[8]
- memory orderings (volatile, seq-cst, acquire-release...)[9]
- concurrency primitives and their integration with programming language[10]
- native debuggers and time-travel debugging[11]

[8] https://preshing.com/20120625/memory-ordering-at-compile-time/

[9] https://www.kernel.org/doc/html/latest/core-api/wrappers/memory-barriers.html

[10] Hans-J. Boehm, Threads cannot be implemented as a library

[11] https://rr-project.org/

How deep is the rabbit hole?

«Obvious» transformations of single-threaded programs break concurrent code.

To catch and fix bugs, you will use:

- compiler barriers[8]
- memory orderings (volatile, seq-cst, acquire-release...)[9]
- concurrency primitives and their integration with programming language[10]
- native debuggers and time-travel debugging[11]
- specialized tools such as model checking, deterministic scheduling ...

[8] https://preshing.com/20120625/memory-ordering-at-compile-time/

[9] https://www.kernel.org/doc/html/latest/core-api/wrappers/memory-barriers.html

[10] Hans-J. Boehm, Threads cannot be implemented as a library

[11] https://rr-project.org/

We are here

CPU is your enemy

Compilers (software) could break your concurrent software by reordering code.

CPU is your enemy

Compilers (software) could break your concurrent software by reordering code. CPU and memory subsystem (hardware) also love to reorder operations.

CPU is your enemy

Compilers (software) could break your concurrent software by reordering code. CPU and memory subsystem (hardware) also love to reorder operations.

- Execute independent instructions simultaneously (out-of-order execution)

CPU is your enemy

Compilers (software) could break your concurrent software by reordering code. CPU and memory subsystem (hardware) also love to reorder operations.

- Execute independent instructions simultaneously (out-of-order execution)
- Use same CPU resources for execution of logically independent threads (hyper-threading)

CPU is your enemy

Compilers (software) could break your concurrent software by reordering code. CPU and memory subsystem (hardware) also love to reorder operations.

- Execute independent instructions simultaneously (out-of-order execution)
- Use same CPU resources for execution of logically independent threads (hyper-threading)
- Speculate[12,13]
  - branch prediction
  - cache prefetching
  - speculative execution
  - ...

[12] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)
[13] https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)

x86: Store buffering

```
int x, y;

void threadA() {
      x = 1;
  int a = y;
}

void threadB() {
      y = 1;
  int b = x;
}
```

x86: Store buffering

```
int x, y;
```

```
void threadA() {
      x = 1;
  int a = y;
}
```

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
void threadB() {
      y = 1;
   int b = x;
}
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

x86: Store buffering

```
# thread A
mov [x] ,  1   # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1   # (B.1)
mov EBX, [x] # (B.2)
```

x86: Store buffering

```
# thread A
mov [x] ,  1   # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1   # (B.1)
mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

Execution traces:

- A.1 -> A.2 -> B.1 -> B.2
- B.1 -> A.2 -> B.2
- B.2 -> A.2
- B.1 -> A.1 -> A.2 -> B.2
- B.2 -> A.2
- B.2 -> A.1 -> A.2

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

(1 1) , (0 1) , (1 0) , (0 0)

Execution traces:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Answer: (1 1) , (0 1) , (1 0)

Execution traces:

- A.1 -> A.2 -> B.1 -> B.2 : (0, 1)
- B.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.1 -> A.1 -> A.2 -> B.2 : (1, 1)
- B.2 -> A.2 : (1, 1)
- B.2 -> A.1 -> A.2 : (1, 0)

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Answer: (1 1) , (0 1) , (1 0)

x86: Store buffering

```
# thread A                    # thread B
mov [x] ,  1   # (A.1)        mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Right answer: (1 1) , (0 1) , (1 0) , (0 0)

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Right answer: (1 1) , (0 1) , (1 0) , (0 0)

Processor could reorder loads and stores.

x86: Store buffering

```
# thread A
mov [x] , 1  # (A.1)
mov EAX , [y] # (A.2)
```

```
# thread B
mov [y] , 1  # (B.1)
mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Right answer: (1 1) , (0 1) , (1 0) , (0 0)

Processor could reorder loads and stores.

Conclusion: instruction order in machine code $\neq$ order of observable effects of these instructions.

x86: Store buffering

```
# thread A                    # thread B
mov [x] , 1   # (A.1)         mov [y] , 1   # (B.1)
mov EAX , [y] # (A.2)         mov EBX, [x] # (B.2)
```

What could be in (EAX EBX)?

Right answer: (1 1) , (0 1) , (1 0) , (0 0)

Processor could reorder loads and stores.

Conclusion: instruction order in machine code $\neq$ order of these instructions.

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

- x86 or x86_64 (TSO): no

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[14]

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[14]

Memory writes could reach other CPUs in different order.

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[14]

Memory writes could reach other CPUs in different order.
Each CPU has it's own timeline and independent «view» of neighbours.
These «views» could be different for different cores.

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1

arm64: Independent Reads of Independent Writes

```
thread1     thread2     thread3     thread4
  x = 1       y = 1       r1 = x      r3 = y
                          r2 = y      r4 = x
```

Is it possible to see `(r1 = 1, r2 = 0, r3 = 1, r4 = 0)`?
Assume reordering of reads cannot happen.

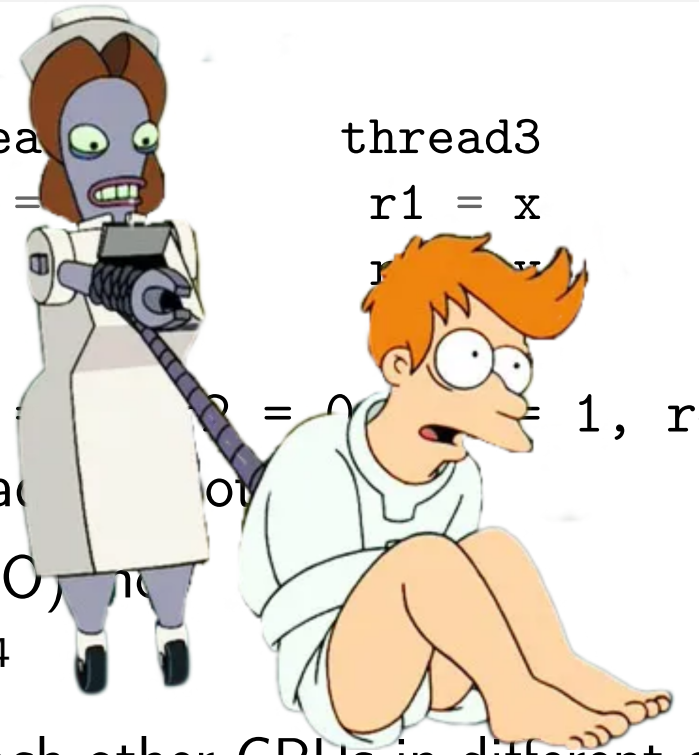
- `x86` or `x86_64` (TSO): no
- `ARM` or `POWER`: yes[14]

Memory writes could reach other CPUs in different order.
Each CPU has it's own timeline and independent «view» of neighbours.
These «views» could be different for different cores.
Conclusion: event «memory write» is not a point on universal timeline[15].

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1
[15] The Art of Multiprocessor Programming by Maurice Herlihy & Nir havit, Chapter 3 "Concurrent Objects"

arm64: Independent Reads of Independent Writes

```
thread1        threa          thread3        thread4
  x = 1          y =           r1 = x         r3 = y
                               r              r4 = x
```



Is it possible to see (r1 = 1, r2 = 0, r3 = 1, r4 = 0)?
Assume reordering of rea ot

- x86 or x86_64 (TSO) n
- ARM or POWER: yes[14]

Memory writes could reach other CPUs in different order.
Each CPU has it's own timeline and independent «view» of neighbours.
These «views» could be different for different cores.
Conclusion: event «memory write» is not a point on universal timeline[15].

[14] A Tutorial Introduction to the ARM and POWER Relaxed Memory Models, section 6.1
[15] The Art of Multiprocessor Programming by Maurice Herlihy & Nir havit, Chapter 3 "Concurrent Objects"

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions

Hardware behaviour is complicated

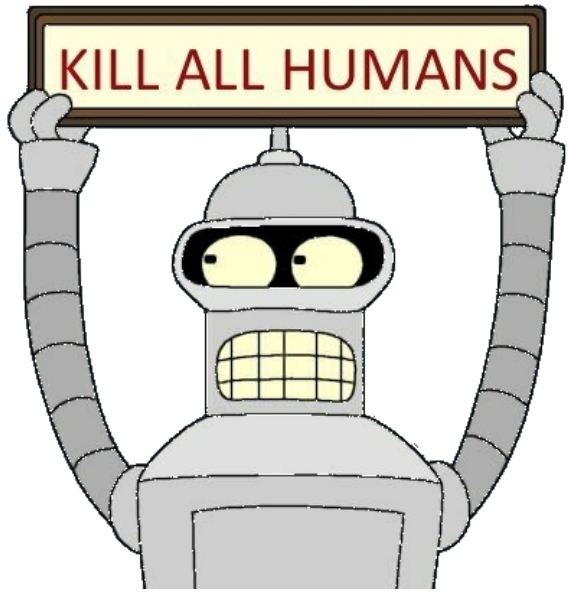
- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules



Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules

Why would anybody use ARM/POWER/RISC-V or other CPUs with weak memory models?

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules

Why would anybody use ARM/POWER/RISC-V or other CPUs with weak memory models?

- performance

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules

Why would anybody use ARM/POWER/RISC-V or other CPUs with weak memory models?

- performance
- performance!

Hardware behaviour is complicated

- instruction order in machine code $\neq$ order of observable effects of these instructions
- event «memory write» is not a point on universal timeline
- every processor has it's own rules

Why would anybody use ARM/POWER/RISC-V or other CPUs with weak memory models?

- performance
- performance!
- energy efficiency :)

We are here

Holy war warning

- All languages are good and useful!

Holy war warning

- All languages are good and useful!
- *Some* features of *some* languages could be helpful for *some* tasks.

Holy war warning

- All languages are good and useful!
- *Some* features of *some* languages could be helpful for *some* tasks.
- Do not be very serious about following comparison.

Swift: race to the crash

Swift[16]

> *Concurrent write/write or read/write access to the same location in memory generally remains undefined/illegal behavior, unless all such access is done through a special set of primitive atomic operations.*

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md

Swift: race to the crash

Swift[16]

> *Concurrent write/write ~~or read/write~~ access to the same location ~~in memory generally~~ remains ~~undefined/~~illegal behavior, unless ~~all such access~~ is done through ~~a special set of primitive~~ atomic operations.*

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md

Swift: race to the crash

Swift[16]

> *Concurrent write/write access to the same location remains illegal behavior, unless is done through atomic operations.*

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md

Swift: race to the crash

Swift[16]

> *Concurrent write/write access to the same location remains illegal behavior, unless is done through atomic operations.*

```
import Foundation
class Bird {}
var S = Bird()
let q = DispatchQueue.global(qos: .default)
q.async { while(true) { S = Bird() } }
while(true) { S = Bird() }
```

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md

Swift: race to the crash

Swift[16]

> *Concurrent write/write access to the same location remains illegal behavior, unless is done through atomic operations.*

```
import Foundation
class Bird {}
var S = Bird()
let q = DispatchQueue.global(qos: .default)
q.async { while(true) { S = Bird() } }
while(true) { S = Bird() }
```

Application crashes with `double free or corruption`.

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md

Swift: race to the crash

Swift[16]

> *Concurrent write/write access to the same location remains illegal behavior, unless is done through atomic operations.*

```
import Foundation
class Bird {}
var S = Bird()
let q = DispatchQueue.global(qos: .default)
q.async { while(true) { S = Bird() } }
while(true) { S = Bird() }
```

Application crashes with `double free or corruption`.
Why? You could guess by yourself[17] or see the answer[18].

[16] https://github.com/apple/swift-evolution/blob/main/proposals/0282-atomics.md
[17] https://tonygoold.github.io/arcempire/
[18] https://github.com/apple/swift/blob/main/docs/proposals/Concurrency.rst

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline[19].

[19] https://en.wikipedia.org/wiki/Consistency_model#Strict_consistency

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.

```
void thread1() {          |     void thread2() {
                          |
      foo()               |           baz()
                          |
                          |
      bar()               |           foo()
                          |
}                         |     }
```

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.

```
void thread1() {          |     void thread2() {
       lock()             |            lock()
      foo()               |           baz()
       unlock()           |            unlock()
       lock()             |            lock()
      bar()               |           foo()
       unlock()           |            unlock()
}                         |     }
```

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

```
static GlobalInterpreterLock GIL = ...;
void thread1() {          |     void thread2() {
    GIL.lock()            |         GIL.lock()
        foo()             |             baz()
    GIL.unlock()          |         GIL.unlock()
    GIL.lock()            |         GIL.lock()
        bar()             |             foo()
    GIL.unlock()          |         GIL.unlock()
}                         |     }
```

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

- Python have threads[20] but their inefficiency is a "special feature" of CPython interpreter.

[20] https://docs.python.org/3/library/threading.html

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

- Python have threads[20] but their inefficiency is a "special feature" of CPython interpreter.
- PyPy still have GIL[21].

[20] https://docs.python.org/3/library/threading.html
[21] https://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

- Python have threads[20] but their inefficiency is a "special feature" of CPython interpreter.
- PyPy still have GIL[21].
- There were (unsuccessful **yet**) attempts to redesign memory model of Python[22].

[20] https://docs.python.org/3/library/threading.html
[21] https://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why
[22] https://peps.python.org/pep-0583/

Python: VIP mutex

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline. Every operation is guarded by global lock.

- Python have threads[20] but their inefficiency is a "special feature" of CPython interpreter.
- PyPy still have GIL[21].
- There were (unsuccessful **yet**) attempts to redesign memory model of Python[22].
- Naive approach for removing GIL will slow-down the language by 10-30% and will break interop with many native libraries[23].

[20] https://docs.python.org/3/library/threading.html
[21] https://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why
[22] https://peps.python.org/pep-0583/
[23] https://peps.python.org/pep-0703/

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events.

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events. Event loop.

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events. Event loop.

- Users would like to use all CPU cores of their systems.

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events. Event loop.

- Users would like to use all CPU cores of their systems.
- It is possible to start additional independent agents (web workers) and use message passing[24].

[24] https://www.w3schools.com/html/html5_webworkers.asp

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events. Event loop.

- Users would like to use all CPU cores of their systems.
- It is possible to start additional independent agents (web workers) and use message passing[24].
- It is possible to share the same byte array among several workers[25].

[24] https://www.w3schools.com/html/html5_webworkers.asp
[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

JavaScript: say no to threading

The simplest way to think about memory operations: all events happen atomically and have a total order on a single timeline.
There are no threads in a language.
The only one thread processes all events. Any event could trigger other, possibly delayed, events. Event loop.

- Users would like to use all CPU cores of their systems.
- It is possible to start additional independent agents (web workers) and use message passing[24].
- It is possible to share the same byte array among several workers[25].
- Data race **could** happen and need to be specified in the doc[26].

[24] https://www.w3schools.com/html/html5_webworkers.asp
[25] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
[26] "Repairing and Mechanising the JavaScript Relaxed Memory Model" https://arxiv.org/abs/2005.10554

Java: computer science!

Java: computer science!

Language designers use advanced mathematics:

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial order, linear order; transitive closure of binary relation; happens-before

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial order, linear order; transitive closure of binary relation; happens-before
- Causality requirements, circular hp, out-of-thin-air problem

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial order, linear order; transitive closure of binary relation; happens-before
- Causality requirements, circular hp, out-of-thin-air problem
- Adaptation to h/w models[27]

[27] "JSR-133 Cookbook for Compiler Writers" https://gee.cs.oswego.edu/dl/jmm/cookbook.html

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial order, linear order; transitive closure of binary relation; happens-before
- Causality requirements, circular hp, out-of-thin-air problem
- Adaptation to h/w models[27]
- Every data race is well-defined: allowed and forbidden outcomes

[27] "JSR-133 Cookbook for Compiler Writers" `https://gee.cs.oswego.edu/dl/jmm/cookbook.html`

Java: computer science!

Language designers use advanced mathematics:

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial order, linear order; transitive closure of binary relation; happens-before
- Causality requirements, circular hp, out-of-thin-air problem
- Adaptation to h/w models[27]
- Every data race is well-defined: allowed and forbidden outcomes

This approach has drawbacks:

- Very complicated description, takes a lot of time to design and very expensive to maintain.
- There still will be some «grey zones»[28].
- Few people in the world fully understand the specification. Even less people able to use it for productive development.

[27] "JSR-133 Cookbook for Compiler Writers" https://gee.cs.oswego.edu/dl/jmm/cookbook.html

[28] "Java Memory Model Examples: Good, Bad and Ugly" https://groups.inf.ed.ac.uk/request/jmmexamples.pdf

Use patterns, Luke

Doug Lea, private communication with Aleksey Shipilev, 2013[29]

> *The best way to build up a small repertoire of constructions that you know the answers for and then never think about the JMM rules again unless you are forced to do so! Literally nobody likes figuring things out from the JMM rules as stated, or can even routinely do so correctly. This is one of the many reasons we need to overhaul JMM someday.*

[29] Citation from `https://shipilev.net/blog/2014/jmm-pragmatics`, slide 109

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...and «slow» ones

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...and «slow» ones
- Highly concurrent...

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...and «slow» ones
- Highly concurrent...and focused on single-thread model

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...and «slow» ones
- Highly concurrent...and focused on single-thread model

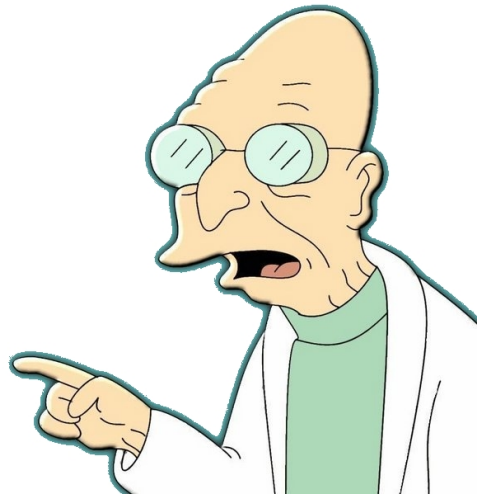
Always use appropriate language for your problem.

Concurrency + Programming Language = ?

There are many flavours of programming languages:

- Simple to write code...and more involved ones
- Performance-oriented...and «slow» ones
- Highly concurrent...and focused on single-thread model

Always use appropriate language for your problem.



We are here

How could I improve my skills in concurrency?

How could I improve my skills in concurrency?

There is no easy way.

How could I improve my skills in concurrency?

There is no easy way.

- Java Concurrency in Practice – 403 pages
- The Art of Multiprocessor Programming – 508 pages
- Is Parallel Programming Hard, And, If So, What Can You Do About It? – 634 pages

How could I improve my skills in concurrency?

There is no easy way.

- Java Concurrency in Practice – 403 pages
- The Art of Multiprocessor Programming – 508 pages
- Is Parallel Programming Hard, And, If So, What Can You Do About It? – 634 pages

Russ Cox, Go programming language tech lead at Google, author of Go memory model:[30]

> *Twenty-five years after the first Java memory model, and after many person-centuries of research effort, we may be starting to be able to formalize entire memory models. Perhaps, one day, we will also fully understand them.*

[30] https://research.swtch.com/plmm

Conclusion

I tried to share my fears and ideas related to concurrent programming.

Conclusion

I tried to share my fears and ideas related to concurrent programming.
There are many other fun things related to this topic:

- OS scheduler and user-space schedulers

Conclusion

I tried to share my fears and ideas related to concurrent programming.
There are many other fun things related to this topic:

- OS scheduler and user-space schedulers
- Different kinds of weak memory models

Conclusion

I tried to share my fears and ideas related to concurrent programming.
There are many other fun things related to this topic:

- OS scheduler and user-space schedulers
- Different kinds of weak memory models
- Fundamental algorithms for concurrency, their properties and unsolved problems

Conclusion

I tried to share my fears and ideas related to concurrent programming.
There are many other fun things related to this topic:

- OS scheduler and user-space schedulers
- Different kinds of weak memory models
- Fundamental algorithms for concurrency, their properties and unsolved problems

Understanding this stuff could *slow-down* your development :)

Conclusion

I tried to share my fears and ideas related to concurrent programming.
There are many other fun things related to this topic:

- OS scheduler and user-space schedulers
- Different kinds of weak memory models
- Fundamental algorithms for concurrency, their properties and unsolved problems

Understanding this stuff could *slow-down* your development :)
Remember, VM engineers could have *strange* ideas.

Thank you!

Reading

Books

- "The Art of Multiprocessor Programming" by M. Herlihy & N. Shavit
- "Is Parallel Programming Hard, And, If So, What Can You Do About It?" by Paul E. McKenney
- "Java Concurrency in Practice" by Brian Goetz et al.

Articles

- "Memory Models" series by Russ Cox[31]
- "Threads Cannot be Implemented as a Library" by Hans-J. Boehm
- "A Tutorial Introduction to the ARM and POWER Relaxed Memory Models" by L. Maranget et al.
- "Memory Barriers: a Hardware View for Software Hackers" by Paul E. McKenney

[31] https://research.swtch.com/mm

Videos

- Aleksey Shipilev, JMM series `https://shipilev.net`
- Herb Sutter, C++ and Beyond 2012, "Atomic Weapons" series `https://youtu.be/A8eCGOqgvH4`