Lecture 10: language memory model

compiler optimizations, compiler barriers, language memory model, strict consistency, threads cannot be implemented as a library, visibility, volatile

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l10.pdf

In previous episodes: homework

"Is Parallel Programming Hard, And, If So, What Can You Do About It" (perfbook) Appendix B "Why Memory Barriers?"

- section B.1 "Cache Structure". Be ready to draw and explain what is associative hardware cache.
- section B.2.4 "MESI Protocol Example"

In previous episodes

Abstractions and algorithms (high level)

- Race conditions, deadlocks, missed signals, ABA problem
- Events, Precedence, Consistency
- Progress conditions: wait-free, lock-free, obstruction-free, starvation-free, deadlock-free
- Read-Modify-Write operations

Programmers need at least: **Linearizability** (composability of different modules)
Hardware (low level)

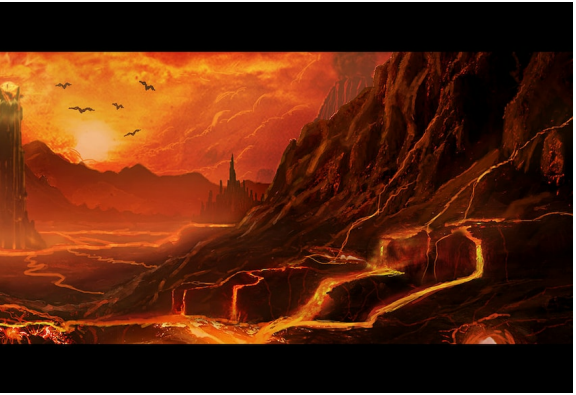
- Cache coherency protocol: states, transitions, message passing
- Store buffering, Load buffering, Invalidate Queues, Interconnect topology
- Reorderings of memory operations on independent memory locations
- Weak (relaxed) consistency

Hardware provides at most: **Coherence** (linear order of writes for particular location)

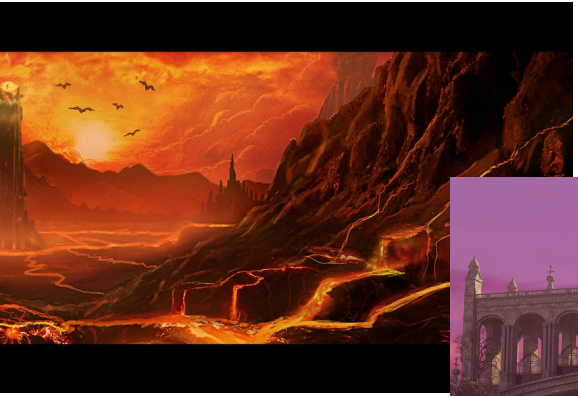
The Gap

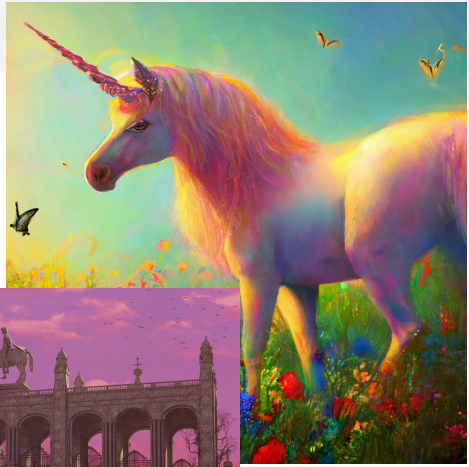
The Gap

The Gap



The Gap





Bridging the gap

High-level concurrent programming focuses on design-level problems:

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

- scalability under contention

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

- scalability under contention
- effective usage of scheduling quantum

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

- scalability under contention
- effective usage of scheduling quantum
- avoiding false sharing, minimizing true sharing

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

- scalability under contention
- effective usage of scheduling quantum
- avoiding false sharing, minimizing true sharing
- replication-friendly data structures

Bridging the gap

High-level concurrent programming focuses on design-level problems:

- separating function/computation and carrier/executor (`Future`, `Executor`)
- providing contracts and interfaces rather than particular implementations
- composability of concurrent modules (deadlock avoidance, restartable transactions)
- avoiding logic errors (race condition, unsynchronized data access, resource leaks)
- user-visible metrics (responsiveness, CPU utilization, memory consumption)
- error recovery, logging, graceful degradation

Low-level concurrent programming focuses on performance and location-level consistency:

- scalability under contention
- effective usage of scheduling quantum
- avoiding false sharing, minimizing true sharing
- replication-friendly data structures
- appropriate and lightweight memory barriers

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language.

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)
- allow to write hardware-agnostic and high-performant concurrent programs

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)
- allow to write hardware-agnostic and high-performant concurrent programs (bugs?)

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)
- allow to write hardware-agnostic and high-performant concurrent programs (bugs?)
- were not designed for modern concurrency

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)
- allow to write hardware-agnostic and high-performant concurrent programs (bugs?)
- were not designed for modern concurrency (single-threaded?)

Bridging the gap

- High level concurrent programming focuses on design-level problems
- Low level concurrent programming focuses on performance and location-level consistency
- Any programming language need to balance between low-level and high-level

There is no *perfect* programming language. But some languages

- avoid most of concurrency pitfalls via source-level constraints (expressive power?)
- simplify writing safe concurrent programs (performance?)
- allow to write hardware-agnostic and high-performant concurrent programs (bugs?)
- were not designed for modern concurrency (single-threaded?)

As a professional, you should be able to use any tool, not only "the favourite" one.

Supplementary materials

Unconditional benefit

- "Threads Cannot be Implemented as a Library" by Hans-J. Boehm[1]
  https://courses.cs.washington.edu/courses/cse590p/05au/HPL-2004-209.pdf
- "Memory Models" series by Russ Cox https://research.swtch.com/mm
- Herb Sutter, C++ and Beyond 2012, "Atomic Weapons" series
  https://youtu.be/A8eCGOqgvH4
- Роман Елизаров, "Многопоточное программирование — теория и практика"
  https://youtu.be/mf4lC6TpclM

Advanced material

- "Using JDK 9 Memory Order Modes" by Doug Lea
  https://gee.cs.oswego.edu/dl/html/j9mm.html
- Aleksey Shipilëv, JMM series https://shipilev.net

[1] https://www.hboehm.info/misc_slides/pldi05_threads.pdf

Lecture plan

1. Compiler optimizations
2. Barriers: kinds and flavours
3. Language memory models
   - Goals and non-goals
   - Partial memory model: bad things can not be expressed
   - Partial memory model: bad things should be avoided
   - Strict consistency: GIL
   - Strict consistency: Event Loop
   - Partial memory model: threading as a service
   - Partial memory model: concurrency-aware language subset
   - Memory model: concurrency embedded into language
4. Advanced topics
   - Visibility
   - Volatile
5. Summary

Lecture plan

1. Compiler optimizations
2. Barriers: kinds and flavours
3. Language memory models
   - Goals and non-goals
   - Partial memory model: bad things can not be expressed
   - Partial memory model: bad things should be avoided
   - Strict consistency: GIL
   - Strict consistency: Event Loop
   - Partial memory model: threading as a service
   - Partial memory model: concurrency-aware language subset
   - Memory model: concurrency embedded into language
4. Advanced topics
   - Visibility
   - Volatile
5. Summary

Inventing reads

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

Inventing reads

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

Could compiler use less registers for intermediate operations?

Inventing reads

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

```
static int a;
void foo_2() {
  while (true) {

    if (a == 0) break;
    do_something_with(a);
  }
}
```

Could compiler use less registers for intermediate operations?

Removing reads

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

Removing reads

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

Could compiler avoid repated memory loads?

Removing reads

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

```
static int a;
void foo_2() {
  int tmp = a;
  if (tmp != 0)
    while (true) {
      do_something_with(tmp);
    }
}
```

Could compiler avoid repated memory loads?

Removing reads

```
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[2]
x86-64 gcc 13.1 -O2[3]

```
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
```

[2] https://godbolt.org/z/99j3erzaE
[3] https://godbolt.org/z/fxzGEo1qf

Load hoisting

```
static int a;
void foo_1(bool c, int r1) {
  if (c) {
    r1 = a;
  }
}
```

```
static int a;
void foo_2(bool c, int r1) {
  int t = a;
  r1 = c ? t : r1;
}
```

CSE over lock

```
static int a;
void foo_1(bool c, int r1, int r2) {
   r1 = a;
   lock();
   r2 = a;
}
```

```
static int a;
void foo_2(bool c, int r1, int r2) {
   r1 = a;
   lock();
   r2 = r1;
}
```

Load hoisting + CSE over lock

```
static int a;
void foo_1() {
   if (c) {
     r1 = a;
   }
   lock();
   r2 = a;
}
```

```
static int a;
void foo_2() {
   int t = a;
   r1 = c ? t : r1;
   lock();
   r2 = a;
}
```

```
static int a;
void foo_3() {
  int t = a;
  r1 = c ? t : r1;
  lock();
  r2 = t;
}
```

Load hoisting + CSE over lock

```
static int a;
void foo_1() {
   if (c) {
     r1 = a;
   }
   lock();
   r2 = a;
}
```

```
static int a;
void foo_2() {
   int t = a;
   r1 = c ? t : r1;
   lock();
   r2 = a;
}
```

```
static int a;
void foo_3() {
  int t = a;
  r1 = c ? t : r1;
  lock();
  r2 = t;
}
```

When `c == false`, **a** is moved out of the critical region![4]

[4] https://people.mpi-sws.org/~viktor/slides/2017-09-concur.pdf

Compiler: friend or foe?

Various optimizations of plain memory accesses

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes
- reorder memory operations

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes
- reorder memory operations

make concurrent reasoning almost impossible

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes
- reorder memory operations

make concurrent reasoning almost impossible

- no **linearizability**

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes
- reorder memory operations

make concurrent reasoning almost impossible

- no **linearizability**
- no **coherence**

Compiler: friend or foe?

Various optimizations of plain memory accesses

- invent reads or writes
- remove reads or writes
- reorder memory operations

make concurrent reasoning almost impossible

- no **linearizability**
- no **coherence**
- no **eventual visibility** (a.k.a. **progress**)

Taming compiler optimizations

Compiler reorders language-level constructs

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

- do not reorder **this** and **that** operation (low-level consistency)

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

- do not reorder **this** and **that** operation (low-level consistency)
- do not move operations out of critical sections (high-level consistency)

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

- do not reorder **this** and **that** operation (low-level consistency)
- do not move operations out of critical sections (high-level consistency)
- do not invent new operations

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

- do not reorder **this** and **that** operation (low-level consistency)
- do not move operations out of critical sections (high-level consistency)
- do not invent new operations
- do not remove or merge some operations

Taming compiler optimizations

Compiler reorders language-level constructs

- to improve performance
- but keep consistent single-threaded behaviour

We need to inform compiler that some invariants are important for multi-threaded execution:

- do not reorder **this** and **that** operation (low-level consistency)
- do not move operations out of critical sections (high-level consistency)
- do not invent new operations
- do not remove or merge some operations

We have already encountered such problems with reorderings of memory effects, remember?

Lecture plan

1. Compiler optimizations
2. Barriers: kinds and flavours
3. Language memory models
   - Goals and non-goals
   - Partial memory model: bad things can not be expressed
   - Partial memory model: bad things should be avoided
   - Strict consistency: GIL
   - Strict consistency: Event Loop
   - Partial memory model: threading as a service
   - Partial memory model: concurrency-aware language subset
   - Memory model: concurrency embedded into language
4. Advanced topics
   - Visibility
   - Volatile
5. Summary

Barriers taxonomy: here we go again

Simplified taxonomy of barriers[5]:

- `Store_Store`, `Store_Load`, `Load_Store`, `Load_Load`

```
int x = static.data1;
Store_Store();
Store_Load();
int y = static.data2;
static.data3 = 17;
```

```
int x = static.data1;
Load_Load();

int y = static.data2;
static.data3 = 17;
```

[5] Could be misleading https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Compiler barriers

```
int x, y;
void foo1() {
        x = 1;
        y = 2;
        x = 3;
}
```

```
int x, y;
void foo2() {
        x = 1;
        barrier();
        y = 2;
        x = 3;
}
```

Compiler barriers

```
int x, y;
void foo1() {
      x = 1;
      y = 2;
      x = 3;
}
```

```
foo1:
    mov [y], 2
    mov [x], 3
    ret
```

```
int x, y;
void foo2() {
      x = 1;
      barrier();
      y = 2;
      x = 3;
}
```

```
foo2:
    mov [x], 1
    mov [y], 2
    mov [x], 3
    ret
```

Compiler barriers

```
int x, y;
void foo1() {
      x = 1;
      y = 2;
      x = 3;
}
```

```
foo1:
    mov [y], 2
    mov [x], 3
    ret
```

```
int x, y;
void foo2() {
      x = 1;
      barrier();
      y = 2;
      x = 3;
}
```

```
foo2:
    mov [x], 1
    mov [y], 2
    mov [x], 3
    ret
```

Warning: avoid such low-level tricks in modern programming languages![6]

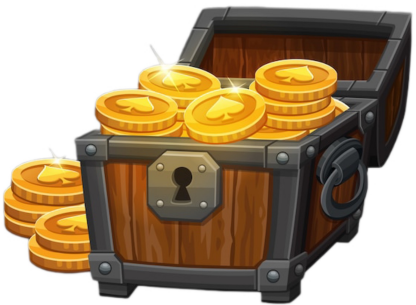
[6] https://preshing.com/20120625/memory-ordering-at-compile-time/

Question time

Question:

- `mov [x], 1`
- `mov [y], 2`
- `mov [x], 3`

Name hardware-level optimizations that will "reorder" these memory operations.



Reorderings: everywhere

Hardware

Reorderings: everywhere

Hardware

- anything could be reordered with everything

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

- anything could be reordered with everything

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

- anything could be reordered with everything
- optimizations do not violate single-threaded behaviour

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

- anything could be reordered with everything
- optimizations do not violate single-threaded behaviour
- some language constructs are **special** (e.g. `synchronized`)

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

- anything could be reordered with everything
- optimizations do not violate single-threaded behaviour
- some language constructs are **special** (e.g. `synchronized`)
- ordering could be enforced by compiler barriers

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

- anything could be reordered with everything
- optimizations do not violate single-threaded behaviour
- some language constructs are **special** (e.g. `synchronized`)
- ordering could be enforced by compiler barriers

Looks like problem is already solved!

Reorderings: everywhere

Hardware

- anything could be reordered with everything
- current CPU will "emulate" execution of single-threaded program "as if" in program order
- single memory cell is **coherent**
- ordering could be enforced by memory barriers

Compiler

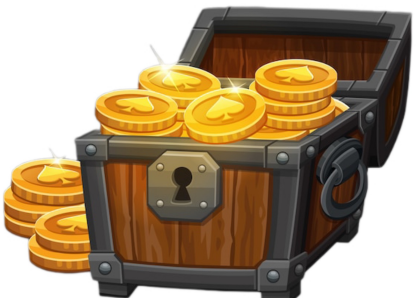
- anything could be reordered with everything
- optimizations do not violate single-threaded behaviour
- some language constructs are **special** (e.g. `synchronized`)
- ordering could be enforced by compiler barriers

Looks like problem is already solved!

Reasoning about memory locations/compiler optimizations is mundane, complex and fragile, shouldn't we use something better?

Question time

Question: Barriers enforce ordering of operations. But compiler could "invent" or "remove" operations which will break some concurrent invariants. What should we do?



Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)
- Language version (maybe)

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)
- Language version (maybe)

Programming language should be low-level enough and **should** provide

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)
- Language version (maybe)

Programming language should be low-level enough and **should** provide

- Consistency guarantees for explicitly synchronized memory operations

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)
- Language version (maybe)

Programming language should be low-level enough and **should** provide

- Consistency guarantees for explicitly synchronized memory operations
- Reasonable rules of concurrent execution

Consistency of memory operations in concurrent environment: challenges

Programming language should be high-level enough and **should not** depend on

- Platform (different OSes, various CPU architectures)
- Optimizer (compilers)
- Runtime (memory management, RTTI, synchronization)
- Language version (maybe)

Programming language should be low-level enough and **should** provide

- Consistency guarantees for explicitly synchronized memory operations
- Reasonable rules of concurrent execution

We need language memory model

Lecture plan

Language Memory Model

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?
- How to check consistency of memory model itself?

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?
- How to check consistency of memory model itself?
- How to guarantee that every program could be unambiguously described?

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?
- How to check consistency of memory model itself?
- How to guarantee that every program could be unambiguously described?
- How to explain the rules to language users?

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?
- How to check consistency of memory model itself?
- How to guarantee that every program could be unambiguously described?
- How to explain the rules to language users?

There is no *perfect* programming language.

Language Memory Model

- How to formalize it?
  - Documentation in human language
  - Machine-readable format
  - Executable algorithm
  - Set of optimizer rules
- Prefer strict rules or use weak models?
- How to check consistency of memory model itself?
- How to guarantee that every program could be unambiguously described?
- How to explain the rules to language users?

There is no *perfect* programming language.
Any language tries to be *performant*, *safe*, *user-friendly*, *stable* ...

Lecture plan

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

SQL

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

SQL , Clojure

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

SQL , Clojure , Haskell

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

SQL , Clojure , Haskell

> *All told, a monad in X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor.*

No man, no problem

Concurrency is complicated only when we have non-trivial communication

- via shared memory locations
- via concurrent primitives

Use plain and simple approaches

- immutable data structures
- declarative description of computation

SQL , Clojure , Haskell

> *All told, a monad in X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor.*

Is it about *language* or *programming style*?

Immutability

Immutable data structure

Immutability

Immutable data structure

- different threads could simultaneously read

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

How to update such data structure when something happens?

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

How to update such data structure when something happens?

- Create new immutable instance with up-to-date information

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

How to update such data structure when something happens?

- Create new immutable instance with up-to-date information

Few problems

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

How to update such data structure when something happens?

- Create new immutable instance with up-to-date information

Few problems

- Overheads for creation of large data structures

Immutability

Immutable data structure

- different threads could simultaneously read
- conflicts (data races) are impossible

How to update such data structure when something happens?

- Create new immutable instance with up-to-date information

Few problems

- Overheads for creation of large data structures
- "Publish" data structure for all threads == write to shared memory location

Declarative DSL

java.util.stream[7]

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

[7] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

Declarative DSL

java.util.stream[7]

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

Solve any embarrassingly parallel problem in few readable lines.

[7] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

Declarative DSL

java.util.stream[7]

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

Solve any embarrassingly parallel problem in few readable lines.
All safety and efficiency will happen under the hood.

[7] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

Declarative DSL

Solve any embarrassingly parallel problem in few readable lines.

Declarative DSL

Solve any embarrassingly parallel problem in few readable lines.

- Java parallel streams https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- OpenMP https://www.openmp.org/
- Intel TBB https://github.com/oneapi-src/oneTBB
- MPI https://www.open-mpi.org/
- MapReduce https://research.google/pubs/pub62/
- Resilient Distributed Datasets https://dl.acm.org/doi/10.5555/2228298.2228301

Declarative DSL

Solve any embarrassingly parallel problem in few readable lines.

- Java parallel streams https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- OpenMP https://www.openmp.org/
- Intel TBB https://github.com/oneapi-src/oneTBB
- MPI https://www.open-mpi.org/
- MapReduce https://research.google/pubs/pub62/
- Resilient Distributed Datasets https://dl.acm.org/doi/10.5555/2228298.2228301

First and obvious choice for any language and any application.

Declarative DSL

Solve any embarrassingly parallel problem in few readable lines.

- Java parallel streams https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- OpenMP https://www.openmp.org/
- Intel TBB https://github.com/oneapi-src/oneTBB
- MPI https://www.open-mpi.org/
- MapReduce https://research.google/pubs/pub62/
- Resilient Distributed Datasets https://dl.acm.org/doi/10.5555/2228298.2228301

First and obvious choice for any language and any application. Unless it does not fit.

Declarative DSL

Solve any embarrassingly parallel problem in few readable lines.

- Java parallel streams https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
- OpenMP https://www.openmp.org/
- Intel TBB https://github.com/oneapi-src/oneTBB
- MPI https://www.open-mpi.org/
- MapReduce https://research.google/pubs/pub62/
- Resilient Distributed Datasets https://dl.acm.org/doi/10.5555/2228298.2228301

First and obvious choice for any language and any application. Unless it does not fit.
Implementations of such DSL are concurrently writing to shared memory locations.

Actor model

Programming model with no shared state[8].

[8] https://en.wikipedia.org/wiki/Actor_model

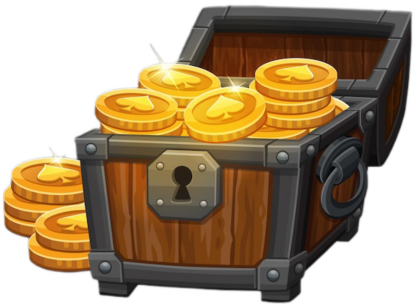
Actor model

Programming model with no shared state[8].
All computational agents (lightweight processes) are independent and communicate via message passing.

[8] https://en.wikipedia.org/wiki/Actor_model

Question time

Question: Name your favourite concurrent message-passing protocol



Actor model

Programming model with no shared state.
All computational agents (lightweight processes) are independent and comunicate via message passing.

Actor model

Programming model with no shared state.
All computational agents (lightweight processes) are independent and comunicate via message passing.

- Erlang[9]
- Akka actors[10]

[9] https://www.erlang.org/
[10] https://doc.akka.io/docs/akka/current/typed/actors.html#akka-actors

Actor model

Programming model with no shared state.
All computational agents (lightweight processes) are independent and comunicate via message passing.

- Erlang[9]
- Akka actors[10]

Implementations of such systems are concurrently writing to shared memory locations.

[9] https://www.erlang.org/
[10] https://doc.akka.io/docs/akka/current/typed/actors.html#akka-actors

Lecture plan

Be in Sync or Die Tryin

Swift[11]

> *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.*

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

Be in Sync or Die Tryin

Swift[11]

> *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.*

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

Be in Sync or Die Tryin

Swift[11]

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

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

Be in Sync or Die Tryin

Swift[11]

> *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() }
```

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

Be in Sync or Die Tryin

Swift[11]

> *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() }
```

Concurrent writes are violating naive implementation of automatic reference counting[12,13]. Program crashes with `double free or corruption`.

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

Lecture plan

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order[14].

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

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.

```
void thread1() {

  foo();


  bar();

}
```

```
void thread2() {

  baz();


  foo();

}
```

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.

```
void thread1() {
  lock();
  foo();
  unlock();
  lock();
  bar();
  unlock();
}
```

```
void thread2() {
  lock();
  baz();
  unlock();
  lock();
  foo();
  unlock();
}
```

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.

```
void thread1() {
  GIL.lock();
  foo();
  GIL.unlock();
  GIL.lock();
  bar();
  GIL.unlock();
}
```

```
void thread2() {
  GIL.lock();
  baz();
  GIL.unlock();
  GIL.lock();
  foo();
  GIL.unlock();
}
```

Global mutex guards every "instruction"

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.
Global Interpreter Lock (GIL)[15] guards every "instruction"

[15] https://en.wikipedia.org/wiki/Global_interpreter_lock

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.
Global Interpreter Lock (GIL)[15] guards every "instruction"
Language do have threads[16], they are "concurrent but not parallel"

[15] https://en.wikipedia.org/wiki/Global_interpreter_lock
[16] https://docs.python.org/3/library/threading.html

Mutex-based strict consistency

Strict consistency – all operations happens atomically and have total order.
Global Interpreter Lock (GIL)[15] guards every "instruction"
Language do have threads[16], they are "concurrent but not parallel"
GIL is technical decision that may highly influence the whole ecosystem

- Alternative language implementations[17]
- Memory model[18]
- Interoperability with other languages[19]

[15] https://en.wikipedia.org/wiki/Global_interpreter_lock
[16] https://docs.python.org/3/library/threading.html
[17] https://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why
[18] https://peps.python.org/pep-0583/
[19] https://peps.python.org/pep-0703/

Lecture plan

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop[20]

[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop[20]

  Each job is processed completely before any other job is processed. This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be preempted and will run entirely before any other code runs (and can modify data the function manipulates).

[20] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop

If you need parallelism – use dedicated API[21]

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

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop

If you need parallelism – use dedicated API[21]

- Safety – message passing with data copying[22]

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

[22] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop

If you need parallelism – use dedicated API[21]

- Safety – message passing with data copying[22]
- Performance – shared byte array[23] + atomics[24]

[21] https://www.w3schools.com/html/html5_webworkers.asp
[22] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm
[23] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer
[24] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics

Single-threaded strict consistency

Strict consistency – all operations happen atomically and have total order.

- Forbid multithreading on language level
- The only thread processes events, every event could create other (possibly delayed) events
- JavaScript Event Loop

If you need parallelism – use dedicated API[21]

- Safety – message passing with data copying[22]
- Performance – shared byte array[23] + atomics[24]

As soon as you have concurrent access – you immediately encounter "serious" problems[25]

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

[22] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm

[23] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

[24] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics

[25] "Repairing and Mechanising the JavaScript Relaxed Memory Model" https://arxiv.org/abs/2005.10554

Lecture plan

Threads as a library

C/C++ before C++11[26]

[26] https://en.wikipedia.org/wiki/C%2B%2B11#Multithreading_memory_model

Threads as a library

C/C++ before C++11[26]

- There were no threads in language specification
  - There were libraries that implement threading
- Every project was free to invent custom designs, data structures, conventions
  - Compiler-specific
  - CPU-specific
  - OS-specific
- Any data race is undefined behaviour

[26] https://en.wikipedia.org/wiki/C%2B%2B11#Multithreading_memory_model

Threads as universal library

C/C++ before concurrent specifications + POSIX threads[27]

[27] https://en.wikipedia.org/wiki/Pthreads

Threads as universal library

C/C++ before concurrent specifications + POSIX threads[27]

- There were no threads in language specification
  - There was **universal** library to implement threading
- Portable solution
- Any data race is undefined behaviour. Use mutexes, Luke!

[27] https://en.wikipedia.org/wiki/Pthreads

Threads as universal library

C/C++ before concurrent specifications + POSIX threads[27]

- There were no threads in language specification
  - There was **universal** library to implement threading
- Portable solution
- Any data race is undefined behaviour. Use mutexes, Luke!

Do not underestimate the complexity of implementing such library[28,29]

[27] https://en.wikipedia.org/wiki/Pthreads

[28] https://probablydance.com/2020/10/31/using-tla-in-the-real-world-to-understand-a-glibc-bug/

[29] https://probablydance.com/2022/09/17/finding-the-second-bug-in-glibcs-condition-variable/

Threads as universal library

C/C++ before concurrent specifications + POSIX threads[27]

- There were no threads in language specification
  - There was **universal** library to implement threading
- Portable solution
- Any data race is undefined behaviour. Use mutexes, Luke!

Do not underestimate the complexity of implementing such library[28,29]

"Threads Cannot Be Implemented As a Library"[30]

[27] https://en.wikipedia.org/wiki/Pthreads

[28] https://probablydance.com/2020/10/31/using-tla-in-the-real-world-to-understand-a-glibc-bug/

[29] https://probablydance.com/2022/09/17/finding-the-second-bug-in-glibcs-condition-variable/

[30] https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

Threads as universal library

C/C++ before concurrent specifications + POSIX threads[27]

- There were no threads in language specification
  - There was **universal** library to implement threading
- Portable solution
- Any data race is undefined behaviour. Use mutexes, Luke!

Do not underestimate the complexity of implementing such library[28,29]

"Threads Cannot Be Implemented As a Library"[30]

> *The Pthreads specification prohibits races, i.e. accesses to a shared variable while another thread is modifying it. ... the problem here is that whether or not a race exists depends on the semantics of the programming language, which in turn requires that we have a properly defined memory model. Thus this definition is circular.*

[27] https://en.wikipedia.org/wiki/Pthreads

[28] https://probablydance.com/2020/10/31/using-tla-in-the-real-world-to-understand-a-glibc-bug/

[29] https://probablydance.com/2022/09/17/finding-the-second-bug-in-glibcs-condition-variable/

[30] https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

Lecture plan

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

- C/C++ atomics

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

- C/C++ atomics
- Swift/ObjC NSLocking

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

- C/C++ atomics
- Swift/ObjC NSLocking
- Java-1996 volatile

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

- C/C++ atomics
- Swift/ObjC NSLocking
- Java-1996 volatile

Part of a **language**, not just a library!

Explicit concurrent operations

Specify subset of programming language that provides consistent multithreaded execution.

- C/C++ atomics
- Swift/ObjC NSLocking
- Java-1996 volatile

Part of a **language**, not just a library!
Not that easy to do: "The Java Memory Model is Fatally Flawed" , William Pugh, 2000[31]

[31] http://www.cs.umd.edu/~pugh/java/broken.pdf

Lecture plan

Formal language memory model

Mathematics to the max

Formal language memory model

Mathematics to the max

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

Formal language memory model

Mathematics to the max

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

Formal language memory model

Mathematics to the max

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

Formal language memory model

Mathematics to the max

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial and linear orders; transitive closure of binary relations; happens-before
- Causality requirements, circular happens-before, out-of-thin-air problem
- Adaptation to h/w models[32]

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

Formal language memory model

Mathematics to the max

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial and linear orders; transitive closure of binary relations; happens-before
- Causality requirements, circular happens-before, out-of-thin-air problem
- Adaptation to h/w models[32]
- Every data race is specified: allowed and forbidden outcomes

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

Formal language memory model

Mathematics to the max

- An action a is described by a tuple $\langle t, k, v, u \rangle$ comprising ...
- Partial and linear orders; transitive closure of binary relations; happens-before
- Causality requirements, circular happens-before, out-of-thin-air problem
- Adaptation to h/w models[32]
- Every data race is specified: allowed and forbidden outcomes

Few problems

- Complicated, takes a lot of time, expensive to maintain
- Not ideal[33]
- Hard to explain, hard to use in practice

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

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

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Non-goals

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Non-goals

- Look smart and mathematically inclined

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Non-goals

- Look smart and mathematically inclined
- Study state-of-the art concurrency

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Non-goals

- Look smart and mathematically inclined
- Study state-of-the art concurrency
- Achieve top performance in our concurrent programs

Formal Memory Models: useful?

Goals of our 1-semester introductory course to concurrency

- Show how to write
  - correct
  - **understandable**
  - performant

  concurrent code
- Show **basic** concurrent concepts
- Convince to use simple rather complicated approaches

Non-goals

- Look smart and mathematically inclined
- Study state-of-the art concurrency
- Achieve top performance in our concurrent programs

Studying concurrent consistency, proving theorems on Lecture 6 and Lecture 7, analyzing cache coherence was enough.

Patterns

Doug Lea, private communication with Aleksey Shipilev, 2013[34]

> *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.*

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

Concurrent patterns

Could be found in books

- "Java Concurrency in Practice"
- "Effective Java"
- "The Art of Multiprocessor Programming"
- "Is Parallel Programming Hard, And, If So, What Can You Do About It?"

Could be found in documentation `java.util.concurrent`
We studied this

- `Lock, Condition` (Lecture 2)
- advanced concurrent primitives (Lecture 3)
- producer-consumer, load balancing, partitioning, replication (Lecture 3 and Lecture 4)

Lecture plan

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

Do you see race condition that prevents `threadB` from printing integer?

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

Do you see race condition that prevents `threadB` from printing integer?
`NullPointerException`

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = (1L << 40) + 1;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = (1L << 40) + 1;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

- Could program print 1?

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = (1L << 40) + 1;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

- Could program print 1?
- Could program print 1099511627776 ($1 \ll 40$)?

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = (1L << 40) + 1;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

- Could program print 1?
- Could program print 1099511627776 ($1 \ll 40$)?
- Could program print 1099511627777?

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

- Program prints 0
- Explain this using some hardware optimization
  - store buffering, load buffering, invalidate queue, interconnect topology

Visibility

```
class Data { long x; }
static Data shared;
void threadA() {
  Data d = new Data();
  d.x = 42;
  shared = d;
}
void threadB() {
  System.out.println(shared.x);
}
```

- Program prints 0
- Explain this using some compiler optimization

Visibility: trust nobody

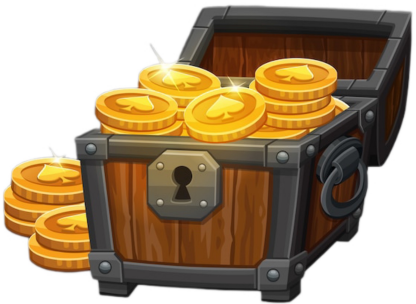
Non-synchronized concurrent access to any memory location

- Static field
- Instance field
- Array element

could lead to subtle and hard-to-diagnose bugs.

Question time

Question: Which Java-specific concepts help you to avoid visibility problems?



Visibility: trust friends only

Non-synchronized concurrent access to any memory location

- Static field
- Instance field
- Array element

could lead to subtle and hard-to-diagnose bugs.
Synchronization operations:

- `Thread.start`, `Thread.join`
- `Lock.lock`, `Lock.unlock` and other `java.util.concurrent` primitives
- `synchronized`
- `AtomicLong`, `AtomicReference` read-modify-write operations

Visibility: trust friends only

Non-synchronized concurrent access to any memory location

- Static field
- Instance field
- Array element

could lead to subtle and hard-to-diagnose bugs.
Synchronization operations:

- `Thread.start`, `Thread.join`
- `Lock.lock`, `Lock.unlock` and other `java.util.concurrent` primitives
- `synchronized`
- `AtomicLong`, `AtomicReference` read-modify-write operations

By default[35] they act as full barrier

[35] Consult with javadoc first

Visibility: trust friends only

Non-synchronized concurrent access to any memory location

- Static field
- Instance field
- Array element

could lead to subtle and hard-to-diagnose bugs.
Synchronization operations:

- `Thread.start`, `Thread.join`
- `Lock.lock`, `Lock.unlock` and other `java.util.concurrent` primitives
- `synchronized`
- `AtomicLong`, `AtomicReference` read-modify-write operations

By default[35] they act as full barrier and as a linearization point

[35] Consult with javadoc first

Lecture plan

Using memory location as synchronization point

```
class LockPreReservedByB { boolean busy = true; }
static LockPreReservedByB lock; static Data data;
void threadA() {  while (lock.busy) { /*await*/ } // lock.lock()
                  Data x = data;
}
void threadB() {  Data d = new Data(); d.x = 42;
                  data = d;
                  lock.busy = false; // lock.unlock()
}
```

Using memory location as synchronization point

```
class LockPreReservedByB { boolean busy = true; }
static LockPreReservedByB lock; static Data data;
void threadA() {  while (lock.busy) { /*await*/ } // lock.lock()
                  Data x = data;
}
void threadB() {  Data d = new Data(); d.x = 42;
                  data = d;
                  lock.busy = false; // lock.unlock()
}
```

We need

- `Lock.unlock` to act as publishing memory barrier (store release)
- `Lock.lock` to repair consistency of data being read (load acquire)
- all operations with `lock.busy` to be ordered (linear order)

Using memory location as synchronization point

Roach motel idiom[36]

```
int a; volatile boolean ready = false;
void threadA() {
  a = 41;
  a = 42;
  ready = true; // release data. Every write above will be visible (flushed)
  a = 43;
}
void threadB() {
  while (!ready) {} // acquire data. Every read below will see up-to-date data
  println(a);
}
```

[36] https://shipilev.net/talks/narnia-2555-jmm-pragmatics-en.pdf

Using memory location as synchronization point

Roach motel idiom[36]

```
int a; volatile boolean ready = false;
void threadA() {
  a = 41;
  a = 42;
  ready = true; // release data. Every write above will be visible (flushed)
  a = 43;
}
void threadB() {
  while (!ready) {} // acquire data. Every read below will see up-to-date data
  println(a);
}
```

Possible to see 42 or 43. Impossible to see 0 or 41.

[36] https://shipilev.net/talks/narnia-2555-jmm-pragmatics-en.pdf

Volatile

`volatile` variable

- `volatile` mode accesses are totally ordered
  - coherence, visibility (progress)
- not a plain variable, enforces some ordering with other locations
  - load acquire, store release
- not an atomic read-modify-write variable, does not provide full barrier semantics

Volatile

`volatile` variable

- `volatile` mode accesses are totally ordered
  - coherence, visibility (progress)
- not a plain variable, enforces some ordering with other locations
  - load acquire, store release
- not an atomic read-modify-write variable, does not provide full barrier semantics

Kinda *weak*

- advanced protocols
- performance

Volatile

`volatile` variable

- `volatile` mode accesses are totally ordered
  - coherence, visibility (progress)
- not a plain variable, enforces some ordering with other locations
  - load acquire, store release
- not an atomic read-modify-write variable, does not provide full barrier semantics

Kinda *weak*

- advanced protocols
- performance

Very Java-specific

- `volatile` in C/C++ is **absolutely different**
- `memory_order_acq_rel` in C/C++ is **subtly different** from Java `volatile`
- many languages are OK with plain variables and full-fence variables
- low-level languages allow to use atomic RMW variables with different "strength"

Volatile: rules of thumb

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

```
volatile boolean asleep;
...
while (!asleep) {
  countSomeSheep();
}
```

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

```
volatile boolean asleep;
...
while (!asleep) {
  countSomeSheep();
}
```

You can use `volatile` variables only when all the following criteria are met:

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

```
volatile boolean asleep;
...
while (!asleep) {
  countSomeSheep();
}
```

You can use `volatile` variables only when all the following criteria are met:

- writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

```
volatile boolean asleep;
...
while (!asleep) {
  countSomeSheep();
}
```

You can use `volatile` variables only when all the following criteria are met:

- writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value
- the variable does not participate in invariants with other state variables

Volatile: rules of thumb

- use `volatile` only when it simplify your synchronization policy
- avoid `volatile` when veryfing correctness would require subtle reasoning about visibility
- good uses of `volatile` include ensuring the visibility of their own state or indicating that an important life-cycle event (such as initialization or shutdown) has occurred

```
volatile boolean asleep;
...
while (!asleep) {
  countSomeSheep();
}
```

You can use `volatile` variables only when all the following criteria are met:

- writes to the variable do not depend on its current value, or you can ensure that only a single thread ever updates the value
- the variable does not participate in invariants with other state variables
- locking is not required for any other reason while the variable is being accessed

Summary

Compiler optimizations

- reorder/invent/delete memory operations, prevent it via compiler barriers

Language memory model – bridge between

- chaos of hardware, anarchy of compiler optimizations
- consistency requirements for high-level concurrent algorithms

Useful concepts

- immutability
- DSL for parallel computations
- strict consistency via mutex
- strict consistency via single executor and cooperative multitasking
- threads cannot be implemented as a library
- language memory model is harder to formalize than hardware memory model
- visibility is subtle but critically important
- `volatile` is for hackers

The most efficient and elegant approach to writing concurrent code

https://go.dev/ref/mem

The most efficient and elegant approach to writing concurrent code

`https://go.dev/ref/mem`

Don't be clever