A love/hate relationship:

The JVM/OS dialectic

Marcus Lagergren, Oracle

A love/hate relationship:

диалектика JVM/OS

Marcus Lagergren, Oracle

A love/hate relationship:

The JVM/OS dialectic

Marcus Lagergren, Oracle

A love/hate relationship:

The JVM/OS dialectic

(and the hardware…)

Marcus Lagergren, Oracle

A love/hate relationship:

The JVM/OS dialectic

(and the hardware…)

Marcus Lagergren, Oracle

Fredrik Öhrström, Spotify

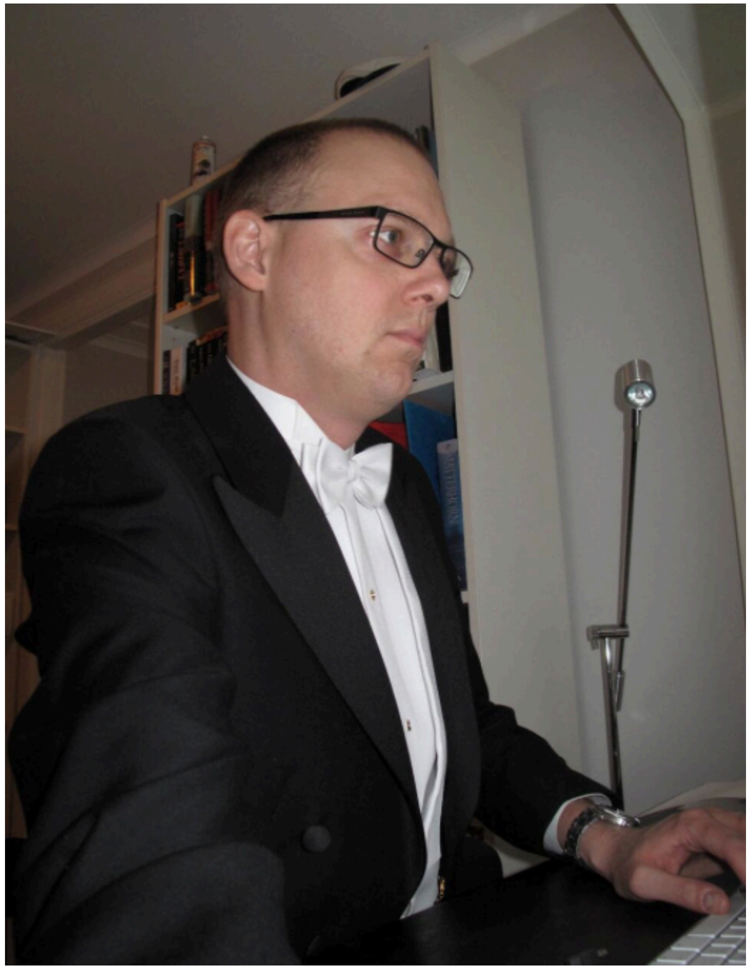
The Legal Slide

"THE FOLLOWING IS INTENDED TO OUTLINE OUR GENERAL PRODUCT DIRECTION. IT IS INTENDED FOR INFORMATION PURPOSES ONLY, AND MAY NOT BE INCORPORATED INTO ANY CONTRACT. IT IS NOT A COMMITMENT TO DELIVER ANY MATERIAL, CODE, OR FUNCTIONALITY, AND SHOULD NOT BE RELIED UPON IN MAKING PURCHASING DECISION. THE DEVELOPMENT, RELEASE, AND TIMING OF ANY FEATURES OR FUNCTIONALITY DESCRIBED FOR ORACLE'S PRODUCTS REMAINS AT THE SOLE DISCRETION OF ORACLE."

Agenda

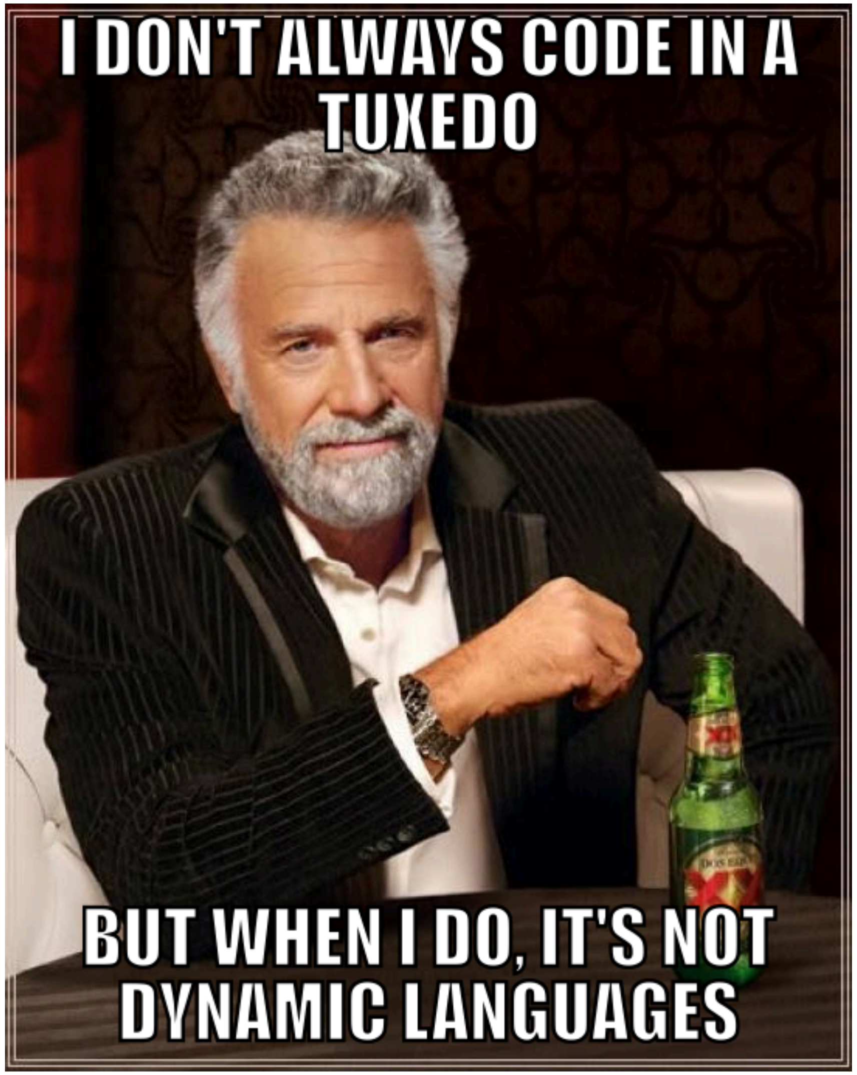
- In the borderlands between hardware, OS and JVM, both good and bad things happen
- Computer history
- How do they affect each other?
- Where is it all going?

Who am I?

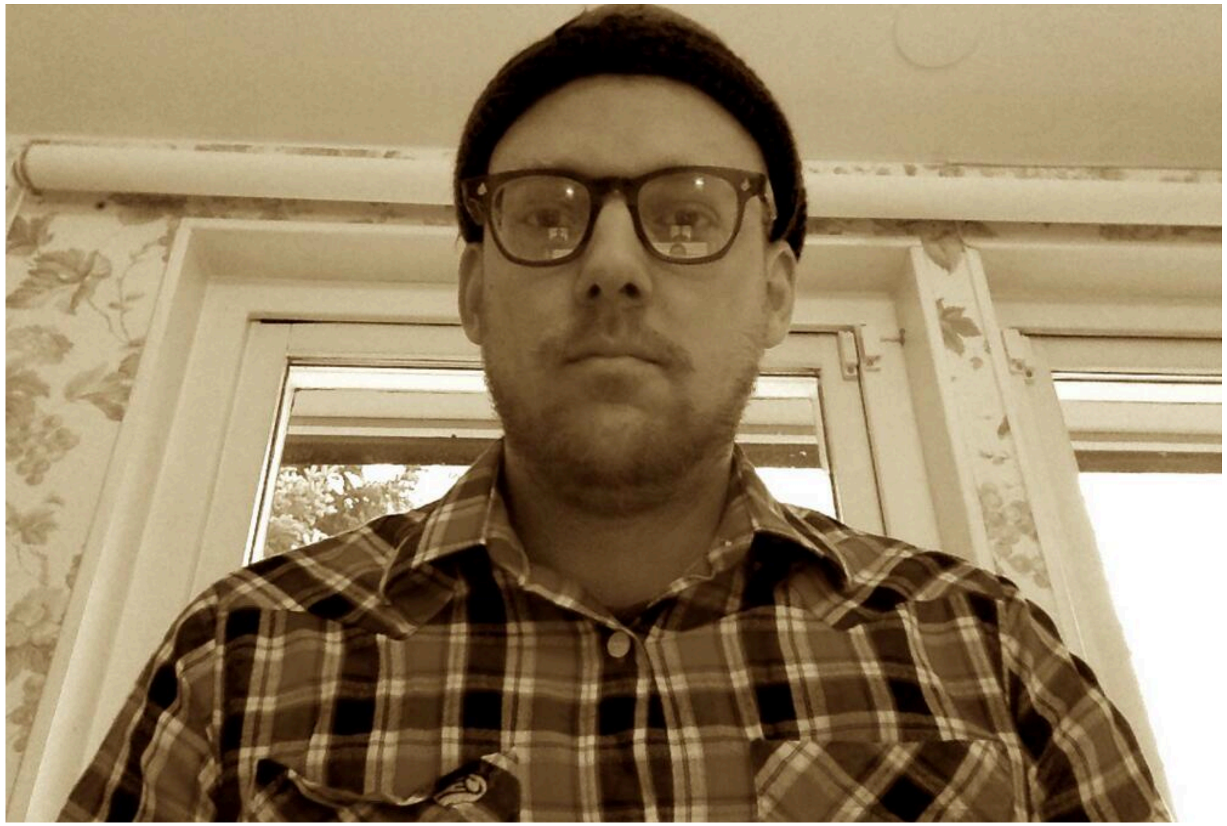


@lagergren

Who am I?

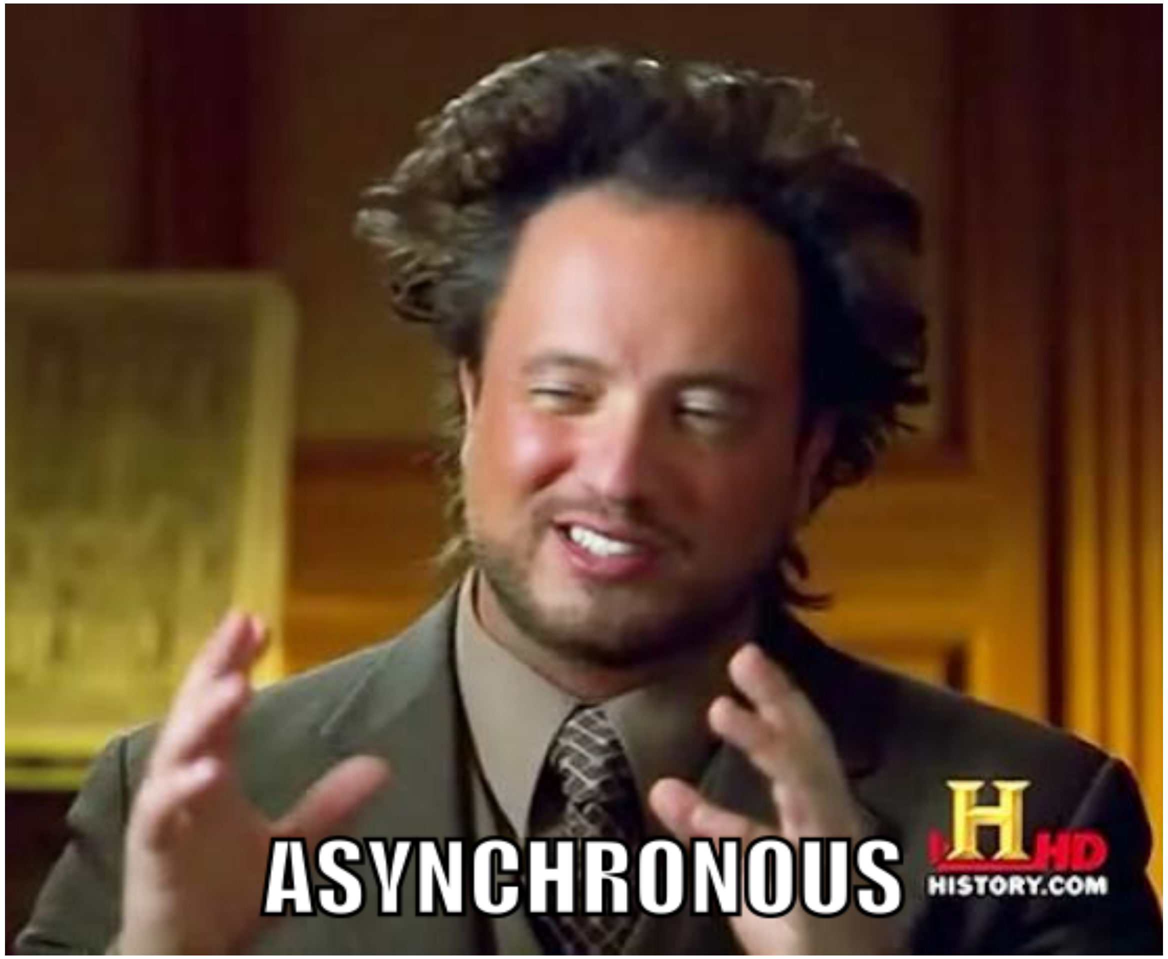

I DON'T ALWAYS CODE IN A TUXEDO
BUT WHEN I DO, IT'S NOT DYNAMIC LANGUAGES

Who am I?

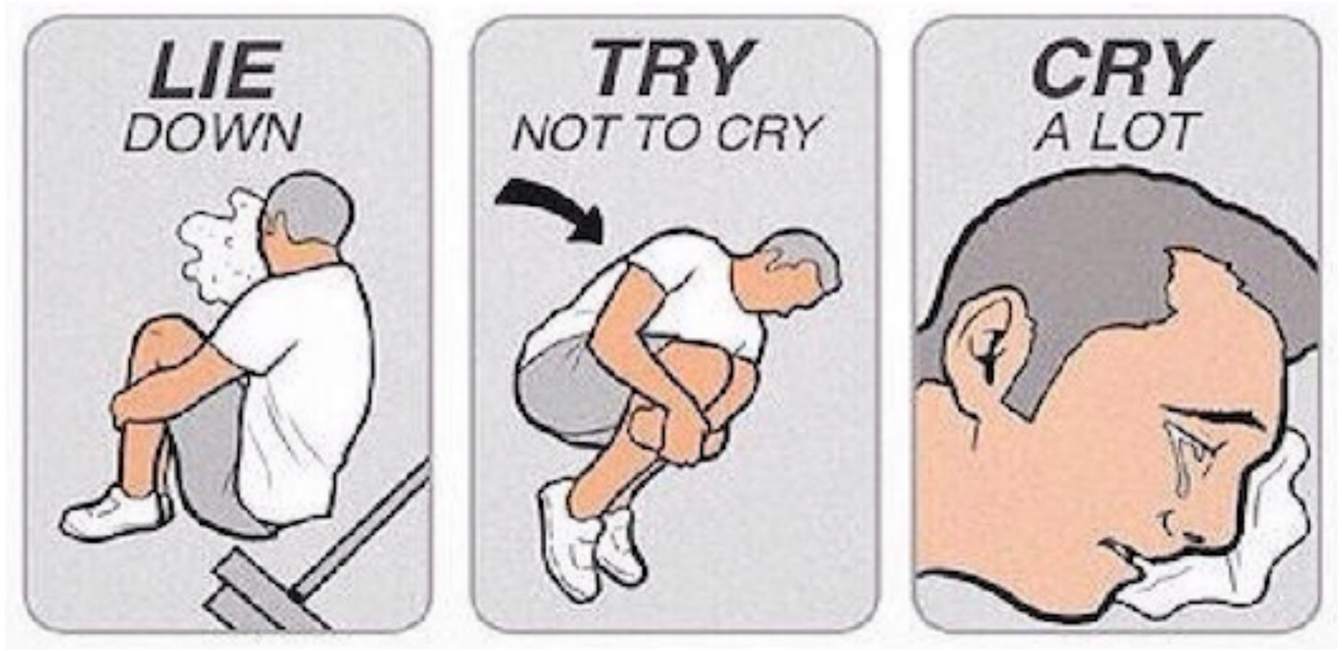


mlvm-dev@openjdk.java.net
nashorn-dev@openjdk.java.net
https://avatar.java.net

Who am I?

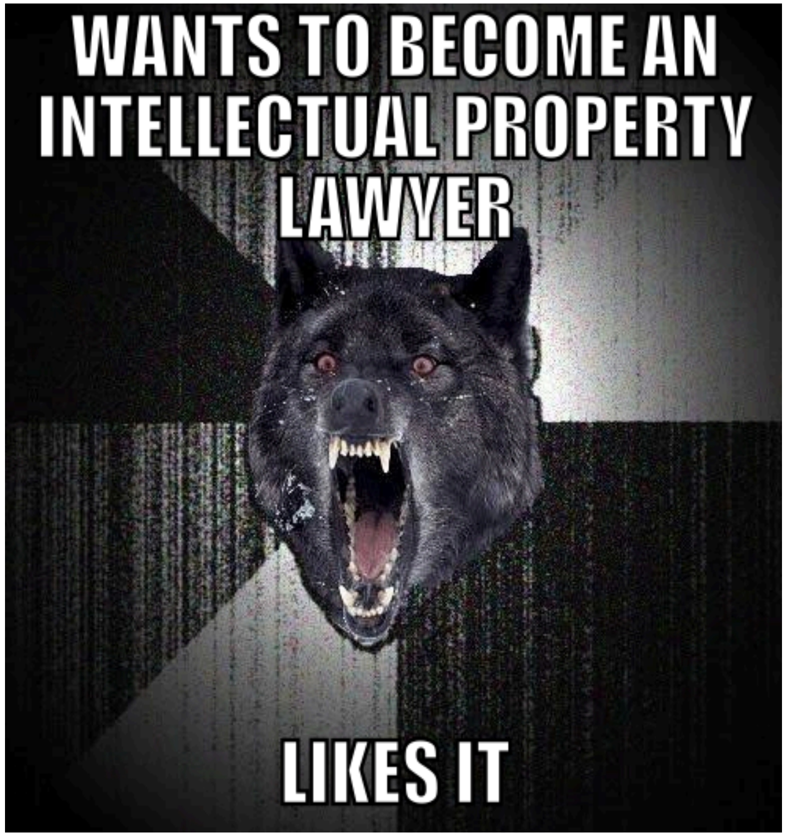


Who am I?

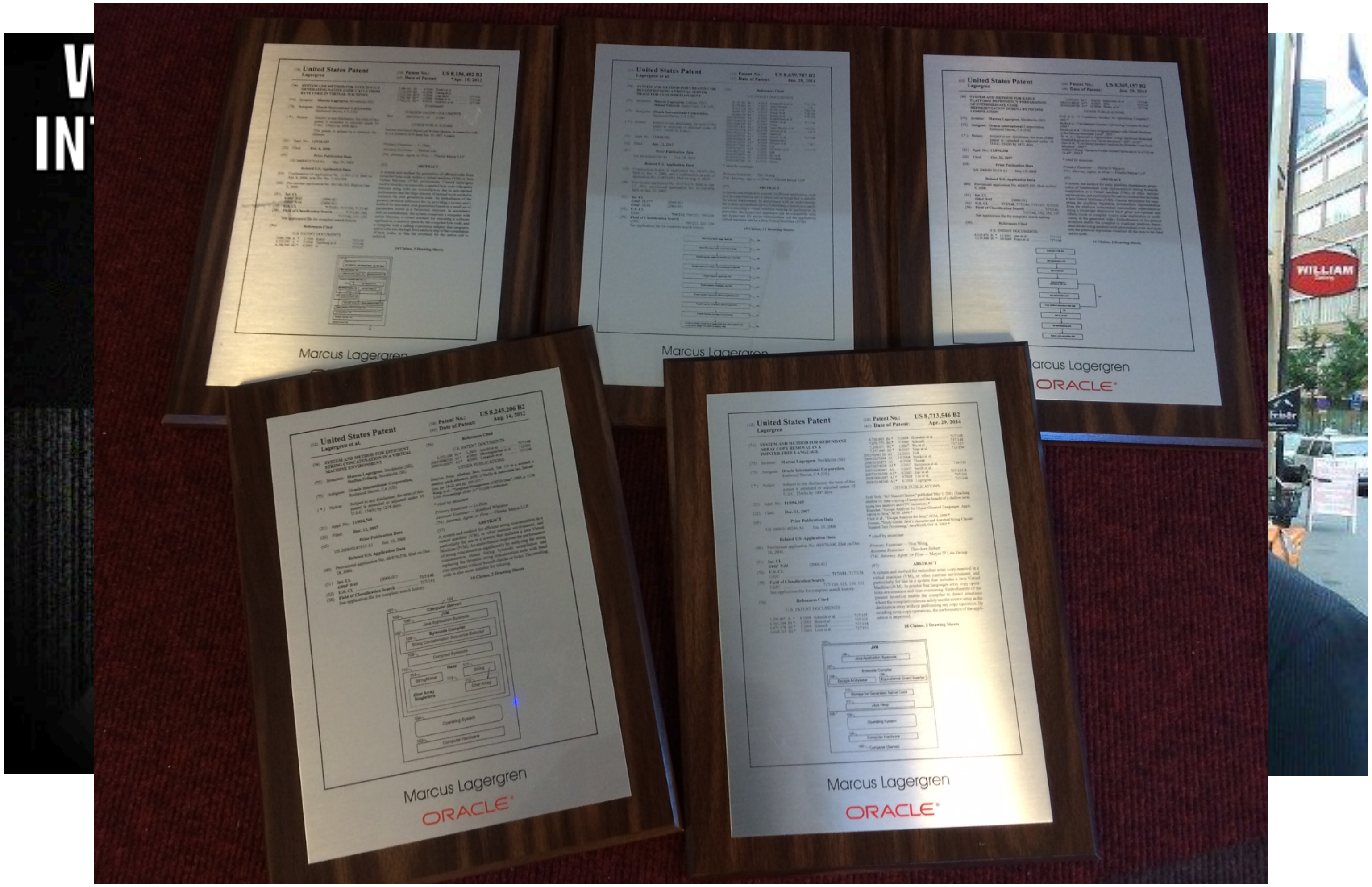


Who is Fredrik?

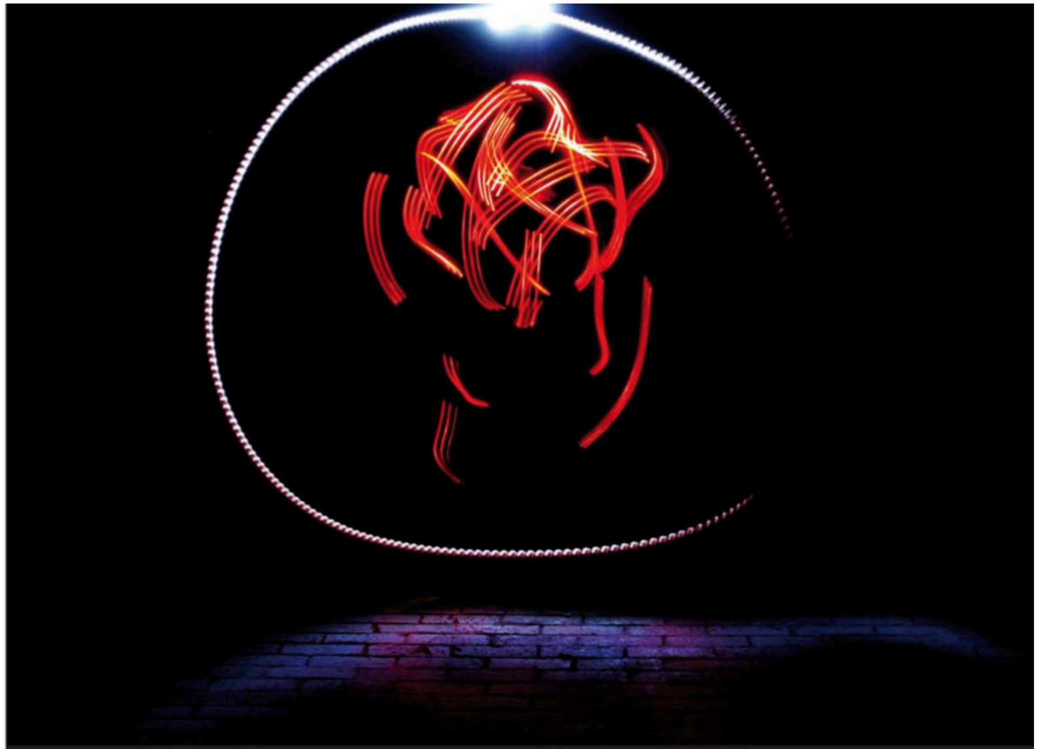
Who is Fredrik?



Who is Fredrik?



Who is Fredrik?



Professional Expertise Distilled

Oracle JRockit

The Definitive Guide

Develop and manage robust Java applications with Oracle's high-performance Java Virtual Machine

Foreword by Adam Messinger,
Vice President of Development in the Oracle Fusion Middleware group

Marcus Hirt Marcus Lagergren

[PACKT] PUBLISHING enterprise professional expertise distilled

The Past

The Past

(Skipping very quickly over a tremendous amount of hardware)

Texas Instruments TI 99/4a

- 1979-1984
- Contains an OS and a GPL interpreter
- Device drivers (DSRs) could be written in GPL

Texas Instruments TI 99/4a

- 1979-1984
- Contains an OS and a GPL interpreter
- Device drivers (DSRs) could be written in GPL
- They intended to execute GPL bytecode natively
  - But they never did

VIC 20 and Commodore 64

- 1980-1986

VIC 20 and Commodore 64

- 1980-1986

`http://codebase64.org`

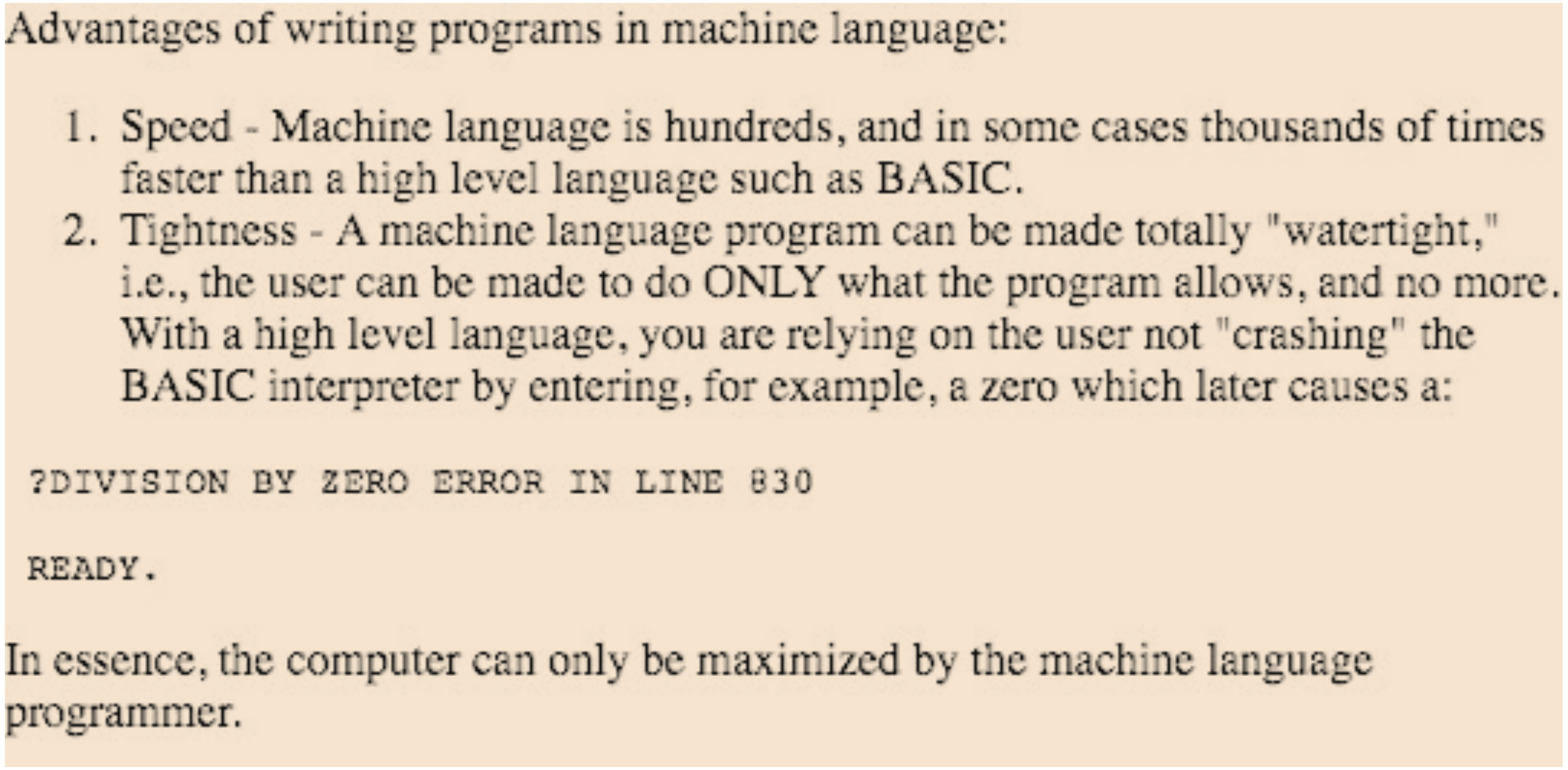
VIC 20 and Commodore 64

- 1980-1986

VIC 20 and Commodore 64

- 1980-1986
- Basic interpreter written in C on my Mac
  - 1000x faster than a physical 6502

Nothing new under the sun?



Advantages of writing programs in machine language:

1. Speed - Machine language is hundreds, and in some cases thousands of times faster than a high level language such as BASIC.
2. Tightness - A machine language program can be made totally "watertight," i.e., the user can be made to do ONLY what the program allows, and no more. With a high level language, you are relying on the user not "crashing" the BASIC interpreter by entering, for example, a zero which later causes a:

```
?DIVISION BY ZERO ERROR IN LINE 830

READY.
```

In essence, the computer can only be maximized by the machine language programmer.

…and stuff

MITS Altair 8800, Commodore PET 2001, Apple II, Atari VCS, Tandy Radio Shack TRS-80, ABC 80, NASCOM-1, Sharp MZ-80k, Atari 400/800, Mattel Intellivision, Tangerine Microtan 65, HP-85, Sinclair ZX80, Acorn Atom, Sinclair ZX81, Osborne 1, IBM PC, BBC Micro, Sinclair ZX Spectrum, Coleco Vision, GCE/MB Vectrex, Grundy Newbrain, Dragon 32, Jupiter ACE, Compaq Portable, Apple Lisa, Oric 1, Mattel Aquarius, Nintendo Famicom/NES, Acorn Electron, Sony MSX, Apple Macintosh, Sinclair QL, Amstrad CPC-464, IBM PC AT, Tatung Einstein, Atari ST, Commodore Amiga, Amstrad PCW, Sega Master System, Acorn Archimedes, NeXT

The JavaStation

- 1996-2000
- Contains JavaOS, a micro kernel in C with an interpreter
- Device drivers were written in Java

The JavaStation

- 1996-2000
- Contains JavaOS, a micro kernel in C with an interpreter
- Device drivers were written in Java
- They intended to execute bytecode natively
  - But they never did

Intermediate Languages

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode

…and more stuff

Actionscript, Adobe Flash objects, BANCStar, CLISP, CMUCL, CLR/.NET, Dalvik, Dis, EiffelStudio, Emacs eLisp->bytecode, Embeddable Common Lisp, Erlang/BEAM, Icon, Unicon, Infocom/Z-machine text adventure games, LLVM, Lua, m-code/MATLAB, OCaml, Parrot Virtual Machine, R, Scheme 48, Smalltalk, SPIN/Parallax Propeller Microcontroller, SWEET16/Apple II Basic ROM, Visual FoxPro bytecode, YARV, Rubinius

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode
  - memory protection, type and control verification and explicit security management, “sandbox” model, “object orientation”

Intermediate Languages

- 1966: O-code (BCPL)
- 1970: p-code (Pascal)
- 1979: GPL
- 1995: Java Bytecode
  – memory protection, type and control verification and explicit security management, “sandbox” model, “object orientation”
- 1999: The JavaOS is discontinued

Intermediate Languages

Java source	C-like language
Java bytecode	p-code with oo
Machine code	x86 or ARM
Native Memory	OS/Linker occupies it
Libc	programmer friendly API to OS
OS	device drivers
Hypervisor	device drivers
Hardware	
Microinstructions	jitted x86 ops

Intermediate Languages

Java source	C-like language
Java bytecode	p-code with oo
[redacted]	x86 or ARM
[redacted]	OS/Linker occupies it
[redacted]	programmer friendly API to OS
[redacted]	device drivers
[redacted]	device drivers
[redacted]	
[redacted]	jitted x86 ops

To put it simply

- The JVM has OS-like behavior
  - Threads
  - Memory management/protection
  - Locking
- All this is somewhat mitigated through libc & other libraries

Threads

Threads

- Heavy weight processes
  - Slow switching
  - fork()

Threads

- Heavy weight processes
  - Slow switching
  - fork()
- Green threads
  - Fast switching, difficult to implement
    - Native Locks?

Threads

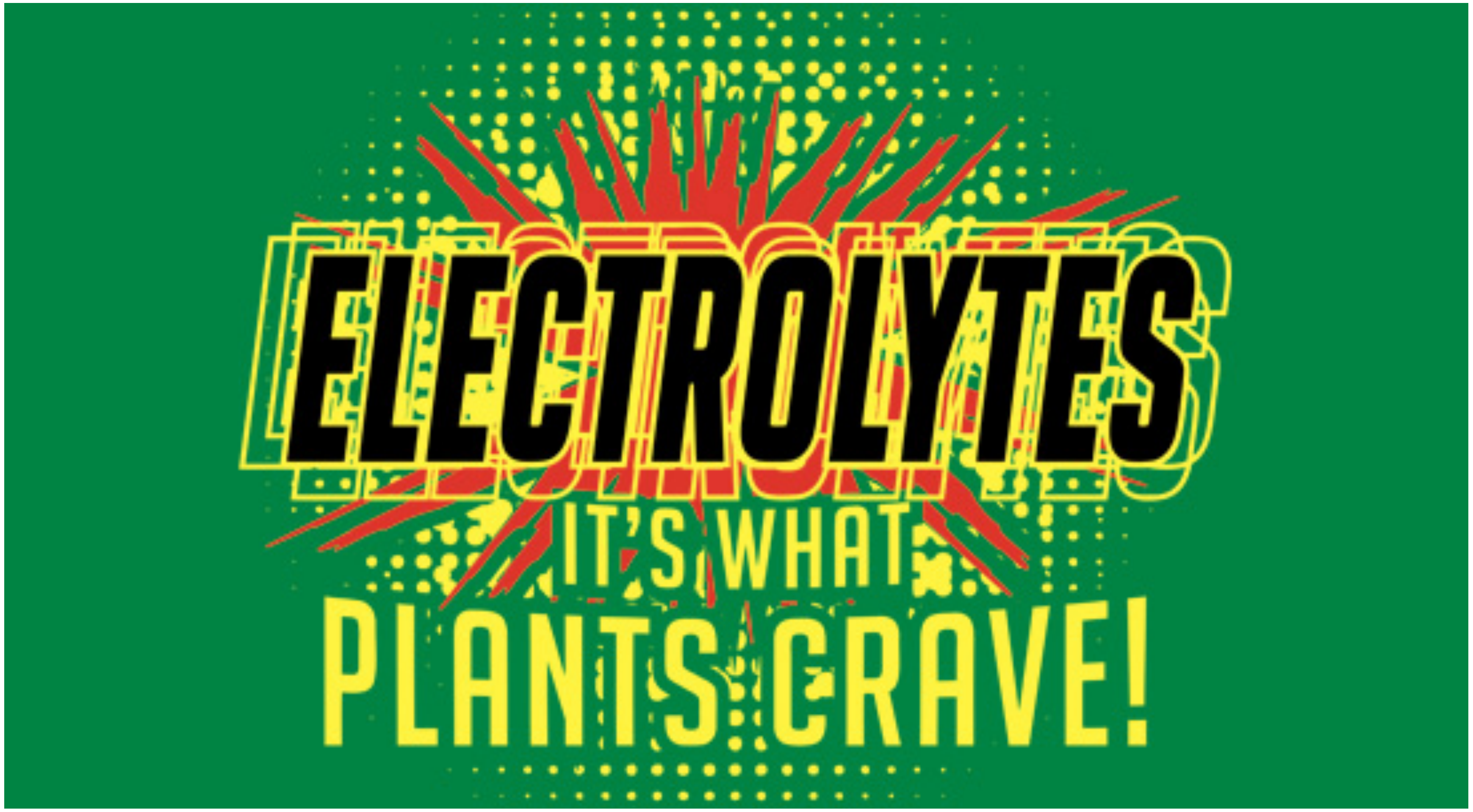
- Heavy weight processes
  - Slow switching
  - fork()
- Green threads
  - Fast switching, difficult to implement
    - Native Locks?
- MxN threads
  - Even more difficult to implement

Threads

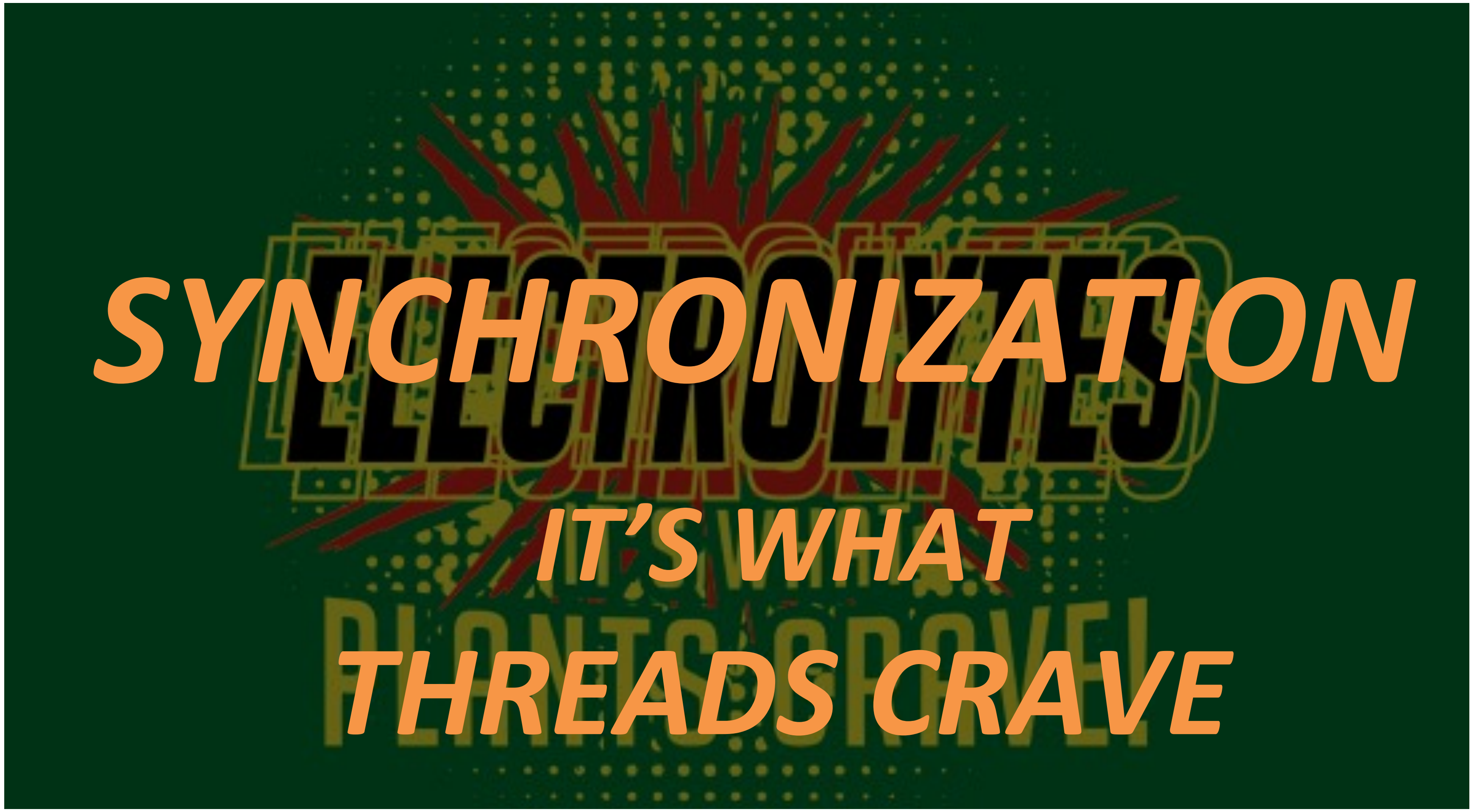
- The old standby: OS Threads
- No support for stack overflow
- By definition, no memory protection between threads

Locks

Locks



Locks



Thin Locks

```
public class PseudoSpinlock {

    private static final int LOCK_FREE = 0;
    private static final int LOCK_TAKEN = 1;

    public void lock() {
        //burn cycles
        while (cmpxchg(LOCK_TAKEN, &lock) == LOCK_TAKEN) {
            micropause(); //optional
        }
    }

    public void unlock() {
        int old = cmpxchg(LOCK_FREE, &lock);
        //guard against recursive locks
        assert(old == LOCK_TAKEN);
    }
}
```

Thin Locks

- Use whatever atomic support there is in the hardware / OS
- Cheap to lock and unlock, expensive to keep locked

Fat Locks

- Use OS lock support
- Expensive to lock and unlock, cheap to keep locked
- Need for more advanced synchronization mechanisms
  - `wait`
  - `notify`

Adaptive Behavior

- Profile based transmutation of thin locks to fat locks
  - …and vice versa
  - Nothing your C program can do

Adaptive Behavior

- Profile based transmutation of thin locks to fat locks
  - …and vice versa
  - Nothing your C program can do
- Biased locking

Adaptive Behavior

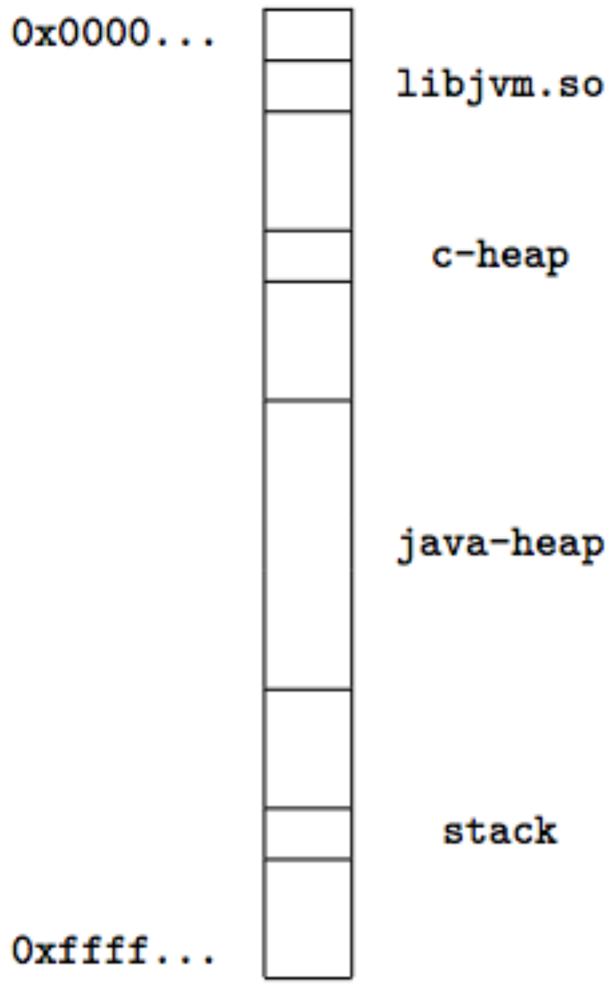
- Profile based transmutation of thin locks to fat locks
  - …and vice versa
  - Nothing your C program can do
- Biased locking
- Thread switching heuristics / Cache warmup

Adaptive Behavior

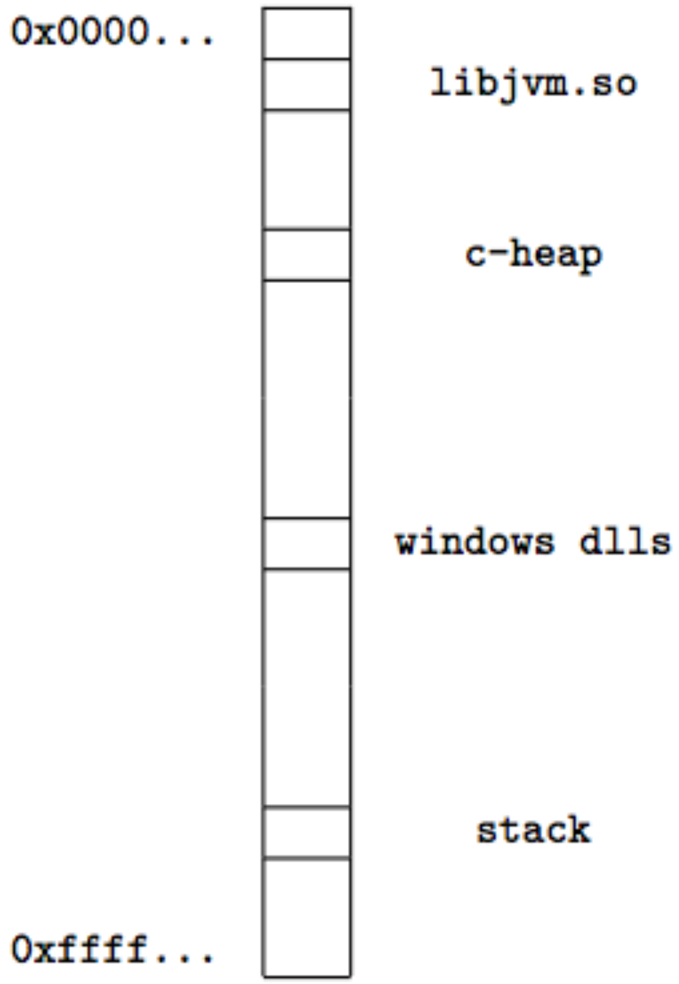
- Constant tension between OS switching and Java switching.
  - One example of a JVM/OS battle

Native Memory

Native Memory



Native Memory



Native Memory Tracking

HotSpot:

```
java -XX:NativeMemoryTracking=<summary|detail> Test
jcmd <pid> VM.native_memory
```

JRockit:

```
USE_OS_MALLOC=0 TRACE_ALLOC_SITES=1 java Test
jrcmd <pid> print_memusage
```

Memory Models

Memory Models

```
public class WhileLoop {
   //can be accessed by other threads
   private boolean finished;

   while (!finished) {
      do something…
   }
}
```

Memory Models

```
public class WhileLoop {
   //can be accessed by other threads
   private boolean finished;

   boolean tmp = finished;
   while (!tmp) {
      do something…
   }
}
```

Memory Models

```
public class WhileLoop {
   //can be accessed by other threads
   private volatile boolean finished;

   while (!finished) {
      do something…
   }
}
```

Memory Models

```java
volatile int x;
int y;
volatile boolean finished;

x = 17;
y = 4711;
finished = true;

if (finished) {
   System.err.println(x);
   System.err.println(y);
}
```

Memory Models

```
volatile int x;
int y;
volatile boolean finished;

x = 17;
y = 4711;
finished = true;

if (finished) {
   System.err.println(x);
   System.err.println(y);
}
```

Memory Models

```java
public class GadgetHolder {
    private Gadget theGadget;

    public synchronized Gadget getGadget() {
        if (this.theGadget == null) {
            this.theGadget = new Gadget();
        }
        return this.theGadget;
    }
}
```

Memory Models

```java
public class GadgetHolder {
    private Gadget theGadget;

    public Gadget getGadget() {
        if (this.theGadget == null) {
            synchronized(this) {
                if (this.theGadget == null) {
                    this.theGadget = new Gadget();
                }
            }
        }
        return this.theGadget;
    }
}
```

Memory Models

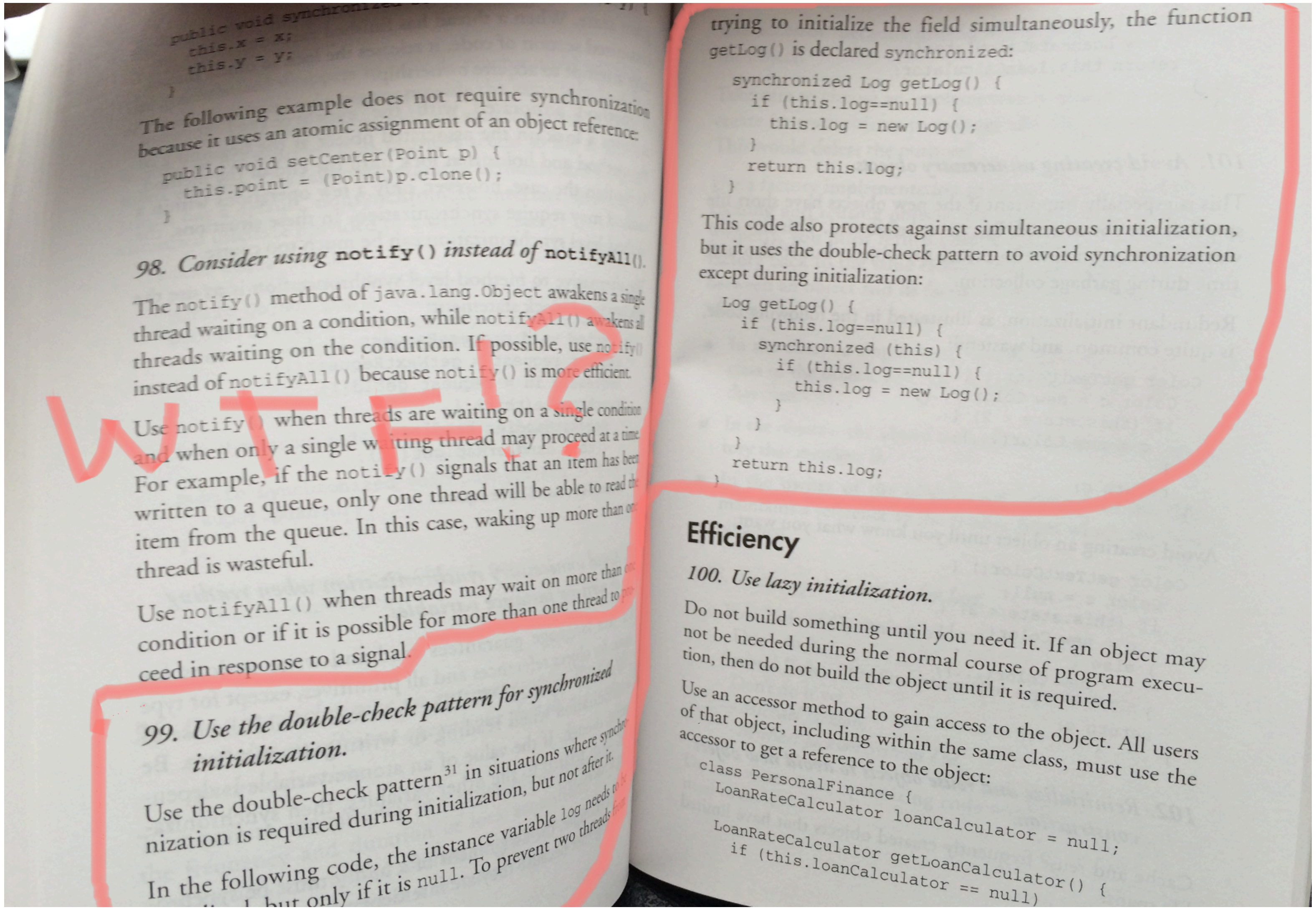
```
public class GadgetHolder {
    private volatile Gadget theGadget;

    public Gadget getGadget() {
        if (this.theGadget == null) {
            synchronized(this) {
                if (this.theGadget == null) {
                    this.theGadget = new Gadget();
                }
            }
        }
        return this.theGadget;
    }
}
```

Memory Models

```
public class GadgetMaker {
    public static Gadget theGadget = new Gadget();
}
```

WTF?



```
public void synchroni
  this.x = x;
  this.y = y;
}
```

The following example does not require synchronization because it uses an atomic assignment of an object reference:

```
public void setCenter(Point p) {
  this.point = (Point)p.clone();
}
```

98. Consider using **notify()** *instead of* **notifyAll()**.

The notify() method of java.lang.Object awakens a single thread waiting on a condition, while notifyAll() awakens all threads waiting on the condition. If possible, use notify() instead of notifyAll() because notify() is more efficient.

Use notify() when threads are waiting on a single condition and when only a single waiting thread may proceed at a time. For example, if the notify() signals that an item has been written to a queue, only one thread will be able to read the item from the queue. In this case, waking up more than one thread is wasteful.

Use notifyAll() when threads may wait on more than one condition or if it is possible for more than one thread to proceed in response to a signal.

99. Use the double-check pattern for synchronized initialization.

Use the double-check pattern[31] in situations where synchronization is required during initialization, but not after it.

In the following code, the instance variable log needs to be initialized, but only if it is null. To prevent two threads from

trying to initialize the field simultaneously, the function getLog() is declared synchronized:

```
synchronized Log getLog() {
  if (this.log==null) {
    this.log = new Log();
  }
  return this.log;
}
```

This code also protects against simultaneous initialization, but it uses the double-check pattern to avoid synchronization except during initialization:

```
Log getLog() {
  if (this.log==null) {
    synchronized (this) {
      if (this.log==null) {
        this.log = new Log();
      }
    }
  }
  return this.log;
}
```

Efficiency

100. Use lazy initialization.

Do not build something until you need it. If an object may not be needed during the normal course of program execution, then do not build the object until it is required.

Use an accessor method to gain access to the object. All users of that object, including within the same class, must use the accessor to get a reference to the object:

```
class PersonalFinance {
  LoanRateCalculator loanCalculator = null;
  LoanRateCalculator getLoanCalculator() {
    if (this.loanCalculator == null)
```

Taking Control of the OS

Taking Control of the OS

- Taking control of the OS
- Taking control of the native memory
- Taking control of the C heap

Taking Control of the OS

- Taking control of the OS
- Taking control of the native memory
- Taking control of the C heap

- Well you can’t really, but you can do your best

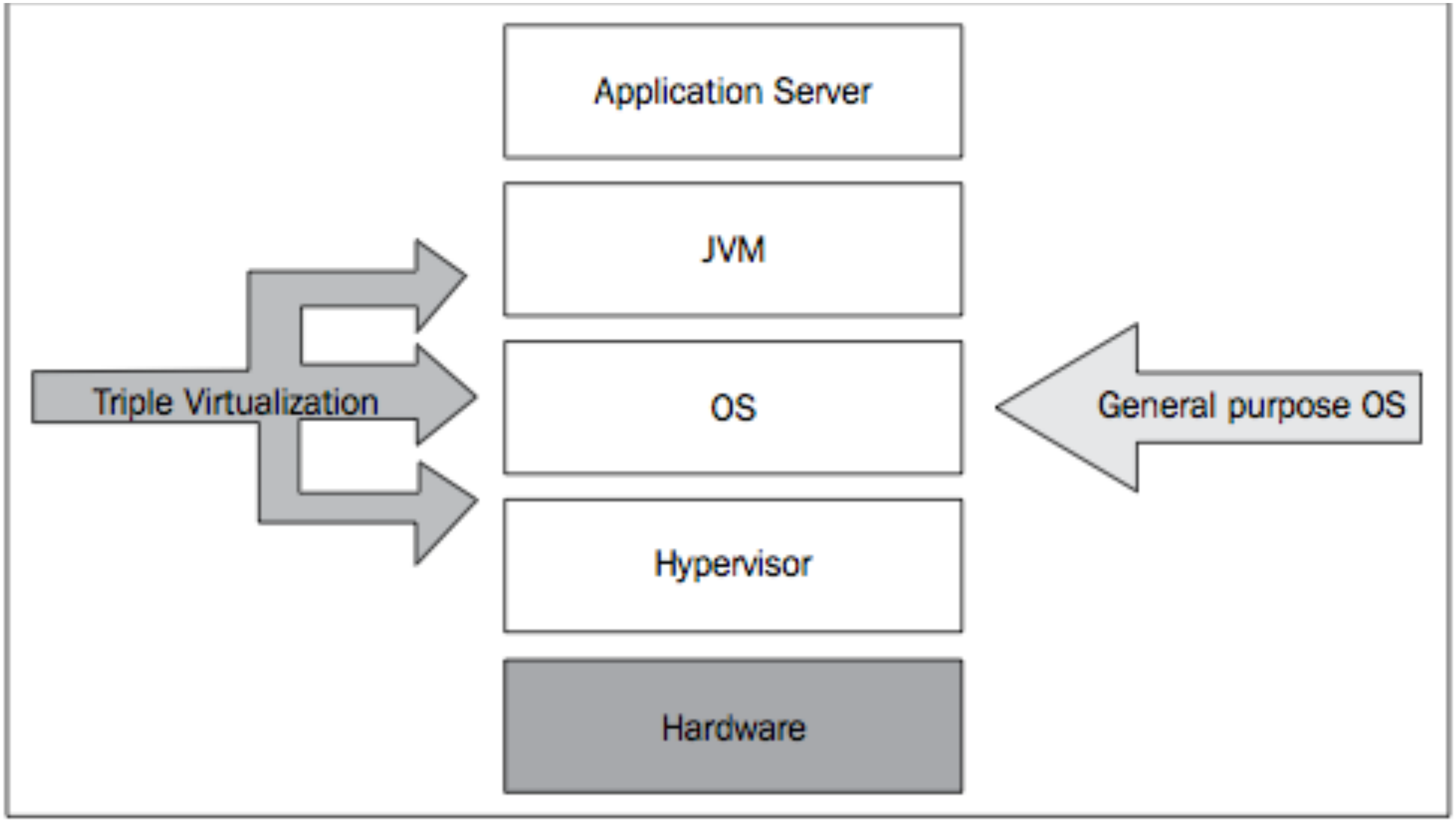
Is the JVM an OS?

- JRockit Virtual Edition
- Azul
- Cloudius
- Jnode
- …

Is the JVM an OS?

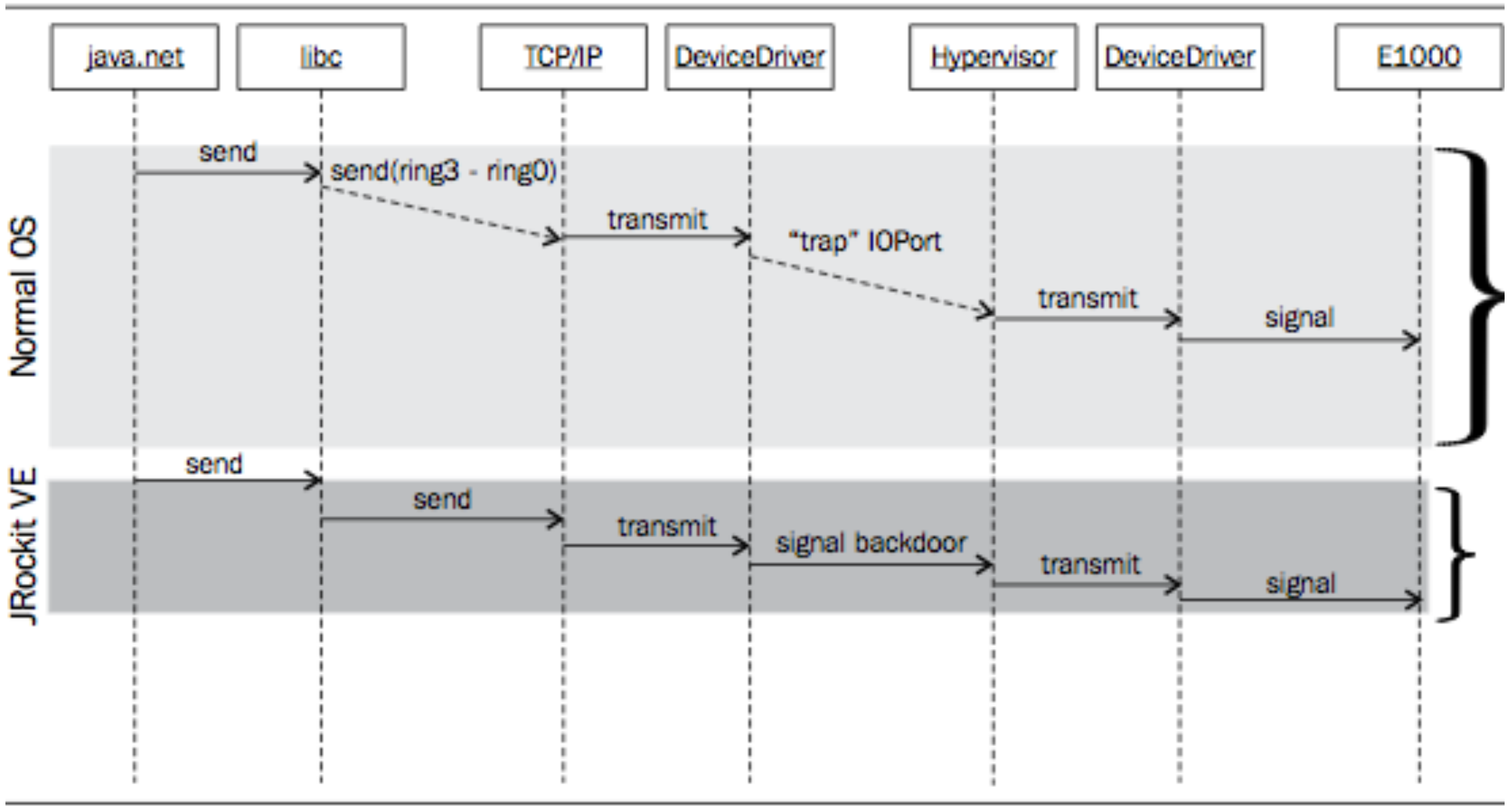
- JRockit Virtual Edition
- Implemented libc, libraries and the OS
  - Not much required for a single process Java OS.
- Finally, the Java OS?

Is the JVM an OS?



AMIGA

Is the JVM an OS?



Is the JVM an OS?

- Add a cooperative aspect to thread switching
- Zero-copy networking code
- Reduce cost of entering OS
- Balloon driver
- Runs only on hypervisor

Pauseless GC

- Hope that a lot of data is thread local and remains thread local
  - (it usually is)

Pauseless GC

- Hope that a lot of data is thread local and remains thread local
  - (it usually is)
- Use one large global heap and one thread local heap per thread

Pauseless GC

- Hope that a lot of data is thread local and remains thread local
  - (it usually is)
- Use one large global heap and one thread local heap per thread
- If thread local data is exposed to another thread – promote to global heap

Pauseless GC

- We need a write barrier

```
//x.field = y
void checkWriteAccess(Object x, Object y) {
   if (x.isOnGlobalHeap() && !y.isOnGlobalHeap()) {
      GC.registerReferenceGlobalToLocal(x, y);
   }
}
```

Pauseless GC

- … and a read barrier

```
//read x.field
void checkReadAccess(Object x) {
    int myTid = getThreadId();

    //if this object is thread local &&
    //belongs to another thread, evacuate to global heap
    if (!x.isOnGlobalHeap() && !x.isInternalTo(myTid)) {
      x.evaculateToGlobalHeap(); //painful
    }
}
```

Pauseless GC

- Barriers have to extremely fast, or everything will disappear in overhead

Pauseless GC

- Barriers have to extremely fast, or everything will disappear in overhead

Pauseless GC

- We can, because we own and implemented the thread system

Even without Pauseless GC, for large app servers with typical workloads, JRockit VE beat physical Linux!

"HOW COOL IS THAT!?!?"

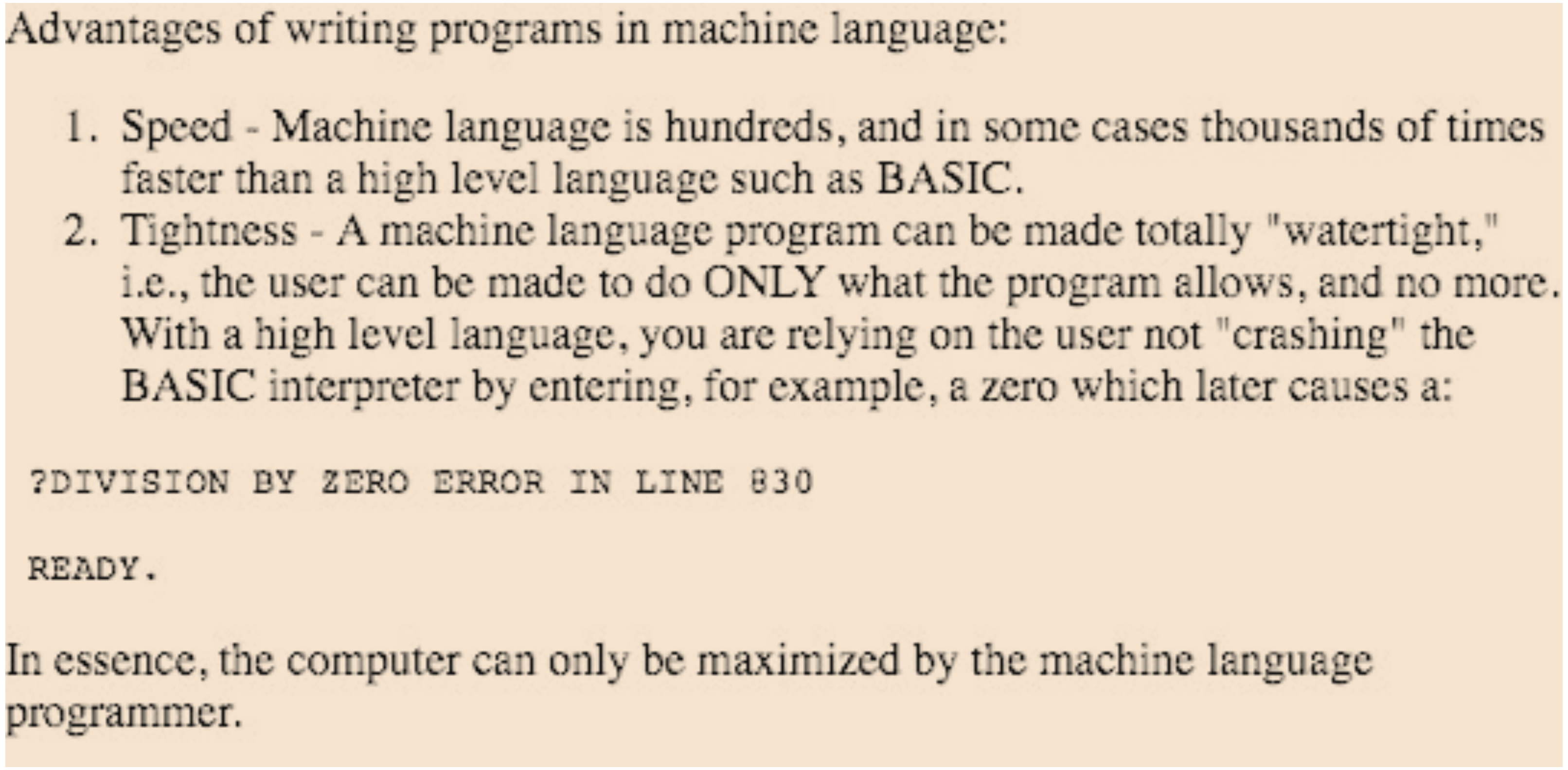
OS/JVM/Hardware improvements

- Threading
- Locking
- Native memory usage
- Virtual address memory usage/exhaustion
- Stack overflows
- Page protection

OS/JVM/Hardware improvements

- Trap on overflow arithmetic
- Read barriers
- Performance counters
  - Instruction pointer (program counter) samples
  - Cache misses
  - Userland, please
- …

So?



Advantages of writing programs in machine language:

1. Speed - Machine language is hundreds, and in some cases thousands of times faster than a high level language such as BASIC.
2. Tightness - A machine language program can be made totally "watertight," i.e., the user can be made to do ONLY what the program allows, and no more. With a high level language, you are relying on the user not "crashing" the BASIC interpreter by entering, for example, a zero which later causes a:

```
?DIVISION BY ZERO ERROR IN LINE 830

READY.
```

In essence, the computer can only be maximized by the machine language programmer.

Conclusion

- It doesn’t hurt to know what’s inside your execution environment
- In the future – the distance between hardware, OS and runtime will decrease or disappear altogether.
  - Likely starting as described
  - But possibly in ways we can’t forsee

Q & A?