Applications

	MySQL	Apache	Mozilla	OpenOffice
Software Type	Server	Server	Client	GUI
Language	C++/C	Mainly C	C++	C++
LOC (M line)	2	0.3	4	6
Bug DB history	6 years	7 years	10 years	8 years

ORACLE

Java Language and Virtual Machine Specifications

Java SE 13

Released September 2019 as JSR 388

The Java Language Specification, Java SE 13 Edition

- HTML | PDF
- Preview features: Switch expressions and Text blocks

The Java Virtual Machine Specification, Java SE 13 Edition

- HTML | PDF

Java SE 12

Released March 2019 as JSR 386

The Java Language Specification, Java SE 12 Edition

- HTML | PDF
- Preview feature: Switch expressions

The Java Virtual Machine Specification, Java SE 12 Edition

- HTML | PDF

JSR-133: Java™ Memory Model and Thread Specification

August 24, 2004, 4:42pm

This document is the JSR-133 specification, the *Java™ Memory Model and Thread Specification* (JMM), as developed by the JSR-133 expert group. This specification is part of the JSR-176 umbrella for the Tiger (5.0) release of the Java™ platform, and the normative contents of this specification will be incorporated into *The Java™ Language Specification* (JLS), *The Java™ Virtual Machine Specification* (JVMS), and the specification for classes in the `java.lang` package. This JSR-133 specification will not be further maintained or changed through the JCP. All future updates, corrections and clarifications to the normative text will occur in those other documents.

The normative contents of this specification are contained in Sections 5, 7, 9.2, 9.3, 11, 12, 14, 15 and 16. The other sections, as well as portions of the above mentioned sections, contain non-normative text that is intended to explain and clarify the normative text. In case of a conflict between the normative text and the non-normative text, the normative text stands.

The discussion and development of this specification has been unusually detailed and technical, involving insights and advances in a number of academic topics. This discussion is archived (and continues) at the JMM web site. The web site provides additional information that may help in understanding how this specification was arrived at; it is located at

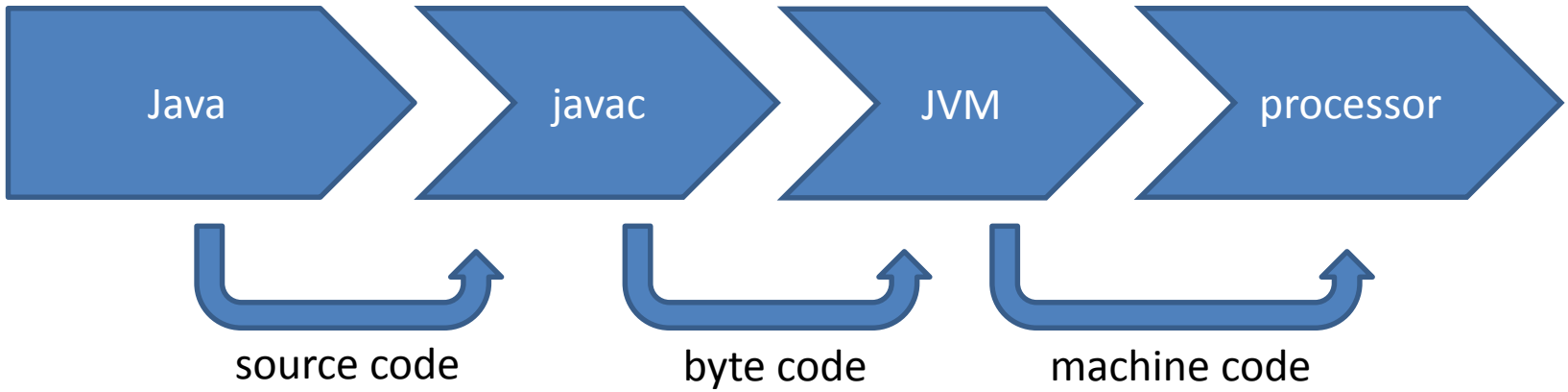
`http://www.cs.umd.edu/~pugh/java/memoryModel`

That web site and mailing list will continue to be updated and maintained, and further updates and expansion of non-normative text intended to help people understand the JSR-133 specification will be available from that web site.

Two changes from the original specification in the JLS are most likely to require that JVM™ implementations be changed:

- The semantics of volatile variables have been strengthened to have acquire and release semantics. In the original specification, accesses to volatile and non-volatile variables could be freely ordered.
- The semantics of final fields have been strengthened to allow for thread-safe immutatability without explicit synchronization. This may require steps such as store-store barriers at the end of constructors in which final fields are set.

How is Java code executed?



Optimizations are applied almost exclusively after handing responsibility to the JVM's runtime where they are **difficult to comprehend**.

A JVM is allowed to alter the executed program as long as it remains correct. The Java memory model **describes a contract** for what a correct program is (in the context of multi-threaded applications).

The degree of optimization is dependent on the **current compilation stage**.

A sequentially inconsistent optimization

```
void method() {
  foo += 1;
  bar += 1;

  foo += 2;
}
```

optimization

```
void method() {
  foo += 1;
  foo += 2;

  bar += 1;
}
```

optimization

```
void method() {
  foo += 3;

  bar += 1;
}
```

```
(foo == 0, bar == 0)
(foo == 1, bar == 0)
(foo == 1, bar == 1)
(foo == 3, bar == 1)
```

```
(foo == 0, bar == 0)
(foo == 1, bar == 0)
(foo == 3, bar == 0)
(foo == 3, bar == 1)
```

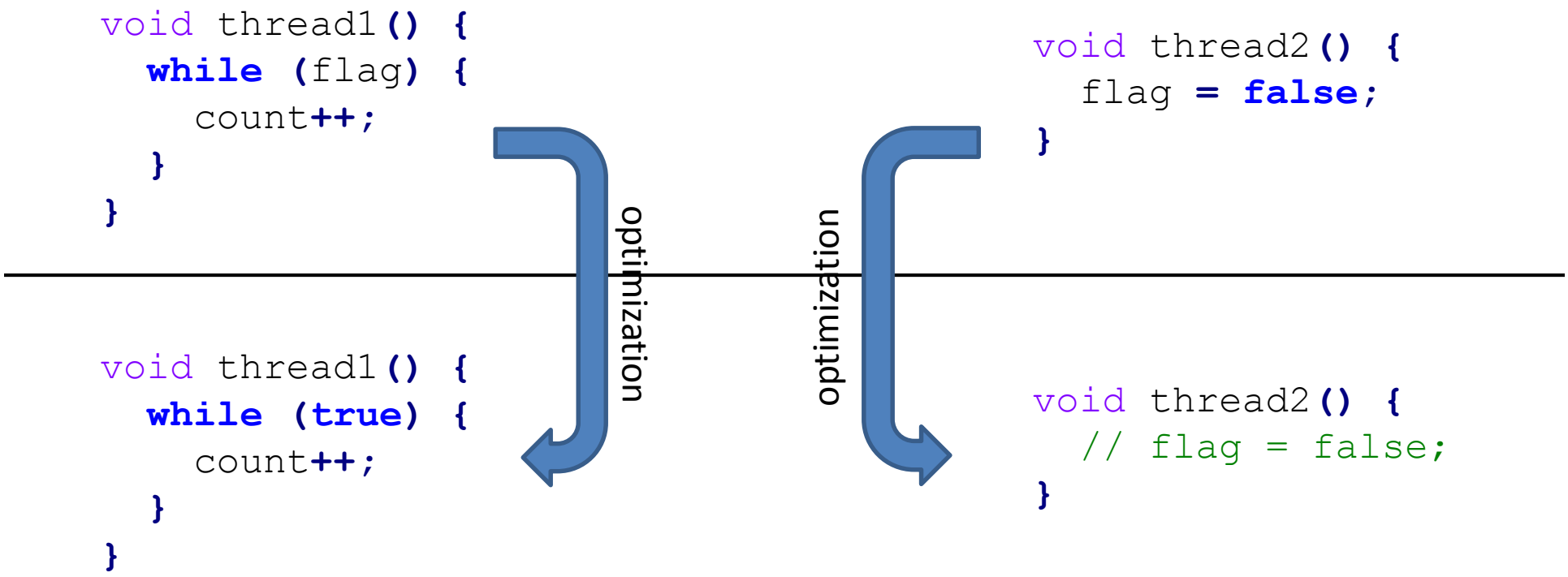
```
(foo == 0, bar == 0)
(foo == 3, bar == 0)
(foo == 3, bar == 1)
```

Scaling performance: cache efficiency does matter

action	approximate time (ns)
typical processor instruction	1
fetch from L1 cache	0.5
branch misprediction	5
fetch from L2 cache	7
mutex lock/unlock	25
fetch from main memory	100
2 kB via 1 GB/s	20.000
seek for new disk location	8.000.000
read 1 MB sequentially from disk	20.000.000

Source: https://gist.github.com/jboner/2841832

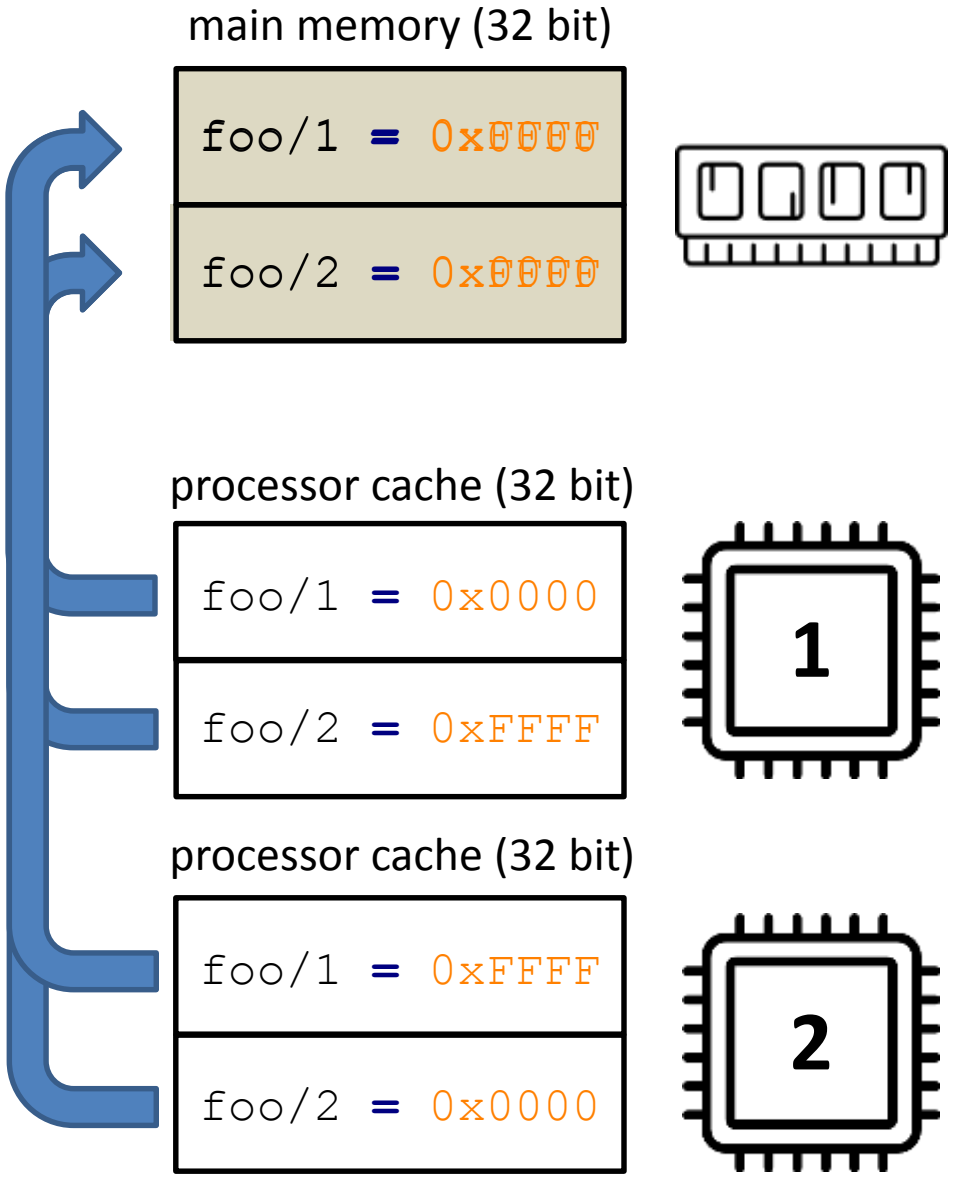
An eventually inconsistent optimization



Mnemonic: Think of each thread as if it owned its own heap (infinitive caches).

Atomicity

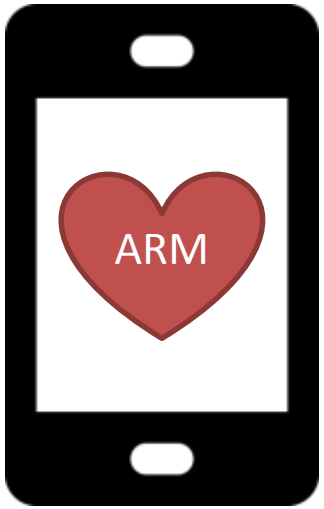
```
class WordTearing {

  long foo = 0L;

  void thread1() {
    foo = 0x0000FFFF;
    //  = 2147483647
  }

  void thread2() {
    foo = 0xFFFF0000;
    //  = -2147483648
  }
}
```

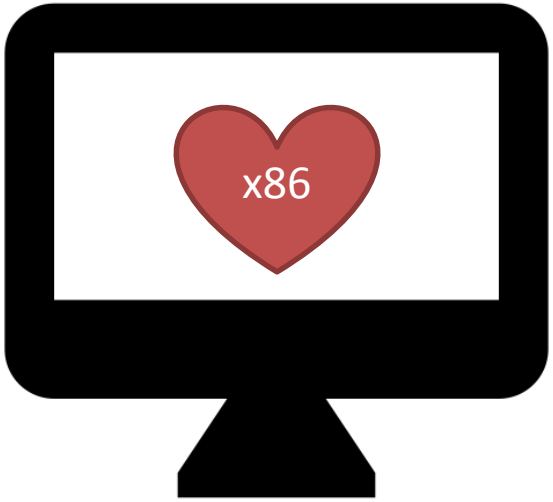


Processor optimization: a question of hardware architecture

	ARM	PowerPC	SPARC TSO	x86	AMD64
load-load	yes	yes	no	no	no
load-store	yes	yes	no	no	no
store-store	yes	yes	no	no	no
store-load	yes	yes	yes	yes	yes

Source: Wikipedia





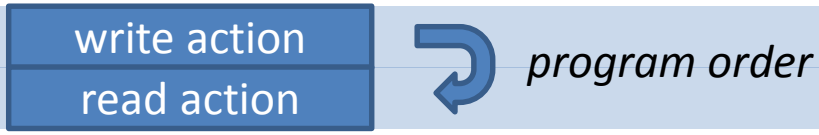
What is the Java memory model?

Answers: **what values can be observed upon reading from a specific field**.

Formally specified by disaggregating a Java program into **actions** and applying several **orderings** to these actions. If one can derive a so-called **happens-before** ordering between **write actions** and a **read** actions of one field, the Java memory model guarantees that the read returns a particular value.

A trivial, single-threaded example:

```
class SingleThreaded {

  int foo = 0;

  void method() {
    foo = 1;
    assert foo == 1;
  }
}
```



The JMM guarantees *intra-thread consistency* resembling sequential consistency.

Java memory model building-blocks

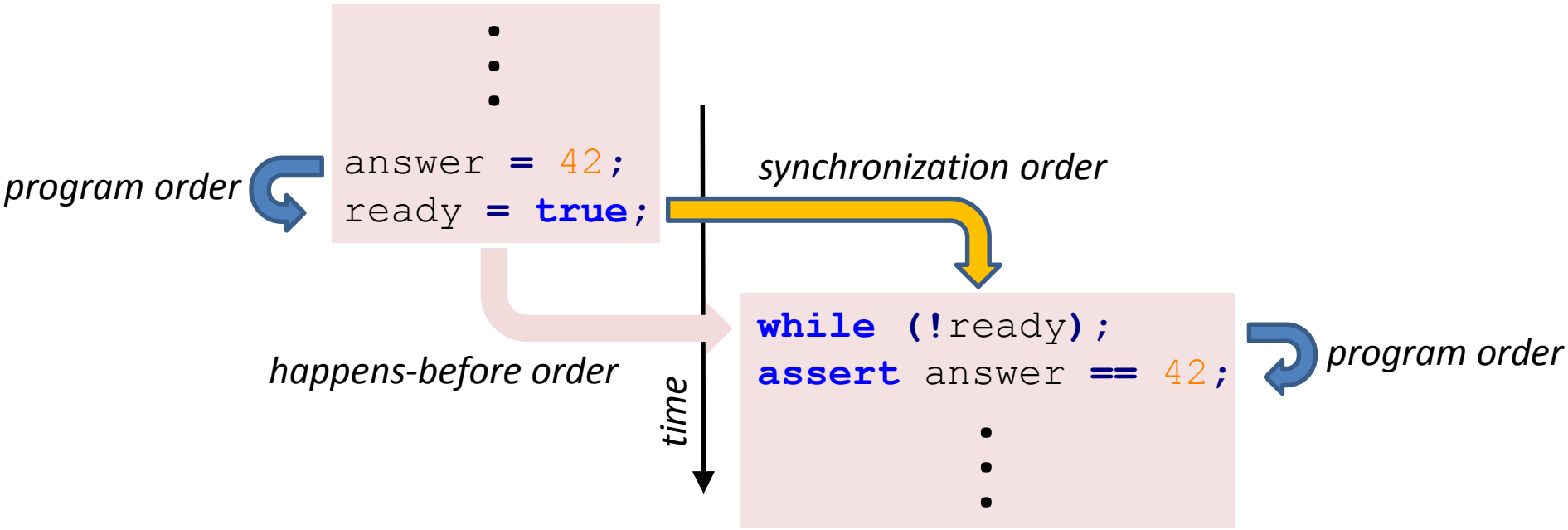
field-scoped	*method-scoped*
`final`	`synchronized` (method/block)
`volatile`	`java.util.concurrent.locks.Lock`

Using the above keywords, a programmer can indicate that a JVM should **refrain from optimizations** that could otherwise cause concurrency issues.

In terms of the Java memory model, the above concepts introduce additional **synchronization actions** which introduce additional (partial) **orders**. Without such modifiers, reads and writes might not be ordered (weak memory model) what results in a **data race**.

A memory model is a **trade-off** between a language's simplicity (consistency/atomicity) and its performance.

Volatile field semantics: reordering restrictions

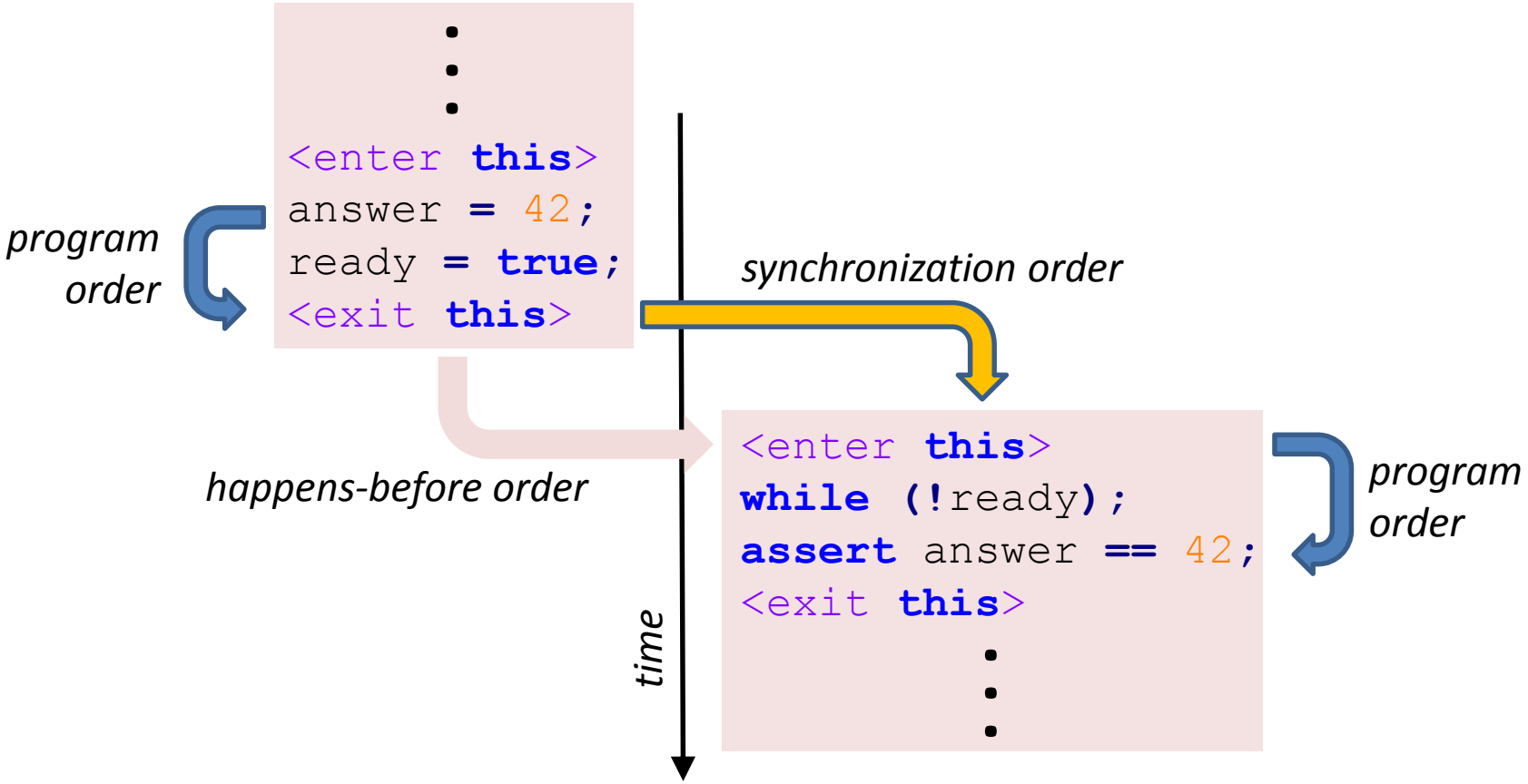


1. When a thread **writes** to a volatile variable, all of its previous writes are guarantted to be visible to another thread when that thread is **reading** the same value.
2. Both threads must align "their" volatile value with that in **main memory**.
3. If the volatile value was a `long` or a `double` value, **word-tearing** was forbidden.

This only applies for two threads with a *write-read relationship* on **the same** field!

Important: the `synchronized` keyword also implies an synchronization order. Synchronization order is however not exclusive to it (as demonstrated here)!

Synchronized block semantics: reordering restrictions



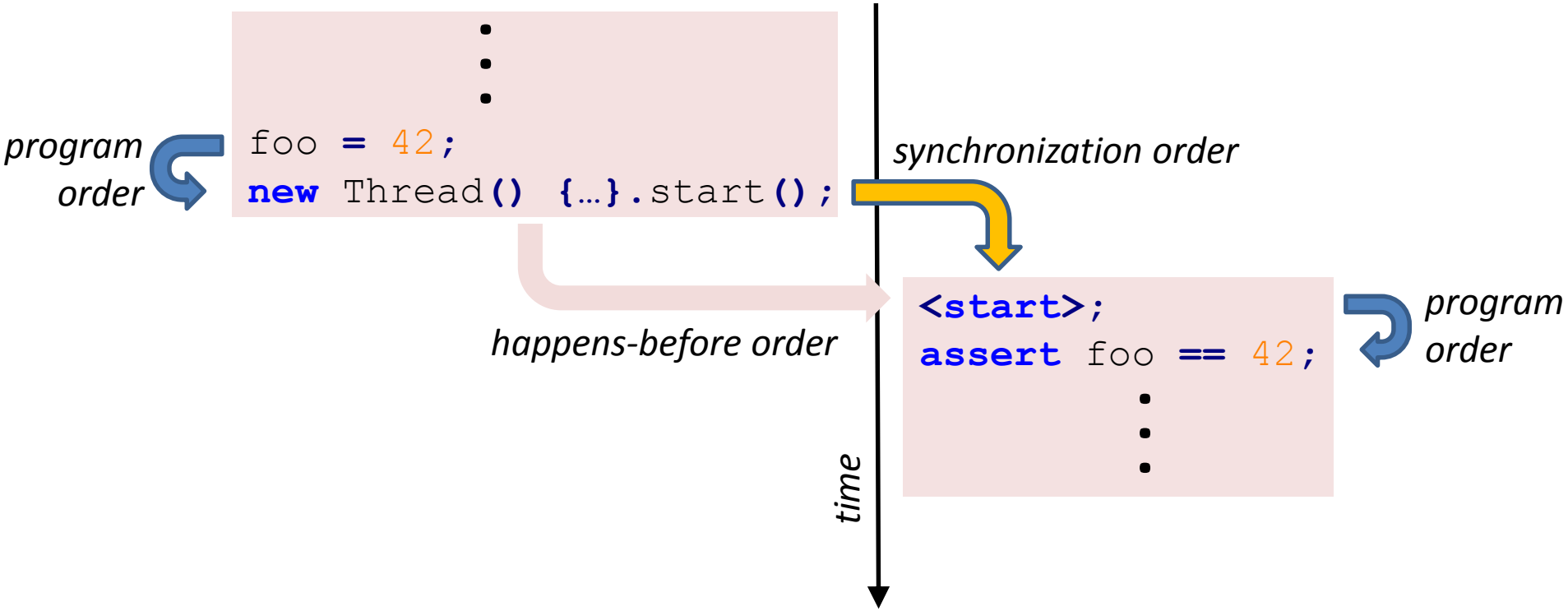
This example assumes that the second thread acquires the monitor lock first.

When a thread **releases** a monitor, all of its previous writes are guaranteed to be visible to another thread after that thread is **locking** the same monitor.
This only applies for two threads with a *unlock-lock relationship* on **the same** monitor!

Thread life-cycle semantics

```
class ThreadLifeCycle {

  int foo = 0;

  void method() {
    foo = 42;
    new Thread() {
      @Override
      public void run() {
        assert foo == 42;
      }
    }.start();
  }
}
```

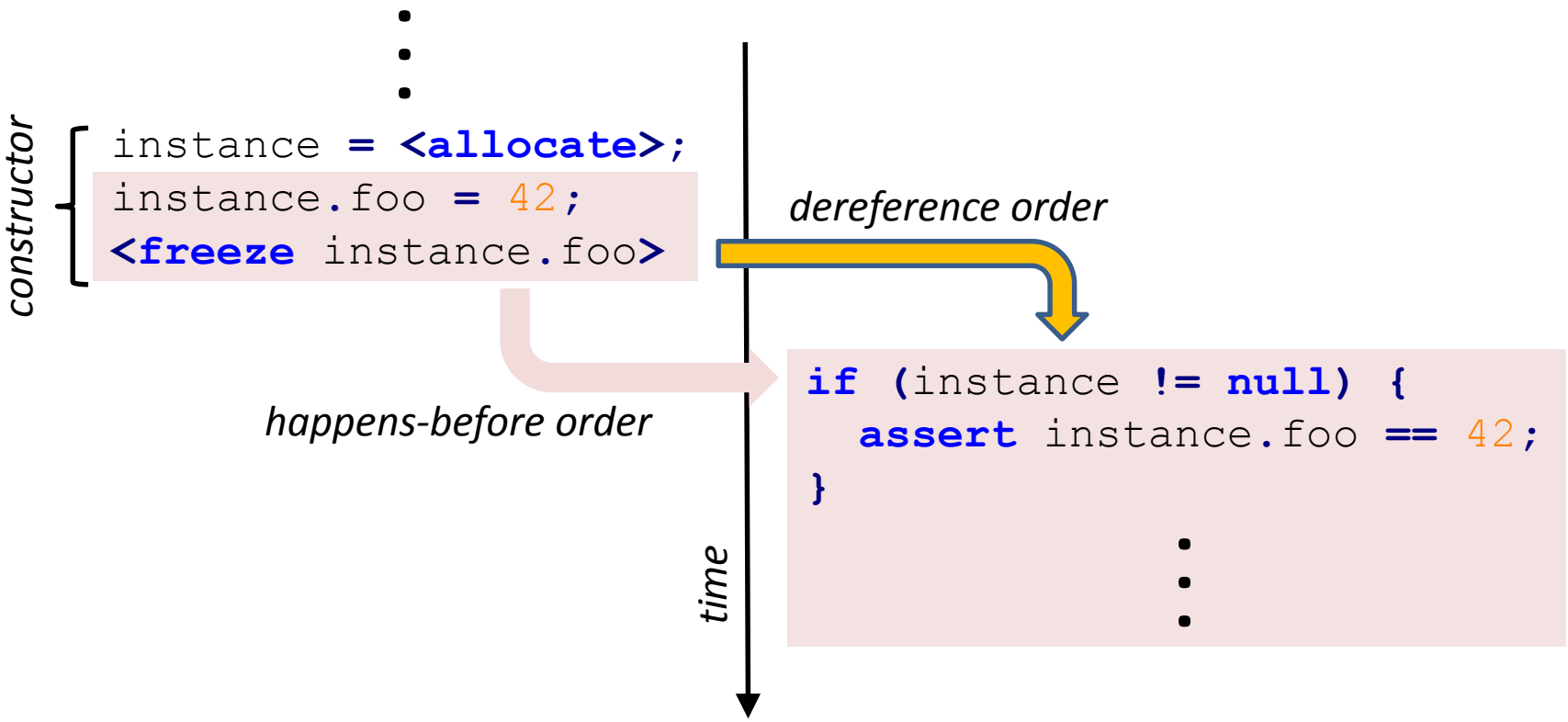
Thread life-cycle semantics: reordering restrictions



When a thread starts another thread, the started thread is guaranteed to see all values that were **set by the starting thread**.

Similarly, a thread that **joins another thread** is guaranteed to see all values that were set by the joined thread.

Final field semantics: reordering restrictions



When a thread creates an instance, the instance's final fields are **frozen**. The Java memory model requires **a field's initial value** to be visible in the initialized form to other threads.

This requirement also holds for properties that are **dereferenced via a final field**, even if the field value's properties are not final themselves (*memory-chain order*).

Does **not apply for (reflective) changes** outside of a constructor / class initializer.

External actions

```
class Externalization {

  int foo = 0;

  void method() {
    foo = 42;
    jni();
  }

  native void jni(); /* {
    assert foo == 42;
  } */
}
```

program order

A JIT-compiler cannot determine the **side-effects** of a native operation. Therefore, **external actions** are guaranteed to not be reordered.

External actions include JNI, socket communication, file system operations or interaction with the console (non-exclusive list).

Thread-divergence actions

```
class ThreadDivergence {

  int foo = 42;

  void thread1() {
    while (true);
    foo = 0;
  }

  void thread2() {
    assert foo == 42;
  }
}
```

program order

Thread-divergence actions are guaranteed to not be reordered. This prevents surprising outcomes of actions that might **never be reached**.

Memory model implementation

A Java virtual machine typically implements **a stricter form** of the Java memory model for pragmatic reasons.

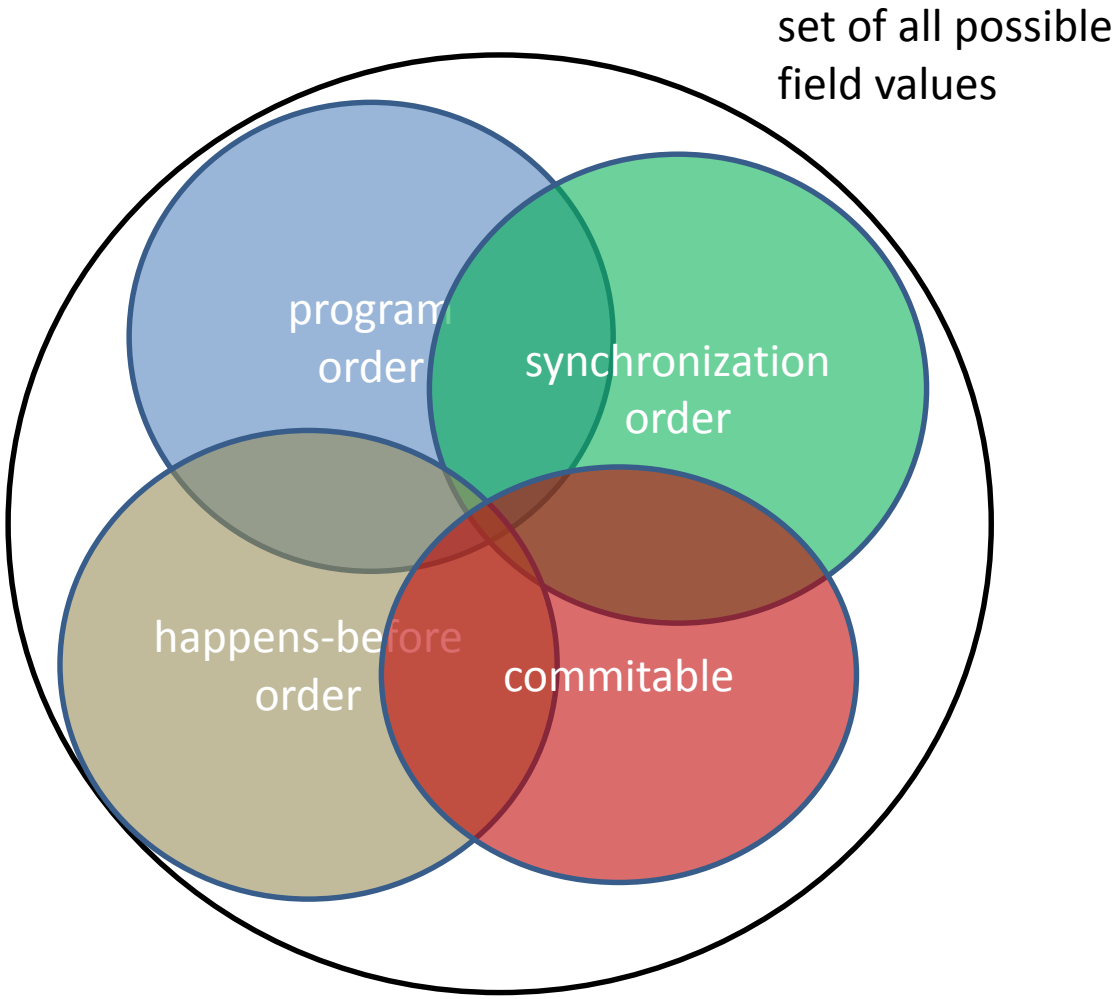
For example, the HotSpot virtual machine issues **memory barriers** after synchronization points. These barriers forbid certain types of memory reordering (load-load, load-store, store-load, store-store).

Relying on such implementation details jeopardizes **cross-platform compatibility**.

~~`synchronized (new Object()) { /* empty */ }`~~

Always code against the specification, not the implementation!

Memory model validation: the academic approach



The transitive closure of all orders determines the set of legal outcomes.

Theory deep dive: "Java Memory Model Pragmatics" by Aleksey Shipilëv

Memory model validation: the pragmatic approach

```
@JCStressTest
@State
class DataRaceTest {

  boolean ready = false;
  int answer = 0;

  @Actor
  void thread1(IntResult1 r) {
    while (!ready);
    r.r1 = answer;
  }

  @Actor
  void thread2() {
    answer = 42;
    ready = true;
  }
}
```

Important limitations:

1. Not a unit test. The outcome is **non-deterministic**.
2. Does not prove correctness, **might** discover incorrectness.
3. Result is **hardware-dependent**.

Other tools:

Concurrency unit-testing frameworks such as *thread-weaver* offer the introduction of an explicit execution order for concurrent code. This is achieved by instrumenting a class's code to include explicit break points which cause synchronization. These tools cannot help with the discovery of synchronization errors.

User space code in Python

```
# Python program to illustrate the concept
# of threading
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":

    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))

    # print name of main thread
    print("Main thread name: {}".format(threading.main_thread().name))

    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')

    # starting threads
    t1.start()
    t2.start()

    # wait until all threads finish
    t1.join()
    t2.join()
```

Run on IDE

```
ID of process running main program: 11758
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 11758
Task 2 assigned to thread: t2
ID of process running task 2: 11758
```

Python Interpreter internally calls pthread APIs

https://github.com/enthought/Python-2.7.3/blob/master/Python/thread_pthread.h
https://github.com/python/cpython/blob/master/Python/thread.c

Branch: master ▾ Python-2.7.3 / Python / thread_pthread.h Find file | Copy path

cournape Python 2.7.3. 69fe0ff on 21 Dec 2013

1 contributor

506 lines (420 sloc) | 13 KB Raw | Blame | History

```
1
2   /* Posix threads interface */
3
4   #include <stdlib.h>
5   #include <string.h>
6   #if defined(__APPLE__) || defined(HAVE_PTHREAD_DESTRUCTOR)
7   #define destructor xxdestructor
8   #endif
9   #include <pthread.h>
```

```
33  /* for safety, ensure a viable minimum stacksize */
34  #define THREAD_STACK_MIN        0x8000  /* 32kB */
```

```
159 long
160 PyThread_start_new_thread(void (*func)(void *), void *arg)
161 {
162     pthread_t th;
163     int status;
164 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
165     pthread_attr_t attrs;
166 #endif
167 #if defined(THREAD_STACK_SIZE)
168     size_t      tss;
169 #endif
170
171     dprintf(("PyThread_start_new_thread called\n"));
172     if (!initialized)
173         PyThread_init_thread();
174
175 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
176     if (pthread_attr_init(&attrs) != 0)
177         return -1;
178 #endif
179 #if defined(THREAD_STACK_SIZE)
180     tss = (_pythread_stacksize != 0) ? _pythread_stacksize
181                                      : THREAD_STACK_SIZE;
182     if (tss != 0) {
183         if (pthread_attr_setstacksize(&attrs, tss) != 0) {
184             pthread_attr_destroy(&attrs);
185             return -1;
186         }
187     }
188 #endif
189 #if defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
190     pthread_attr_setscope(&attrs, PTHREAD_SCOPE_SYSTEM);
191 #endif
192
193     status = pthread_create(&th,
194 #if defined(THREAD_STACK_SIZE) || defined(PTHREAD_SYSTEM_SCHED_SUPPORTED)
195                             &attrs,
196 #else
197                             (pthread_attr_t*)NULL,
198 #endif
199                             (void* (*)(void *))func,
200                             (void *)arg
201                             );
```

User space code in Java, JVM internally calls pthread APIs

```java

public class Thread {
   static AtomicInteger threadCount = new AtomicInteger(1);

   public void run() {
     System.out.println("Running Thread " + threadCount.getAndIncrement());
   }

   public void start() {
       start0();
   }
   private native void start0();
}
```

```cpp
JNIEXPORT void JNICALL Java_com_threading_Thread_start0(JNIEnv *env, jobject javaThreadObjectRef)
{
    //Get jvm instance and global reference to Thread java object to be passed to
    //pthread entry point function.
    JavaThreadWrapper* args = new JavaThreadWrapper(env, javaThreadObjectRef);

    //init thread attributes
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    //native thread id
    pthread_t tid;
    if (pthread_create(&tid, &attr, thread_entry_point, args))
    {
        fprintf(stderr, "Error creating thread\n");
        return;
    }

    std::cout << "Started a linux thread " << tid << "!" << endl;
    return;
}
```

Inside the Python GIL

David Beazley
http://www.dabeaz.com

June 11, 2009 @ chipy

Originally presented at my "Python Concurrency Workshop", May 14-15, 2009 (Chicago)

Video Presentation

You can watch the video of this presentation here:

http://blip.tv/file/2232410

It expands upon the slides and is recommended.

A Performance Experiment

- Consider this trivial CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Run it twice in series

```
count(100000000)
count(100000000)
```

- Now, run it in parallel in two threads

```
t1 = Thread(target=count,args=(100000000,))
t1.start()
t2 = Thread(target=count,args=(100000000,))
t2.start()
t1.join(); t2.join()
```


A Mystery

- Why do I get these performance results on my Dual-Core MacBook?

  Sequential : 24.6s
  Threaded : 45.5s (1.8X slower!)

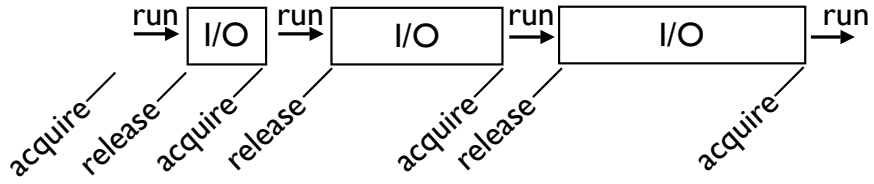
- And if I disable one of the CPU cores, why does the threaded performance get better?

  Threaded : 38.0s

- Think about that for a minute... Bloody hell!

GIL Behavior

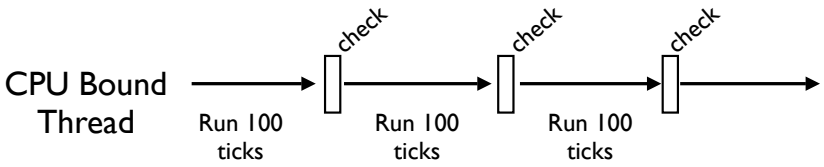
- It's simple : threads hold the GIL when running
- However, they release it when blocking for I/O



- So, any time a thread is forced to wait, other "ready" threads get their chance to run
- Basically a kind of "cooperative" multitasking

CPU Bound Processing

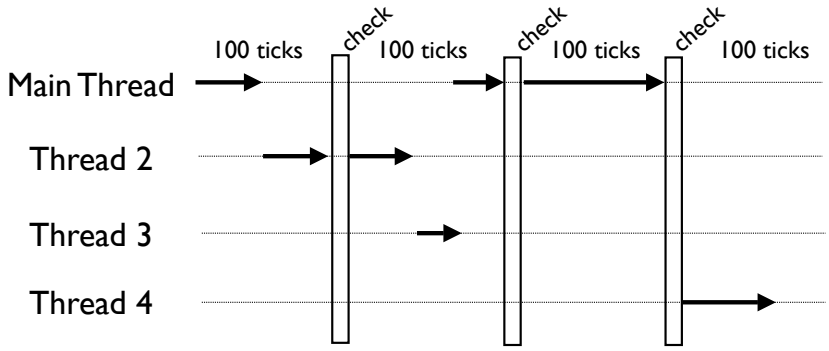
- To deal with CPU-bound threads that never perform any I/O, the interpreter periodically performs a "check"
- By default, every 100 interpreter "ticks"



- sys.setcheckinterval() changes the setting

The Check Interval

- The check interval is a global counter that is completely independent of thread scheduling



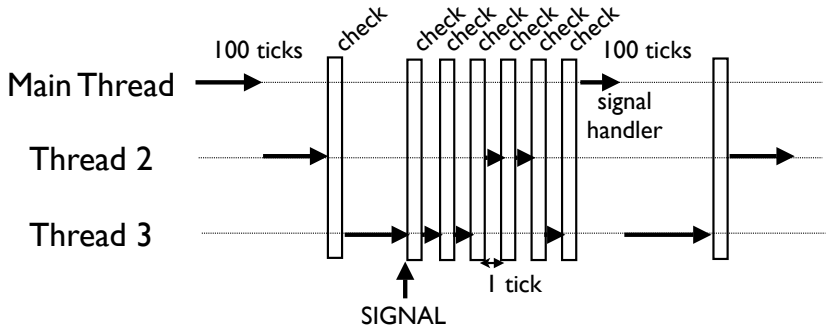
- A "check" is simply made every 100 "ticks"

The Periodic Check

- What happens during the periodic check?
  - In the main thread only, signal handlers will execute if there are any pending signals (more shortly)
  - Release and reacquire the GIL
- That last bullet describes how multiple CPU-bound threads get to run (by briefly releasing the GIL, other threads get a chance to run).

Signal Handling

- If a signal arrives, the interpreter runs the "check" after every tick until the main thread runs



- Since signal handlers can only run in the main thread, the interpreter quickly acquires/releases the GIL after every tick until it gets scheduled

Thread Scheduling

- Python does not have a thread scheduler
- There is no notion of thread priorities, preemption, round-robin scheduling, etc.
- All thread scheduling is left to the host operating system (e.g., Linux, Windows, etc.)
- This is partly why signals get so weird (the interpreter has no control over scheduling so it just attempts to thread switch as fast as possible with the hope that main will run)

CPU-Bound Threads

- As we saw earlier, CPU-bound threads have horrible performance properties
- Far worse than simple sequential execution
  - 24.6 seconds (sequential)
  - 45.5 seconds (2 threads)
- A big question : Why?
  - What is the source of that overhead?

Signaling Overhead

- GIL thread signaling is the source of that
- After every 100 ticks, the interpreter
  - Locks a mutex
  - Signals on a condition variable/semaphore where another thread is always waiting
  - Because another thread is waiting, extra pthreads processing and system calls get triggered to deliver the signal

A Rough Measurement

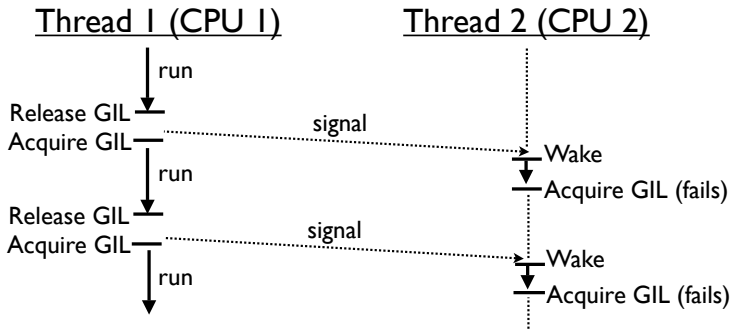
- Sequential Execution (OS-X, 1 CPU)
  - 736 Unix system calls
  - 117 Mach System Calls
- Two CPU-bound threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~ 3.3 Million Mach System Calls
- Yow! Look at that last figure.

Multiple CPU Cores

- The penalty gets far worse on multiple cores
- Two CPU-bound threads (OS-X, 1 CPU)
  - 1149 Unix system calls
  - ~3.3 Million Mach System Calls
- Two CPU-bound threads (OS-X, 2 CPUs)
  - 1149 Unix system calls
  - ~9.5 Million Mach System calls

Multicore GIL Contention

- With multiple cores, CPU-bound threads get scheduled simultaneously (on different cores) and then have a GIL battle



- The waiting thread (T2) may make 100s of failed GIL acquisitions before any success

The GIL Battle (Traced)

```
t2 100 5392 ENTRY
t2 100 5392 ACQUIRE
t2 100 5393 RELEASE
t1 100 5393 ACQUIRE
t2 100 5393 ENTRY
t2  27 5393 BUSY
t1 100 5394 RELEASE
t1 100 5394 ENTRY
t1 100 5394 ACQUIRE
t2  74 5394 RETRY
t1 100 5395 RELEASE
t1 100 5395 ENTRY
t1 100 5395 ACQUIRE
t2  83 5395 RETRY
t1 100 5396 RELEASE
t1 100 5396 ENTRY
t1 100 5396 ACQUIRE
t2  80 5396 RETRY
t1 100 5397 RELEASE
t1 100 5397 ENTRY
t1 100 5397 ACQUIRE
t2  79 5397 RETRY
...
```

signal

signal

signal

signal

A thread switch

t2 tries to keep running, but immediately has to block because t1 acquired the GIL

Here, the GIL battle begins. Every RELEASE of the GIL signals t2. Since there are two cores, the OS schedules t2, but leaves t1 running on the other core. Since t1 is left running, it immediately reacquires the GIL before t2 can get to it (so, t2 wakes up, finds the GIL is in use, and blocks again)