JVM at Google

Jeremy Manson

Google

Java at Google

- Large number of 0s of Java devs/code
  - What can JDK/JVM level technologies to do help?
  - At the very least, we can support them when things go wrong
  - And hopefully, we find time to do something that they notice, too
- Built a JDK team at Google
  - Deploy, maintain and enhance JDK/JVM
  - Used by Gmail, Google+, Docs, Blogger, Build system, AdWords…
  - And many, many others!

We have to do everything

- Best playground a language enthusiast could want
  - But got to keep the engines going…
  - No matter what needs doing, we do it!

- In that spirit, I will clumsily lurch from topic to topic
  - This closely resembles my weekdays

- Last year: Static analysis, monitoring, GC
- This year: Native code, threading, static analysis (maybe)

Native Code Interoperation

C++ and Java: Why do we care?

- Lots of talks about JNI / JNA / JNR / Packed Arrays...
  - Let's talk about the whys.
  - Goes beyond libc / syscalls
- Performance / predictability **can** be an issue
  - Hey, maybe you need a 2^32 array
- Infrastructure often in C++, frontend / business logic in Java
  - Native code is a lingua franca
  - If you need code shared across Go and Python and Java, you write it in C/C++

Obvious Engineering-Level Challenges

- Mostly covered in the FFI workshop yesterday
- Data layout is different
- Object lifetime in Java is a dubious notion
- There is no such thing as a pointer in Java
- JNI is slow
- Mismatches between memory models

- Project Panama is there to help us (hopefully)

Google

Lots and lots of workflow pain points

- Different assumptions about runtime environment
    - How do you install a signal handler? What do TIDs look like? `malloc`?
- Different best practices
    - C++ users stop their applications on error
- Debugging is painful (mixed stacks, core files…)
- Monitoring is painful
    - Even hard to explain to users why Java and native heap are different!
- Communities are very wary of each others’ languages
- Automatic wrapping state of the art is SWIG
    - ...which doesn’t really understand C++

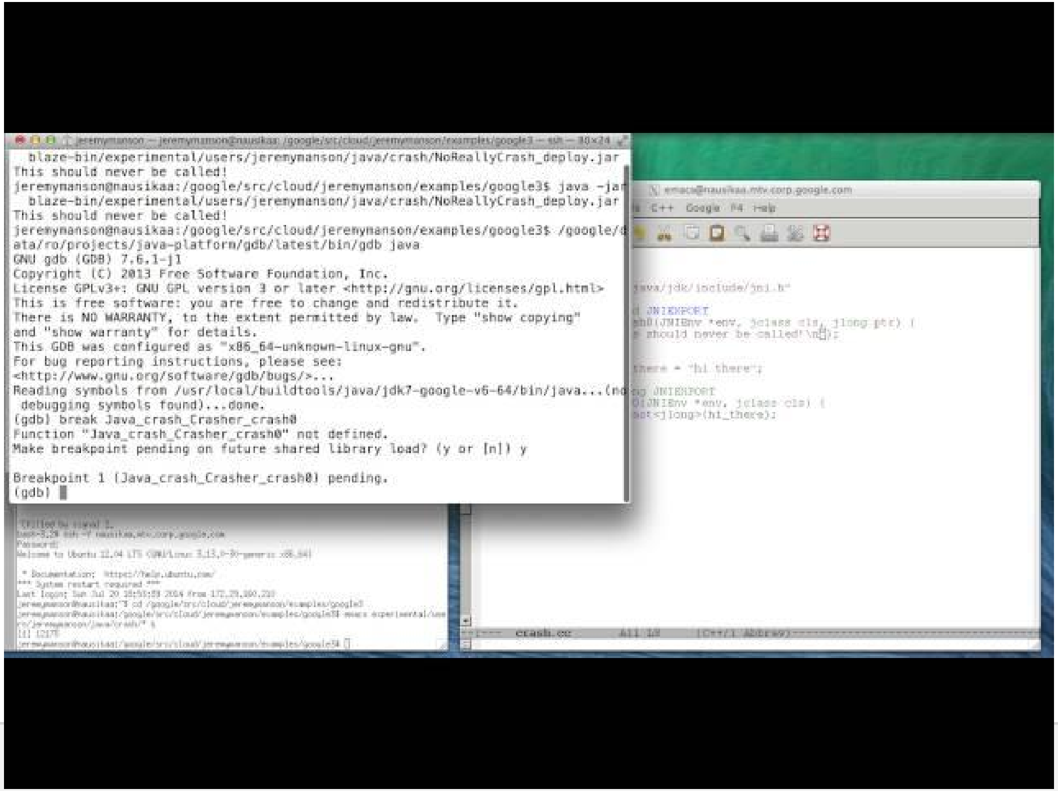
What do we do?

- Many users have separate C++ / Java applications; talk via RPC
- Use existing technology to aid in production / deployment
  - Heavily reliant on Launchers / Invocation API, JEP 178-alike
- Adjust our tooling to deal with the fact that mixed mode is painful…
  - Debugging
  - Monitoring

Debugging

- State of the art - attach two debuggers, flip between them

This is a clickable link to a YouTube video:



Google

Dynamic Analysis

- Our performance analysis tools have to understand both
  - Distinguish between Java and native heap analysis
  - Produce CPU profiles unified across both
  - Adjusted various stack trace mechanisms in JVM to provide mixed-mode stack traces
  - To track heap usage, need to instrument malloc/free and Java heap allocation
- Our valgrind-alike has to work with JNI
  - All modules need to use its instrumented malloc / free
  - … but JVM has lots of memory leaks
  - http://clang.llvm.org/docs/AddressSanitizer.html

Dynamic Analysis: Data Race Detection

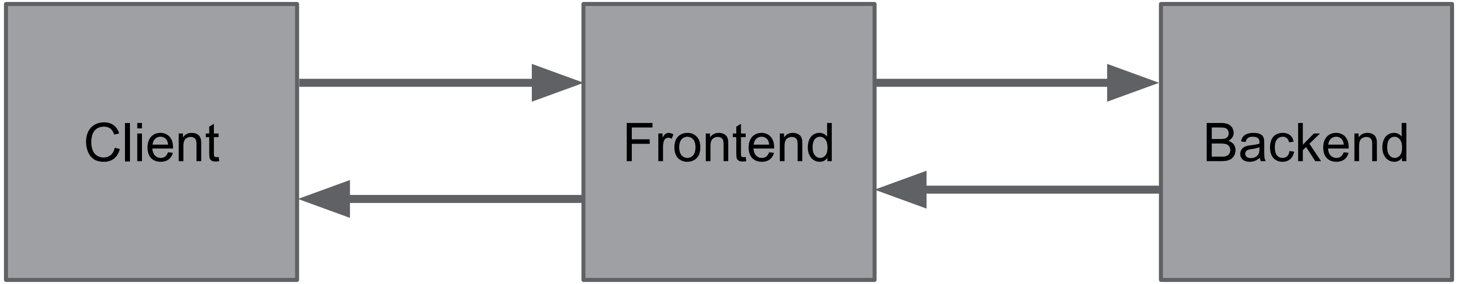
- Have a data race detector for native code
  - https://code.google.com/p/thread-sanitizer/
- What happens when synchronization for native code is done in Java?
  - Finalizers are *very* often used to free native memory
  - Java locks are used to protect native memory
- Need to make tools aware of each other…
  - Working on integration
- Starting down this path towards a complete data race checker
  - Hopefully, we'll be talking about that some other time

But really, I just wanted to talk about...

Threads

Very experimental
Will stop abruptly at where we are today

What Kind of Programs do we Care About?



How do Servers Handle Requests: Synchronous

Every time a request comes in, spawn a thread to deal with it.

- If the thread has to do more I/O, it blocks.
- Referred to as "synchronous"

```
Result handler(Request req) {
  Result a = rpc(req.id);
  Result b = rpc(a);
  return b;
}
```

How do Servers Handle Requests: Synchronous

Pros:

- Straight line code, ultra simple
- Good locality

Cons:

- In large servers, spawns **a lot** of threads
- With a lot of (unpredictable) contention and context switching
- And it turns out that thread scheduling is **really** expensive
- And a lot of thread stack usage
- Harder to parallelize individual requests

How do servers handle requests: Asynchronous

- Every time a request comes in, dispatch to a thread pool
  - If server needs to block on I/O, register a callback that is run when blocking operation is done

```
void handler(Request req, Response res) {
  rpc(req.id, (Result a) ->
      rpc(a, (Result b) ->
          res.finish(b)));
}
```

How do servers handle requests: Asynchronous

- Pros:
  - Scales **really** well - can have ~1 thread per core
- Cons:
  - Need a state machine to handle a request
  - Debuggers stink: a stack trace doesn’t tell you anything, can’t walk through code
  - Non-multithreaded requests are now multithreaded
  - Lousy locality - resources are smeared across threads

This is really awful!

- Want simplicity of synchronous, performance of asynchronous
- Can’t the language do something complicated to take care of it?

- Well, sure, lots of programming languages have solved it.
- Write the code synchronously
- Instead of blocking and letting the OS decide what to schedule, explicitly transfer control to something else.
    - Basically, take yield / coroutines / call/cc and turn them multithreaded
    - Green threads / Goroutines / Fibers, it’s all the same stuff
- Add some form of user scheduling to that
    - Probably involving queues / channels

This is usually a big win

- Less memory usage from threads
  - ~1M stacks * 10K pending requests == a lot
- Pass through the OS scheduler less
  - 10K threads * trying to schedule stuff == high variance
- If the user scheduler is careful, better locality
  - Network thread communicates over a socket
  - Passes directly to the thread that owns the socket

Approaches

- ~80 bazillion prior JVMLS talks on continuations
- When you block, save state, switch to something else
- Need some user code that tells you what to switch to.
- There is a thread management component

- Could use bytecode rewriting to break your code around statements that might block
  - Doesn’t work for non-Java code
  - Wait for it...

Build support into JVM?

- Without language support, have save() and resume() API
  - enter() means start-the-bit-you-might-save
  - save() saves everything since enter()
  - resume() resumes it (maybe passing back a param)
  - Instrument blocking calls either via rewriting or by hand

- See Hiroshi Yamauchi's 2010 JVMLS talk

  https://wikis.oracle.com/display/mlvm/StackContinuations

Build support into JVM?

Pros:

- Debugging is better - Java stacks make sense
- Memory consumption couple of orders of magnitude better
  - 10K threads == 1.2G RSS 10K continuations = 30M RSS
- Performance comparable with async
  - A couple of percent off with deeper stacks, attributable to the experimental nature of patch

Cons:

- Still doesn’t work for non-Java code.
  - Have to instrument park / unpark, epoll, everything that blocks

Go Deeper?

- Do it in the kernel
  - All you need to do is swap out a bunch of registers
  - If you know what you want to schedule next, no scheduler overhead at all.

- Very simple API, with just three operations:
  - switchto_wait(): gives up control
  - switchto_resume(tid): resume tid
  - switchto_switch(tid): transfer control to tid

Go Deeper?

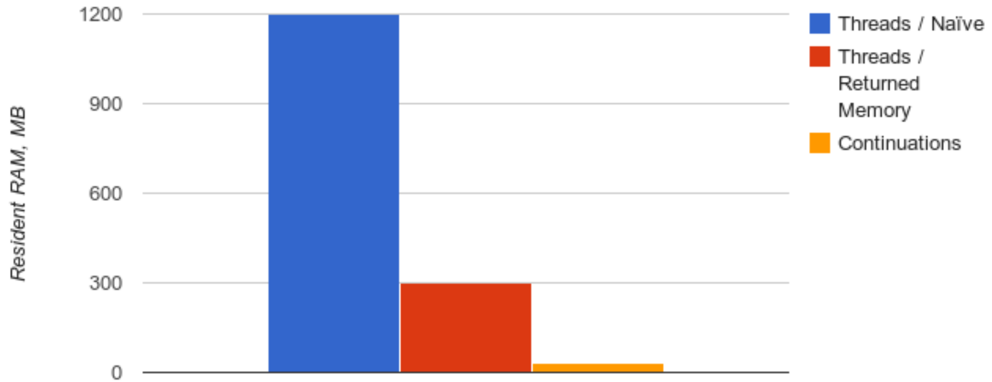
- Works for JNI
  - Don’t need to instrument JVM’s blocking operations anymore
  - Native thread identity / tids are maintained

- Context switches for handoffs are now ~150-200ns
  - Kernel call is 10s of ns
  - Scheduling is expensive and unnecessary with switch()
  - Don’t have two-step 1) wakeup other thread, 2) go to sleep
    - Which means other thread is usually scheduled on other CPU

Go Deeper?

- Debugging is still good
  - A thread stack is a thread stack
  - Existing code “just works”
- Locality is nicer
  - Can switchto the context that will need the resources you are using (e.g., socket)
- Don’t need a nanny scheduler
  - (BTW, this is what makes it different from Windows UMS)
- What about thread stacks?

Hmm… What **about** thread stacks?



- Return RAM to system forcibly
- 10K threads; 1 thread-per-core, 10K continuations
- Okay, but still not great

Future Work: Make this make sense

- Not a solution for everyone (unless Linux picks up switchto patch)
- Easy to get comparable performance:
    - Have a dedicated green thread
    - Swap out registers on demand
- Cactus stack could improve memory situation
- Also, language support would be nice
- This is a 20% project for me

- Now I’m stopping abruptly.
- No great morals to be found here. Questions?

Static Analysis Update

Google

Error Prone Update

- Our easy-to-extend static analysis bug checker
  - Works at compile time, easy to integrate into build systems
  - **Really** careful about error messages and false positive rate
  - Easy to write new checks
  - No dependencies on particular IDEs
  - Can write tools that pass over entire code bases and report problems easily

- Coming features:
  - Checking @GuardedBy
  - Dataflow analysis extensions in progress

Error Prone is pretty easy

Built on top of Java AST matching API:

```
@SuppressWarnings("unchecked")
private static final Matcher<MethodInvocationTree> instanceEqualsMatcher = Matchers.allOf(
    methodSelect(instanceMethod(Matchers.<ExpressionTree>isArrayType(), "equals")),
    argument(0, Matchers.<ExpressionTree>isArrayType()));
```

```
Fix fix = SuggestedFix.builder()
    .replace(t, "Arrays.equals(" + arg1 + ", " + arg2 + ")")
    .addImport("java.util.Arrays")
    .build();
```

But you can never be too easy!

Refaster

A scalable, example-based analysis tool - built on top of error-prone:

```
static class ToCharArrayIndex {
  @BeforeTemplate public char toCharArrayAt(String str, int index) {
    return str.toCharArray()[index];
  }
  @AfterTemplate public char charAt(String str, int index) {
    return str.charAt(index);
  }
}
```

Can be used for writing checks, doing refactoring, automating code reviews...

I know I promised, but...

Shows up in code review...

```
package com.google.common.collect;

class Refaster {
  public static void testMethod() {
    char c = "foo".toCharArray()[0];
```

▾ JavaOptionalSuggestions
ToCharArrayIndex
3:39 PM

Unnecessary array copy.

Suggested replacement:
"foo".charAt(0)
go/klippy

Suggested fix attached: show

Even more cool...

Hit the “show” button and...

☐ Show original fix

//depot/google3/java/com/google/common/collect/Refaster.java

```java
package com.google.common.collect;

class Refaster {
  public static void testMethod() {
    char c = "foo".toCharArray()[0];
  }
}
```

```java
package com.google.common.collect;

class Refaster {
  public static void testMethod() {
    char c = "foo".charAt(0);
  }
}
```

Apply Cancel

About to change subjects abruptly...

Questions?

Links:
Error Prone: https://code.google.com/p/error-prone/
Refaster (work in progress): https://github.com/google/Refaster

All Done!