WiX Engineering

Async vs. Sync Programming

Dmitry (Mitya) Komanov

dmitryk@wix.com | @dkomanov | linkedin/dkomanov | github.com/dkomanov

01

Terms

Synchronous (or blocking)

- Once blocking method called, calling thread will wait.
- Price: context switch

Asynchronous (or non-blocking)

- Thread continues to work after call to async method. Result of async method execution may be acquired via callback or via polling.
- Price: queue, poll, await.

02

That’s it?

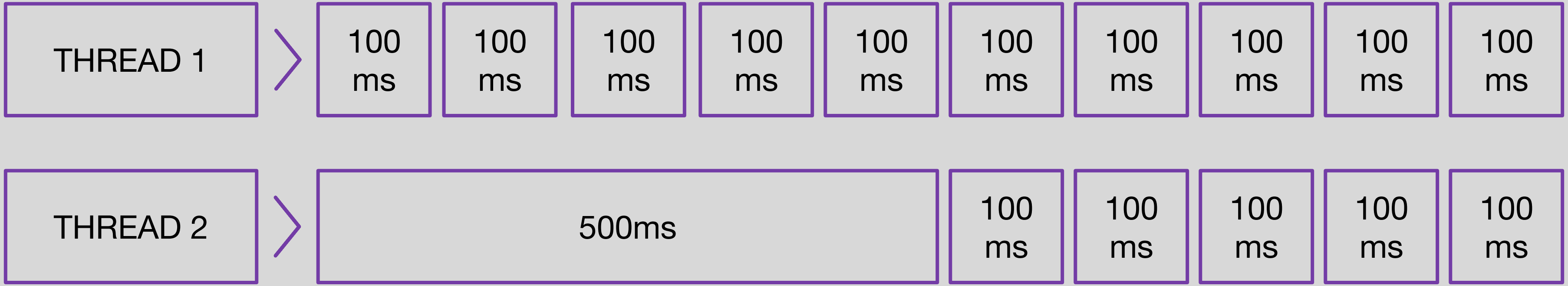
We’re talking about I/O:
disk and network

Averaged flow

THREAD 1	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms
THREAD 2	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms	100 ms

This is how 20 RPS looks like

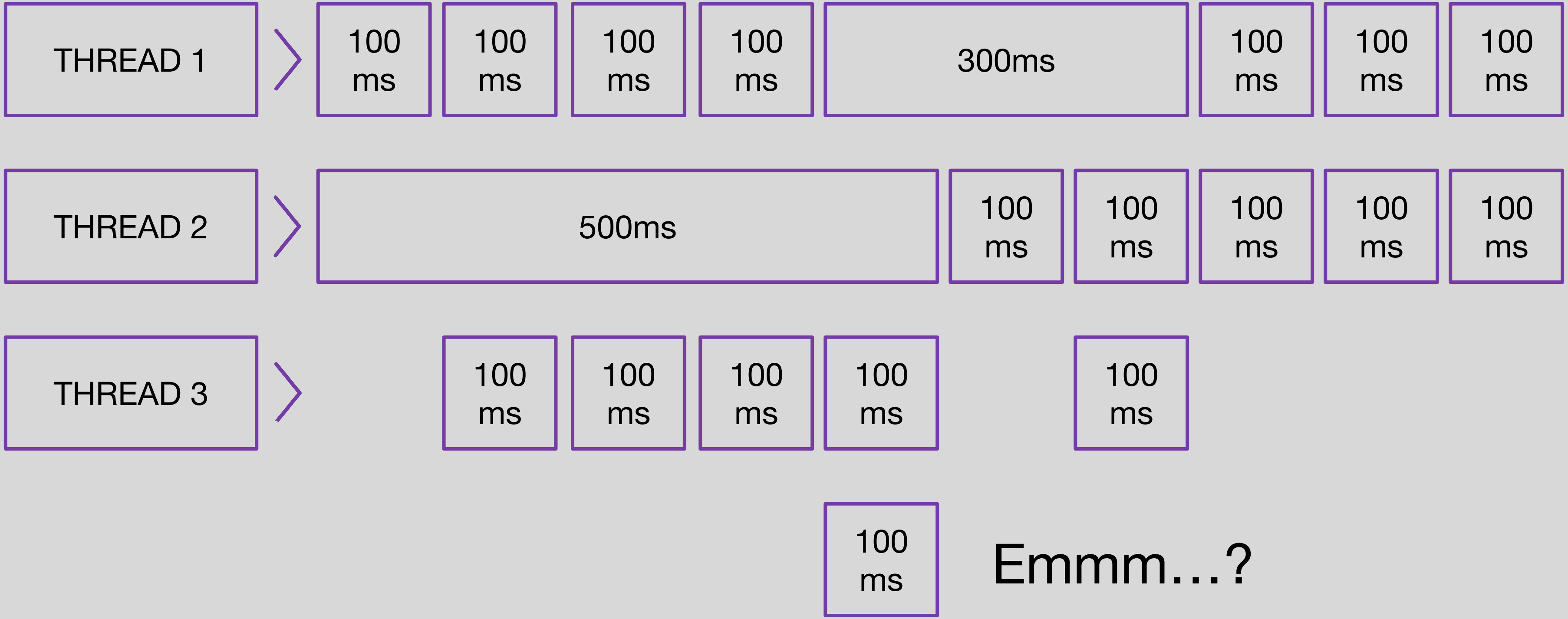
Something happened flow



This is how 16 RPS looks like
Wait, where are 4 more RPS?

Add another thread

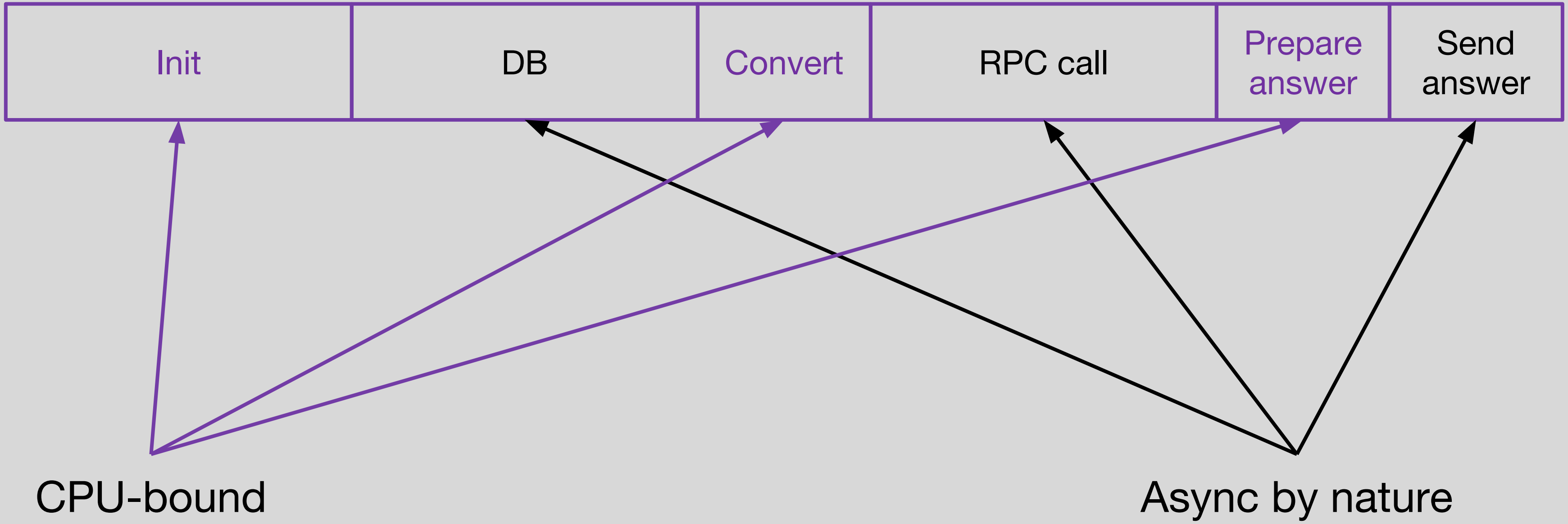
But… the real life is far from picture



03

Going deeper

Inside request



Context Switch is somewhat expensive operation

Context Switch

- Default time slice is 100 msec
- Context Switch cost 3-30 usec
- 3+ orders of magnitude difference...
- … but how much time do you spent for computations in your code?
- 3 ns access to L1 cache, 70 ns access to memory [16].
- Not to mention wait time till reschedule.

Thread is an expensive thing by itself

Thread

- Thread allocates **X** KB for its stack (via -Xss option).
- By default on amd64 it's 1024 KB [4].
- Minimal value is 228 KB [5].
- 200 Jetty threads will consume 200MB non-heap memory [6] :)

Scaling in blocking system

- Adding more threads.
- More strict timeouts.
- Throttlers, circuit breakers etc.

But what if we still want to serve some of that rejected requests?

04

Let’s go async!

Async request processing

Init	DB	Convert	RPC call	Prepare answer	Send answer

- Init and start async request to DB;
- Process another request;
- Once DB responded, convert data and issue an RPC call;
- Process another request;
- Once RPC responded, prepare answer and start sending.

Cons of async programming

- Request can be executed in different threads.
- Therefore, monitoring is more complicated.
- Harder to debug.
- Harder to control concurrency, resource usage. Therefore, harder to implement backpressure.
- Code is more complex.

Monitoring

- NewRelic supports limited amount of async libraries/frameworks *
- You can’t just wrap piece of code into try { meter } finally { report} block, you have to pass start time along with request.
- Monitoring of queues for shared resources isn’t optional anymore.

Debug

- Thread dump is almost useless. You will see polling threads without any context.
- Need to search by transaction id.
- Logs are essential (or any other trace tool).

Concurrency Control

- No more single Jetty thread pool to control concurrency on application.
- Need to configure all queues and make it granular (i.e. queue for each rpc client).
- Have to think about resource limits (for each resource and global limit).

Scaling in non-blocking system

- Fine-tuning of queue size and behavior (LIFO vs FIFO).
- More strict timeouts.
- Throttlers, circuit breakers etc.

All async-by-nature operations already implemented async in kernel

05

Where is the gain?

No magic.
Sorry.

According to Netflix[8]

- You will get resilience for your system.
- You will get ability to handle more concurrent connections.
- You won’t get performance boost (or maybe yes).
- You won’t get latency improvement (or maybe yes).
- You will get more complicated code.

Async is like TDD

:)

06

Under the Hood

Based on:

- epoll (linux) [9] or
- kqueue (bsd, mac os) [10] or
- IOCP (windows) [11]

epoll[12]

- Single file descriptor aggregates many file descriptors.
- Return fds that are available to operate.
- Works for constant time (as opposite to O(N)).
- 1 syscall vs N syscalls.

07

Why to use sync?

You already use it.
Proven to work.

But really, why?

- Linear execution flow: easier code.
- Everyone got used to it.
- Sometimes you don’t care.

08

When to use sync?

Use blocking if

- You don’t care about handling a lot of requests.
- Your application is mostly CPU-bound.
- Your application read/write a lot from few files/sockets.
- You aren’t afraid of OOM storms.

09

Async Programming

Use node.js, it’s async
from the first day.

Event Loop[13]

This is how event loop can look like:

- You can't spawn threads
- You may only enqueue tasks to the queue
- Different tasks will be executed somehow in background and will issue an event once it's done.

```
function main() {
  while (true) {
    val task = taskQueue.take();
    if (task.type === 'Function') {
      task.execute();
    } else if (task.type === 'BytesReadFromFile') {
      task.callback.call(task.bytes);
    } else if (task.type === 'BytesReadFromSocket') {
      task.callback.call(task.bytes);
    }
  }
}
```

Future/Promise

- Typically it’s an abstraction over thread pool (but not necessarily).
- Callbacks!
- But you may use for comprehensions in Scala [14].
- Very important to restrain temptation of using Await.

Future/Promise

Future itself doesn’t mean using async IO, but it allows you to think so.

In this example:

- We use DeferredResult which allow to return worker thread to jetty back (a kind of continuation).
- Decoupling of worker threads of jetty and application logic threads.

```scala
@RequestMapping("/action")
def action(): DeferredResult[Response] = {
  val result = new DeferredResult[Response]()
  val future: Future[Response] = executeActionAsync()
  future.onComplete {
    case Success(a) => result.setResult(a)
    case Failure(e) => result.setErrorResult(e)
  }
  result
}
```

Actor Model[15]

- Actor is an execution unit.
- Actor has an input queue (mailbox).
- Actor can send [immutable] messages to other actors.
- One actor can be executed only in one thread at any point of time (each actor is single-threaded).
- Therefore, actor can store mutable state, access to this state doesn’t require synchronization.

Actor Example

This is simple example of actors:

- SumActor have a cache, it's safe to access it.
- Actor framework take care of execution in a single thread for each actor.

```
class Actor[T] {
  private val queue: NonBlockingQueue[T] = ...
  protected def execute(message: T): Unit
  final def send(message: T) = queue.put(message)
  final def run = // magic to get messages from queue and pass it to execute
}

case class SumMessage(a: Int, b: Int, nextActor: Actor[Int])

class SumActor extends Actor[Sum] {
  val cache: Cache[(Int, Int), Int] = ...
  override def process(m: Sum): Unit = {
    val sum = cache.getOrElseUpdate(m.a -> m.b, {
      m.a + m.b
    }
    m.nextActor.send(sum)
  }
}
```

WiX Engineering

Thank You!

https://goo.gl/t4Ack2

dmitryk@wix.com | @dkomanov | linkedin/dkomanov | github.com/dkomanov

References

1. http://stackoverflow.com/questions/16401294/how-to-know-linux-scheduler-time-slice
2. http://www.linuxplumbersconf.org/2013/ocw//system/presentations/1653/original/LPC%20-%20User%20Threading.pdf
3. http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
4. http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html
5. http://xmlandmore.blogspot.com/2014/09/jdk-8-thread-stack-size-tuning.html
6. https://github.com/eclipse/jetty.project/blob/jetty-9.4.x/jetty-util/src/main/java/org/eclipse/jetty/util/thread/QueuedThreadPool.java
7. https://docs.newrelic.com/docs/agents/java-agent/getting-started/compatibility-requirements-java-agent
8. http://techblog.netflix.com/2016/09/zuul-2-netflix-journey-to-asynchronous.html
9. https://en.wikipedia.org/wiki/Epoll
10. https://en.wikipedia.org/wiki/Kqueue
11. https://en.wikipedia.org/wiki/Asynchronous_I/O
12. https://www.quora.com/Network-Programming-How-is-epoll-implemented
13. https://www.youtube.com/watch?v=8aGhZQkoFbQ
14. http://danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html
15. https://en.wikipedia.org/wiki/Actor_model
16. https://news.ycombinator.com/item?id=13934288