... continued

This lesson continues the discussion on asynchronous programming.

WebServers

A common application of the asynchronous paradigm is in web servers, which spend most of their time waiting for network I/O. A more recent approach to implementing web servers uses event loops. An event loop is a programming construct that waits for events to happen and then dispatches them to an event handler. Languages such as JavaScript, Ruby (MRI implementation) and Python (standard C implementation) enable the asynchronous programming model using an event loop. The idea is that a single thread runs in a loop waiting for an event to occur. Once an event arrives, it is appropriately dispatched to an event handler. The event loop's thread immediately goes back to listening for another event. The event handler may run on another thread if the language supports multiple threads. This design achieves very high concurrency if an application is frequently involved in either of the following:

- Network I/O
- Disk I/O

If an application spends most of its time waiting for I/O, it can benefit from an asynchronous design. One of the most common use cases you'll find in the wild is of web servers implemented using asynchronous design. A web server waits for an HTTP request to arrive and returns the matching resource. Folks familiar with JavaScript would recall NodeJS works on the same principle. It is a web server that runs an event loop to receive web requests in a single thread. Contrast that to web servers, which create a new thread, or worse, fork a new process, to handle each web request. In some benchmarks, the asynchronous event loop based web request. In some benchmarks, the asynchronous event loop based

web servers outperformed multithreaded ones, which may seem counterintuitive.

To truly appreciate asynchronous programming model, we'll present Ryan Dahl's motivation for creating NodeJS which also runs an event loop. Ryan classifies disk and network I/O operations as blocking operations and presents the following table to put the latency for various operations in perspective.

Device	**CPU Cycles**	**Humanified**
L1 cache	3	1 seconds
L2 cache	14	4.6 seconds
RAM	250	83 seconds
Disk	41000000	158 days
Network	240000000	2.5 years

The third column *humanizes* the CPU cycles by assuming if 3 CPU cycles equaled one second then a network I/O would feel equivalent of 2.5 years. This may dawn upon you how slow can disk and network I/O be in comparison to access times for other devices. The long waiting times suggest that if these blocking calls are synchronous then we end up wasting a lot of CPU cycles. The CPU can be better utilized especially in environments with a lot of I/O.

As a thought exercise, if we implement a chat server using the asynchronous model, a single thread can serve several hundred clients since the thread spends most of its time waiting for clients to send messages. In such a scenario, if we created one thread per client, we would quickly use up system resources since threads aren't free. Creating, maintaining and tearing down threads takes CPU cycles in addition to maintaining and tearing down threads takes CPU cycles in addition to memory. In fact, this difference becomes more visible in web servers that use threads to handle HTTP web requests vs. which use an event loop. Apache is an example of the former and NGINX of the latter. NGINX outshines Apache in memory usage under high load.

If your application is CPU bound then shifting to an asynchronous model may not display significant performance gains. In fact, you'll see none for a hundred percent compute bound program. In such a case a hardware upgrade is more likely to improve performance.

Asynchronous Programming in Ruby

Various Ruby frameworks/gems exist that enable asynchronous programming. We'll not be discussing anyone in particular but list them below for reference:

- async gem
- EM Synchrony
- Event Machine
- Concurrent Rub's Async Mixin