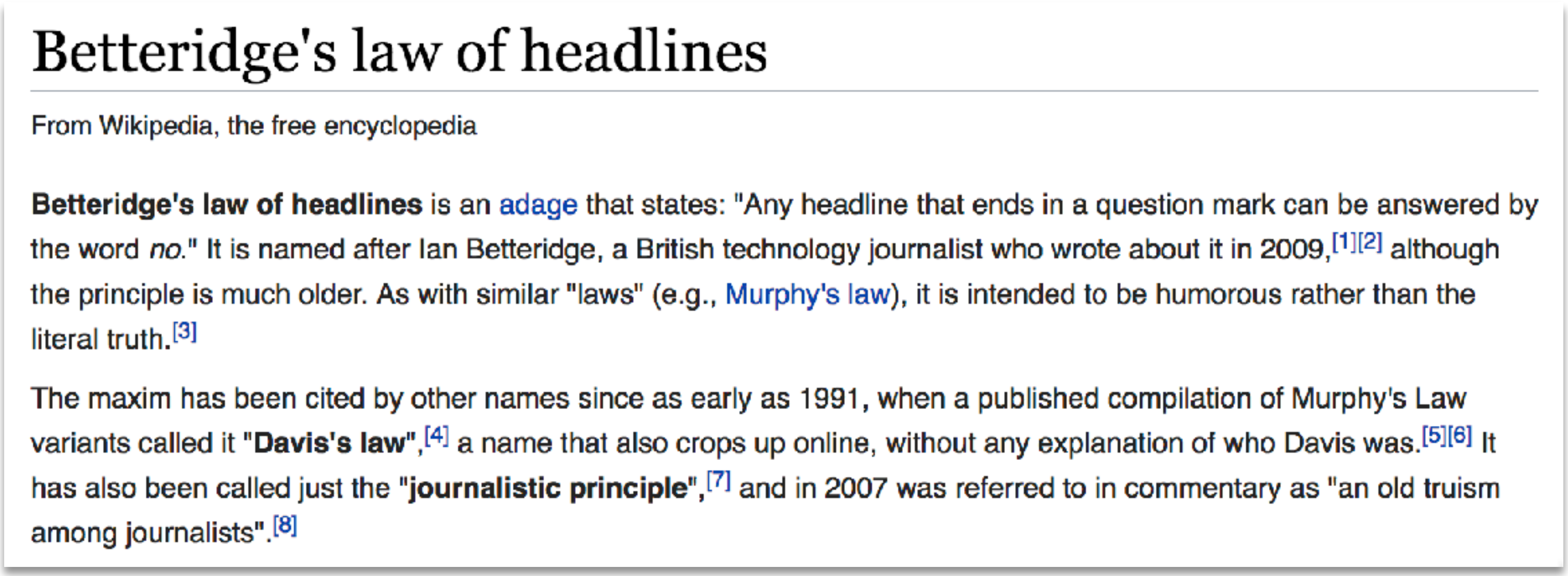
Is it time to rewrite the operating system in Rust?

Bryan Cantrill
CTO

bryan@joyent.com
@bcantrill

Spoiler alert



Betteridge's law of headlines

From Wikipedia, the free encyclopedia

Betteridge's law of headlines is an adage that states: "Any headline that ends in a question mark can be answered by the word *no*." It is named after Ian Betteridge, a British technology journalist who wrote about it in 2009,[1][2] although the principle is much older. As with similar "laws" (e.g., Murphy's law), it is intended to be humorous rather than the literal truth.[3]

The maxim has been cited by other names since as early as 1991, when a published compilation of Murphy's Law variants called it "**Davis's law**",[4] a name that also crops up online, without any explanation of who Davis was.[5][6] It has also been called just the "**journalistic principle**",[7] and in 2007 was referred to in commentary as "an old truism among journalists".[8]

What even is the operating system?

- The operating system is harder to define than it might seem…
- For every definition, it can be easy to come up with exceptions
- At minimum: the operating system is the program that abstracts hardware to allow execution of other programs
- The operating system defines the liveness of the machine: without it, no program can run
- The operating system software that runs with the highest level of architectural privilege is the operating system kernel
- …but the kernel is not the entire operating system!

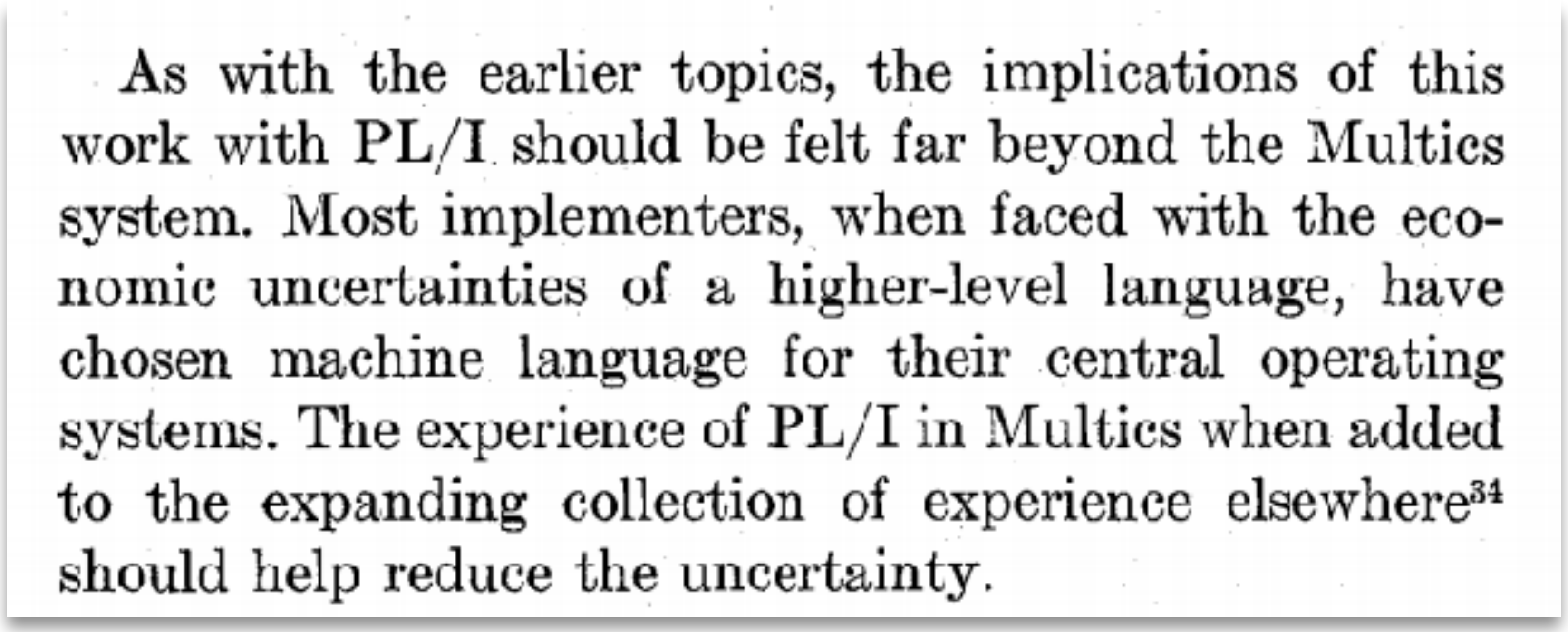
Operating system implementation history

- Historically, operating systems — née “executives” — were written entirely in assembly
- Starting with the Burroughs B5000 MCP in 1961, operating systems started to be written in higher level languages…
- In 1964, when Project MAC at MIT sought to build a successor to their Compatible Timesharing System (CTSS), they selected the language (PL/I) before writing any code (!)
- But PL/I had no functioning compiler — and wouldn’t until 1966

PL/I in Multics

Joyent

- The decision to use PL/I in Multics was seen by its creators as a great strength, even when reflecting back in 1971:



As with the earlier topics, the implications of this work with PL/I should be felt far beyond the Multics system. Most implementers, when faced with the economic uncertainties of a higher-level language, have chosen machine language for their central operating systems. The experience of PL/I in Multics when added to the expanding collection of experience elsewhere[34] should help reduce the uncertainty.

Source: “Multics: The first seven years,” Corbato et al.

- …but that the compiler was unavailable for so long (and when was available, performed poorly) was a nearly-fatal weakness

The birth of Unix

- Bell Labs pulled out of the Multics project in 1969
- A researcher formerly on the Multics effort, Ken Thompson, implemented a new operating system for the PDP-7
- The system was later ported to the PDP-11/20, where it was named Unix — a play on “eunuchs” and a contrast to the top-down complexity of Multics
- Unix was implemented entirely in assembly!

Unix and high-level languages

- The interpreted language B (a BCPL derivative), was present in Unix, but only used for auxiliary functionality, e.g. the assembler and an early version of dc(1)
- Some of the B that was in use in Unix was replaced with assembly for reasons of performance!
- Dennis Ritchie and Thompson developed a B-inspired language focused on better abstracting the machine, naming it “C”
- Perhaps contrary to myth, C and Unix were *not* born at the same instant — they are siblings, not twins!

The C revolution

Joyent

- C is rightfully called “portable assembly”: it is designed to closely match the abstraction of the machine itself
- C features memory addressability at its core
- Unlike PL/I, C grew as concrete needs arose
- e.g., C organically adopted important facilities like macro processing through the C preprocessor
- Standardization efforts came late and were contentious: C remains infamous for its undefined behaviors

Operating systems in the 1980s

- As the minimal abstraction above the machine, C — despite its blemishes — proved to be an excellent fit for operating systems implementation
- With few exceptions, operating systems — Unix or otherwise — were implemented in C throughout the 1980s
- Other systems existed as research systems, but struggled to offer comparable performance to C-based systems

Operating systems in the 1990s

- In the 1990s, object oriented programming came into vogue, with languages like C++ and Java
- By the mid-1990s, C-based systems were thought to be relics
- …but the systems putatively replacing them were rewrites — and suffered from rampant Second System Syndrome
- They were infamously late (e.g. Apple’s Copland), infamously slow (e.g. Sun’s Spring), or both (Taligent’s Pink)
- Java-based operating systems like Sun’s JavaOS fared no better; hard to interact with hardware without unsigned types!

Operating systems in the 2000s

- With the arrival of Linux, Unix enjoyed a resurgence — and C-based operating systems became deeply entrenched
- With only a few exceptions (e.g., Haiku), serious attempts at C++-based kernels withered
- At the same time, non-Java/non-C++ languages blossomed: first Ruby, and then Python and JavaScript
- These languages were focused on ease of development rather than performance — and there appears to be no serious effort to implement an operating system in any of these

Systems software in the 2010s

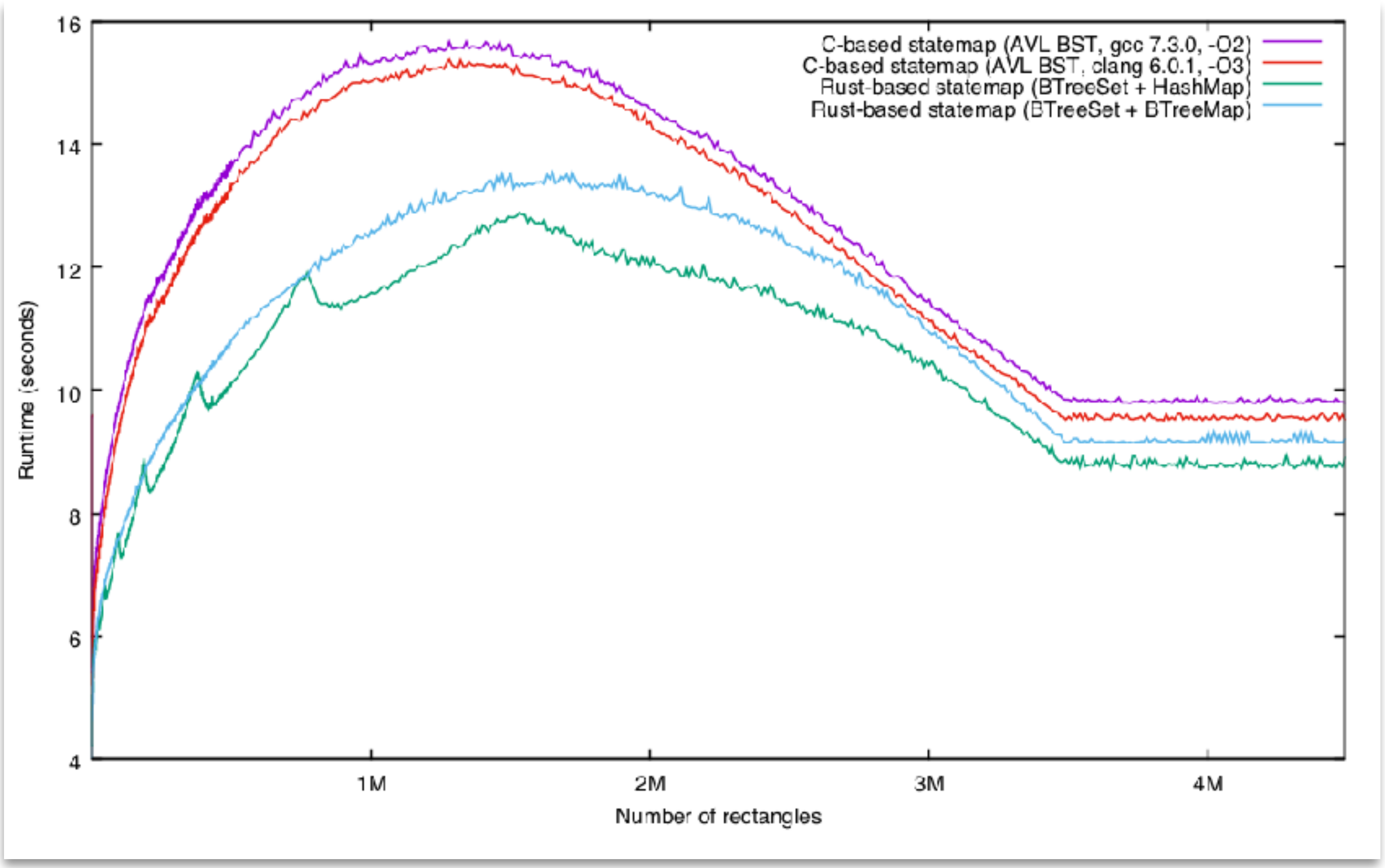
- Systems programmers began pining for something different: the performance of C, but with more powerful constructs as enjoyed in other languages
- High-performance JavaScript runtimes allowed for a surprising use in node.js — but otherwise left much to be desired
- Bell Labs refugees at Google developed Go, which solves some problems, but with many idiosyncrasies
- Go, JavaScript and others are **garbage collected**, making interacting with C either impossible or excruciatingly slow

Rust?

- Rust is a systems software programming language designed around safety, parallelism, and speed
- Rust has a novel system of ownership, whereby it can *statically* determine when a memory object is no longer in use
- This allows for the *power* of a garbage-collected language, but with the *performance* of manual memory management
- This is important because — *unlike C* — Rust is highly composable, allowing for more sophisticated (and higher performing!) primitives

Rust performance (my experience)

Joyent



Source: http://dtrace.org/blogs/bmc/2018/09/28/the-relative-performance-of-c-and-rust/

Rust: Beyond ownership

Joyent

- Rust has a number of other features that make it highly compelling for systems software implementation:
  - Algebraic types allow robust, concise error handling
  - Hygienic macros allow for safe syntax extensions
  - Foreign function interface allows for full-duplex integration with C without sacrificing performance
  - "unsafe" keyword allows for some safety guarantees to be surgically overruled (though with obvious peril)
- Also: terrific community, thriving ecosystem, etc.

Operating systems in Rust?

Joyent

- If the history of operating systems implementation teaches us anything, it's that runtime characteristics **trump** development challenges!
- Structured languages (broadly) replaced assembly because they **performed** as well
- Viz., every operating system retains some assembly for reasons of performance!
- With its focus on performance and zero-cost abstractions, Rust **does** represent a real, new candidate programming language for operating systems implementation

Operating systems in Rust: A first attempt

Joyent

- First attempt at an operating system kernel in Rust seems to be Alex Light's Reenix, ca. 2015: a re-implementation of a teaching operating system in Rust as an undergrad thesis
- Biggest challenge in Reenix was that Rust forbids an application from handling allocation failure
- The addition of a global allocator API has improved this in that now a C-based system can at least handle pressure…
- …but dealing with memory allocation failure is still very much an unsettled area for Rust (see Rust RFC 2116)

Operating systems in Rust since 2015

- Since Reenix's first efforts, there have been quite a few small systems in Rust, e.g.: Redox, Tifflin, Tock, intermezzOS, RustOS/QuiltOS, Rux, and Philipp Oppermann's Blog OS
- Some of these are teaching systems (intermezzOS, Blog OS), some are unikernels (QuiltOS) and/or targeted at IoT (Tock)
- These systems are all *de novo*, which represents its own challenges, e.g. forsaking binary compatibility with Linux and fighting Second System Syndrome

Operating systems in Rust: The challenges

- While Rust's advantages are themselves clear, it's less clear what the advantage is when replacing otherwise working code
- For in-kernel code in particular, the safety argument for Rust carries less weight: in-kernel C tends to be *de facto* safe
- Rust does, however, presents new challenges for kernel development, esp. with respect to **multiply-owned** structures
- An OS kernel — despite its historic appeal and superficial fit for Rust — may represent more challenge than its worth
- But what of hybrid approaches?

Hybrid approach I: Rust in-kernel components

Joyent

- One appeal of Rust is its ability to interoperate with C
- One hybrid approach to explore would be to retain a C-/assembly-based kernel while allowing for Rust-based in-kernel components like device drivers and filesystems
- This would allow for an incremental approach — and instead of rewriting, Rust can be used for *new* development
- There is a prototype example of this in FreeBSD; others are presumably possible

Hybrid approach II: Rust OS components

Joyent

- An operating system is *not* just a kernel!
- Operating systems have significant functionality at user-level: utilities, daemons, service-/device-/fault- management facilities, debuggers, etc.
- If anything, the definition of the OS is expanding to distributed system that represents a multi-computer control plane — that itself includes many components
- These components are much more prone to run-time failure!
- Many of these are an *excellent* candidate for Rust!

Hybrid approach III: Rust-based firmware

Joyent

- Below the operating system lurks hardware-facing special-purpose software: firmware
- Firmware is a sewer of unobservable software with a long history of infamous quality problems
- Firmware has some of the same challenges as kernel development (e.g., dealing with allocation failures), but may otherwise be more amenable to Rust
- This is especially true when/where firmware is in user-space and is network-facing! (e.g., OpenBMC)

Looking forward: Systems software in Rust

Joyent

- Rust represents something that we haven’t seen in a long time: a modern language that represents an alternative throughout the stack of software abstraction
- Despite the interest in operating system kernel implementation, that might not be a good first fit for Rust
- Rust allows hybrid approaches, allowing for productive kernel incrementalism rather than whole-system rewrites
- Firmware and user-level operating system software are two very promising candidates for implementation in Rust!