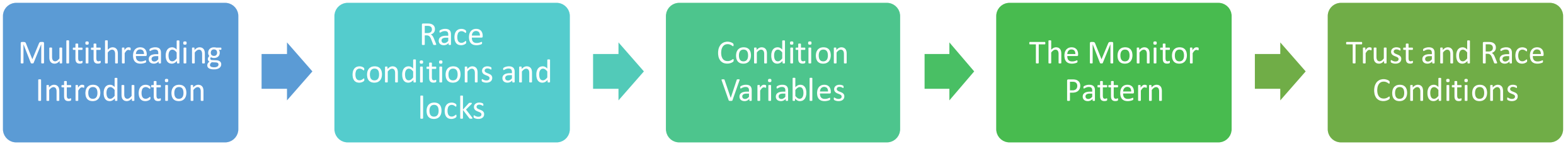
CS111, Lecture 16
Trust, Race Conditions and Operating Systems

Including slides based on a presentation by Dr. Benjamin Xie

This document is copyright (C) Stanford Computer Science and Nick Troccoli, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.
NOTICE RE UPLOADING TO WEBSITES: This content is protected and may not be shared, uploaded, or distributed. (without expressed written permission)

CS111 Topic 3: Multithreading, Part 1

Topic 3: **Multithreading** - How can we have concurrency within a single process? How does the operating system support this?



Lecture 12 **Lecture 13** **Lecture 14** **Lecture 15** **This Lecture**

assign4: implement several multithreaded programs while eliminating race conditions!

The Go Blog

Introducing the Go Race Detector

Dmitry Vyukov and Andrew Gerrand
26 June 2013

Introduction

Race conditions are among the most insidious and elusive programming errors. They typically cause erratic and mysterious failures, often long after the code has been deployed to production. While Go's concurrency mechanisms make it easy to write clean concurrent code, they don't prevent race conditions. Care, diligence, and testing are required. And tools can help.

We're happy to announce that Go 1.1 includes a race detector, a new tool for finding race conditions in Go code. It is currently available for Linux, OS X, and Windows systems with 64-bit x86 processors.

The race detector is based on the C/C++ ThreadSanitizer runtime library, which has been used to detect many errors in Google's internal code base and in Chromium. The technology was integrated with Go in September 2012; since then it has detected 42 races in the standard library. It is now part of our continuous build process, where it continues to catch race conditions as they arise.

Uber

A data race occurs in Go when two or more goroutines access the same memory location, at least one of them is a write, and there is no ordering between them, as defined by the Go memory model. Outages caused by data races in Go programs are a recurring and painful problem in our microservices. These issues have brought down our critical, customer-facing services for hours in total, causing inconvenience to our customers and impacting our revenue. In this blog, we discuss deploying Go's default dynamic race detector to continuously detect data races in our Go development environment. This deployment has enabled detection of more than 2,000 races resulting in ~1,000 data races fixed by more than two hundred engineers.

Eliminating Data Races in Firefox – A Technical Report



By Christian Holler, Aria Beingessner, Kris Wright

Posted on April 6, 2021 in Developer Tools, Featured Article, Firefox, and Security

We successfully deployed *ThreadSanitizer* in the Firefox project to eliminate data races in our remaining C/C++ components. In the process, we found several impactful bugs and can safely say that data races are often underestimated in terms of their impact on program correctness. We recommend that all multithreaded C/C++ projects adopt the ThreadSanitizer tool to enhance code quality.

What is ThreadSanitizer?

ThreadSanitizer (TSan) is compile-time instrumentation to detect *data races* according to the C/C++ memory model on Linux. It is important to note that these data races are considered *undefined behavior* within the C/C++ specification. As such, the compiler is free to assume that data races do not happen and perform optimizations under that assumption. Detecting bugs resulting from such optimizations can be hard, and data races often have an

Firefox

Stylo: a parallel CSS engine

Firefox Quantum includes Stylo, a pure-Rust CSS engine that makes full use of Rust's "Fearless Concurrency" to speed up page styling. It's the first major component of Servo to be integrated with Firefox, and is a major milestone for Servo, Firefox, and Rust. It replaces approximately 160,000 lines of C++ with 85,000 lines of Rust.

When a browser is loading a web page, it looks at the CSS and parses the rules. It then determines which rules apply to which elements and their precedence, and "cascades" these down the DOM tree, computing the final style for each element. Styling is a top-down process: you need to know the style of a parent to calculate the styles of its children, but the styles of its children can be calculated independently thereafter.

This top-down structure is ripe for parallelism; however, since styling is a complex process, it's hard to get right. Mozilla made two previous attempts to parallelize its style system in C++, and both of them failed. But Rust's fearless concurrency has made parallelism practical! We use rayon —one of the hundreds of crates Servo uses from Rust's ecosystem — to drive a work-stealing cascade algorithm. You can read more about that in Lin Clark's post. Parallelism leads to a lot of performance improvements, including a 30% page load speedup for Amazon's homepage.

Learning Goals

- Reflect on how we trust the software we use and how we choose what to trust
- Explore operating systems as a case study in trusting (and not trusting) large, far-reaching software systems.

Key Questions

1. What do we trust?
2. How do we trust those things?
3. How do we choose what to trust?
4. What are the impacts if that trust is violated?

Plan For Today

- Trust and Operating Systems
- What is trust?
- How do we decide what to trust?
- Example: Linux

Plan For Today

- **Trust and Operating Systems**
- What is trust?
- How do we decide what to trust?
- Example: Linux

Trust and Operating Systems

- Writing synchronization code is *hard* – difficult to reason about, bugs are tricky if they are hard to reproduce
- Lots of software incorporates concurrent code – OSes are one example
- Concurrency + unique properties of OSes (**immense scale**) provides a unique lens through which to examine how we **trust** software.

Trust and Operating Systems

Operating systems provide the root of trust for software.

- All software runs on top of OSes, so applications are only as trustworthy as the OS they run on
- Operating systems implement mechanisms used by applications

Operating systems also have unique characteristics that lead to their immense scale and impact.

OS Scale: Codebase

Scale of codebase: millions of lines of code, thousands of files

Example: as of 2020, the Linux kernel had > 27.8M lines, > 66K files

- Extremely large, hard to maintain (likely order of 10 years to make new one), systems outlive builders
- Bugs! E.g., even old OS/360 rule of thumb was ~1000 bugs was good enough to ship
  - New bugs surface about as quickly as old ones are fixed

Thought question: how does this impact whether / how much we trust OSes?

OS Scale: Users

Scale of users: millions – billions of users

Example: as of 2023, Windows had > 1B users

- Nowadays, not the only software with massive scale, but certainly among them
- Issues/bugs/vulnerabilities can have massive impact

Thought question: how does this impact how we think about OS bugs?

Examples of OS Bugs

- iOS Bug in 2011 caused some set alarms to not go off
- 2021 Windows **PrintNightmare** vulnerability in printing software allowed remote code execution
- 2017 macOS bug allowed admin access without a password
- 2022 Linux bug gives root access due to 12-year-old vulnerability
- 2015 Android Stagefright vulnerability in multimedia software
- EternalBlue Windows vulnerability (kept secret by NSA, hackers later leaked) led to WannaCry ransomware, affecting many

OS Scale: Impact

Scale of impact: many users, many applications built on top of OSes

A bug in an application can be very bad; a bug in an OS can potentially be tremendously bad

- Imagine if a bug allowed for root access to entire system – running code in privileged mode! E.g. **readSector**.
- Potential impact to entire system – large potential surface area
- OSes provide the root of trust for software
  - All software runs on top of Oses, so applications are only as trustworthy as OS beneath

Thought question: how does this impact how we think about OS bugs?

OS Scale: Longevity

Scale of longevity: OSes used for very long time

Users still on old OS versions, vulnerabilities still found in old OS versions

- Not all users update to new OSes, or update quickly
- *Example: 2023 update for almost-10-year-old iPhone 5S*
- Challenge of continuing to keep OSes secure
- Latest Google Pixel phones get 7 years of OS and security updates, Samsung Galaxy S24 series also gets 7 years of security updates and 7 Android upgrades
- **Assign2 question:** how do we think about mandated long-term support?

Thought question: how should OS-makers approach this to earn user trust?

What is an OS that you use? Why do you choose to trust it?

Respond on PollEv:
pollev.com/cs111



L16. What is an OS that you use? Why do you choose to trust it?

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Plan For Today

- Trust and Operating Systems
- **What is trust?**
- How do we decide what to trust?
- Example: Linux

Trust

Trust is to stop questioning the dependability of something.

- Efficiency/safety tradeoff: trust lowers the barriers of monitoring and questioning (more efficient)
- Involves *intentions, dependence, vulnerability/risk*
- **Agency:** our capacity to take actions that align with our goals
  - "when we trust, we try to make something a part of our agency... To unquestioningly trust something is to let it in—to attempt to bring it inside one's practical functioning." - CT Nguyen
  - **Trusting software is extending agency**
- ***Agential gullibility:*** *trusting more than is warranted*

Trust is powerful, necessary, risky

If I trust people or things (e.g. software), I …

- Integrate it with my own functioning
- Work more efficiently with them (stop questioning)
- Feel betrayed when they fail us

=> Trust (by extending agency) with great care!

Self-assessment on Trust

Small-group discussion:

1. *What do you trust and why?*
2. *How does it extend your agency?*
3. *Examples of agential gullibility?*

Plan For Today

- Trust and Operating Systems
- What is trust?
- **How do we decide what to trust?**
- Example: Linux

3 Paths to Trust

1. **Trust by Assumption:** trust absent any clues to warrant it
    1. Example: using unknown 3rd party library because deadline is approaching
    2. Example: warnings from others about imminent danger (e.g. “look out for the car!”)
2. **Trust by Inference:** trust based on information, e.g. past performance, characteristics, institutions
    1. Example: trust in brands or affiliation (weaker)
    2. Example: past performance (stronger)
    3. Example: trust in prior versions of software
3. **Trust by Substitution:** trust by implementing system to partly replace the need to trust something (“Plan B”)
    1. **E.g.** set an alarm on a second device in case the alarm on your phone doesn’t work
    2. **E.g.** using unique, fake per-app emails for login, in case your personal info is leaked

Paul B. de Laat: *How can contributors to open-source communities be trusted? On the assumption, inference, and substitution of trust*

Earning Trust

How do OSes try to earn our trust? Perhaps:

- Bug bounty / open security programs
- Reputation / track record
- Open-source software (code publicly available)

Plan For Today

- Trust and Operating Systems
- What is trust?
- How do we decide what to trust?
- **Example: Linux**

Linux Kernel

- Around since 1991
- as of 2020, the Linux kernel had > 27.8M lines, > 66K files
- > 150,000 developers
- Widely used in devices (e.g. Android based on Linux), datacenters

Different groups with different angles of trust:

- Users
- App developers building apps for Linux users
- Linux developers working on Linux itself

Users Trusting Linux

- Why: People use Linux-based tools to extend their agency
  - Android smartphones
  - 13.6% of servers
  - Almost all supercomputers
- How trust emerges?
  - Assumption
    - “never thought about it”
    - ”no other option”
  - Inference
    - open source, many eyes to detect and fix problems?
    - previous use
  - Substitution
    - Redundant security protocols (e.g. strong password, isolate/encrypt sensitive files, antivirus)

App Developers Trusting Linux

- Why: Standardization and tools of OS enable efficiency
  - High cost to build and maintain new OS
  - Familiar => lowers learning time developers
- How trust emerges?
  - Assumption: rare
  - Inference
    - Used by other app developers (lots of stars on GitHub)
    - trust Linus Torvalds
  - Substitution
    - code is open source (read it, fork it)
    - Add “redundant” checks in code (ex: spurious wakeups)



Linux Programmers Trusting Each Other

- Why: No single person can build & maintain an OS. Need to extend agency to others to support.
- How trust emerges?
  - Assumption: rarely happens
  - Inference
    - Known in community
    - Quality of previous code submissions
  - Substitution
    - Changes must be reviewed, accepted in layered process
    - Decision making: Linus has final authority

"I don't like the idea of having developers do their own updates in my kernel source tree. (...)
"there really aren't that many people that I trust enough to give write permissions to the kernel tree."
– Linus Torvalds

Agential Gullibility Example

Example of *agential gullibility* within Linux programmer community: a “Trojan Horse” discovered in the Linux **ssh** program! Would have enabled attackers to gain access to any Linux system. Caught by chance!

- Attack was in a package ”xz” used for encryption; ssh developers trusted xz.
- Someone posed as a legitimate open source developer for multiple years.
- They began to express impatience with the xz lead maintainer for not incorporating patches fast enough.
- Multiple other developers (probably also phonies) pressured the lead maintainer to accept help from this attacker. Eventually they were given permission to merge changes into xz.
- Attacker also submitted changes for OSS-Fuzz, which scans open-source packages for malicious code; the change disabled a check that would have exposed the Trojan Horse.

Trusting Software vs. Non-Software

Thought question: does our approach to trusting software differ from our approach to trusting other things/services/products? If not, should it? If so, how does it differ?

https://www.theatlantic.com/technology/archive/2015/11/programmers-should-not-call-themselves-engineers/414271/

Key Takeaways

Trust is often required, powerful, and dangerous. Key design challenge is how we design structures that enable us to substitute trust.

1. Trust amongst tech users, app developers, and system programmers is intertwined
2. Trust is about extending agency, enabling “unquestioning attitude”
3. Trust emerges through assumption, inference, substitution
4. Can design ways to (partially) substitute need to trust

Recap

- Trust and Operating Systems
- What is trust?
- How do we decide what to trust?
- Example: Linux

Lecture 16 takeaway: Trust is often required, powerful, and dangerous. A key design challenge is how we design structures that enable us to substitute trust.

Next time: how does the OS run and switch between threads?