CS111, Lecture 15

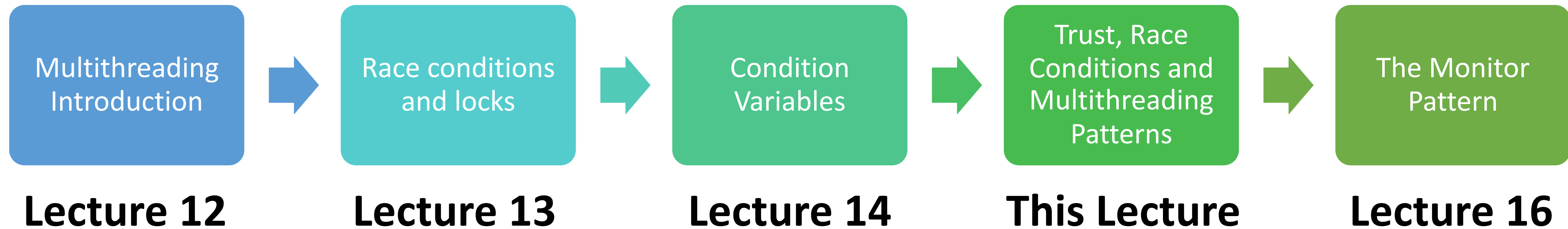
Race Conditions, Trust and Multithreading Patterns

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?



assign4: implement several multithreaded programs while eliminating race conditions!

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.
- Learn about the **monitor** pattern for designing multithreaded code in the simplest way possible, using classes.

Plan For Today

- **Recap:** mutexes, condition variables and dining philosophers
- Trust and Operating Systems
- How do we decide what to trust?
- Example: Linux
- Monitor pattern

Plan For Today

- **Recap: mutexes, condition variables and dining philosophers**
- Trust and Operating Systems
- How do we decide what to trust?
- Example: Linux
- Monitor pattern

Condition Variables

A **condition variable** is a variable type that can be shared across threads and used for one thread to notify other thread(s) when something happens. Conversely, a thread can also use this to wait until it is notified by another thread.

- You make one for each distinct event you need to wait / notify for.
- We can call **wait(lock)** on the condition variable to sleep until another thread signals this condition variable (no busy waiting). The condition variable will unlock (at the beginning) and re-lock (at the end) the specified lock for us.
- You call **notify_all** on the condition variable to send a notification to all waiting threads and wake them up.
- Analogy: radio station – broadcast and tune in

Condition Variables

1. Identify a single kind of event that we need to wait / notify for
2. Ensure there is proper state to check if the event has happened
3. Create a condition variable and share it among all threads either waiting for that event to happen or triggering that event
4. Identify who will notify that this happens, and have them notify via the condition variable
5. Identify who will wait for this to happen, and have them wait via the condition variable

waitForPermission (Final version)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    while (permits == 0) {
        permitsCV.wait(permitsLock);
    }
    permits--;
    permitsLock.unlock();
}
```

This is the final implementation with the final version of wait() that takes a mutex parameter and which is called in a while loop.

Passing a Lock To CV.wait()

Why do we need to pass our mutex as a parameter to wait()?

- We must release the lock when waiting so someone else can put a permit back (which requires having the lock)
- But if we release the lock before calling wait, someone else could swoop in and put a permit back before we call wait(), meaning we will miss the notification!

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsLock.unlock();
      // AIR GAP HERE – someone could acquire the lock before we wait!
      permitsCV.wait();     // (note: not final form of wait)
      permitsLock.lock();
    }
    permits--;
    permitsLock.unlock();
}
```

Passing a Lock To CV.wait()

Why do we need to call wait() in a while loop?

- If we are waiting and then woken up by a notification, it's possible by the time we exit wait(), there are no permits, so we must wait again.
- Note: wait() reacquires the lock before returning
- *spurious wakeups* – wakeups up even when not being notified by another thread (!)

```
static void waitForPermission(size_t& permits, condition_variable_any& permitsCV,
mutex& permitsLock) {
    permitsLock.lock();
    if (permits == 0) {
      permitsCV.wait(permitsLock);
      // by the time we wake up here, all the permits could already be gone!
    }
    permits--;
    permitsLock.unlock();
}
```

Plan For Today

- **Recap:** mutexes, condition variables and dining philosophers
- **Trust and Operating Systems**
- How do we decide what to trust?
- Example: Linux
- Monitor pattern

```
cp -r /afs/ir/class/cs111/lecture-code/lect15 .
```

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.

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

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
- **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
or text CS111 to 22333 once to join.

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

- Recap: mutexes, condition variables and dining philosophers
- Trust and Operating Systems
- **How do we decide what to trust?**
- Example: Linux
- Monitor pattern

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)

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

Think back to the person/thing/service you trusted…

How does trusting them extend your agency/functioning?

How might/did you exhibit *agential gullibility*? (trust more than is warranted)

What would be/was the result of your trust being violated?

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*

Plan For Today

- Recap: mutexes, condition variables and dining philosophers
- Trust and Operating Systems
- How do we decide what to trust?
- **Example: Linux**
- Monitor pattern

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
    - previous use
  - Substitution
    - Redundant security protocols (e.g. strong password, isolate/encrypt sensitive files)

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 given affordances to infer trust
  - 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)

Systems Programmers Trusting Linux

- 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
    - Formalization: tools and procedures to streamline cooperation
    - Division of roles
    - 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

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

Plan For Today

- **Recap:** mutexes, condition variables and dining philosophers
- Trust and Operating Systems
- How do we decide what to trust?
- Example: Linux
- **Monitor pattern**

Multithreading Patterns

- Writing synchronization code is *hard* – difficult to reason about, bugs are tricky if they are hard to reproduce
- E.g. how many locks should we use for a given program?
  - Just one? Doesn't allow for much concurrency
  - One lock per shared variable? Very hard to manage, gets complex, inefficient
- Like with dining philosophers, we must consider many scenarios and have lots of state to track and manage
- **One design idea to help:** the "monitor" design pattern - associate a single lock with a collection of related variables, e.g. a **class**
  - That lock is required to access any of those variables

Monitor Design Pattern

- For a multithreaded program, we can define a class that encapsulates the key multithreading logic and make an instance of it in our program.
- This class will have 1 mutex instance variable, and in all its methods we'll lock and unlock it as needed when accessing our shared state, so multiple threads can call the methods
- We can add any other state or condition variables we need as well – but the key idea is there is **one mutex** protecting access to all shared state, and which is locked/unlocked in the class methods that use the shared state.

Bridge Crossing

Next time: we'll use the monitor pattern to write a program that simulates cars crossing a one-lane bridge.

Recap

- **Recap:** mutexes, condition variables and dining philosophers
- Trust and Operating Systems
- How do we decide what to trust?
- Example: Linux
- Monitor pattern

Next time: more about the monitor pattern

Lecture 14 takeaway: Trust is often required, powerful, and dangerous. Key design challenge is how we design structures that enable us to substitute trust. The monitor pattern is a design pattern that can help reason about multithreaded code.