Parallel And Sequential Data Structures and Algorithms

Concurrency (The Finale)

Announcements

- **Quick 3 solution session** this Friday
- **Final exam review** after the quiz solution session
- Then **more final exam review** on Saturday!

Concurrency and Data Races

Structured Parallelism

- In this course we focused on:
  - Nested Fork-Join parallelism
  - (Mostly) pure functions
  - No shared mutable variables
  - Work and span in the "fork-join RAM"

This was a safe world free of data races, interference, undefined behavior, …

- The real world is **not** always so kind…

When Structure Disappears

- Consider the following "algorithm"

```
x = 0
parallel for i in 0...n-1:
  x = x + 1
```

- What is the final value of x?
- What is the span of this computation?

This is a **data race**, this is **undefined behavior**

The Race

```
x = 1
```

Thread 1

```
x = x + 1
```

```
read x: 0


compute x+1 = 0+1 = 1
store 1 to x
```

Thread 2

```
x = x + 1
```

```
read x: 0
compute x+1 = 0+1 = 1


store 1 to x
```

Data Races

Definition (Data Race): A **data race** occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

- We say that the presence of a data race makes the behavior of a program **undefined**

This is a **data race**, this is **undefined behavior**

Data Races

This is a **data race**, this is **undefined behavior**

- This is **not** just a theoretical restriction
- Data races are undefined behavior in major programming languages that support concurrency and parallelism

Data Races and Race Conditions

Safe Rust guarantees an absence of data races, which are defined as:

- two or more threads concurrently accessing a location of memory
- one or more of them is a write
- one or more of them is unsynchronized

A data race has Undefined Behavior, and is therefore impossible to perform in Safe Rust. Data races are prevented *mostly* through Rust's ownership system alone: it's impossible to alias a mutable reference, so it's impossible to perform a data race. Interior mutability makes this more complicated, which is largely why we have the Send and Sync traits (see the next section for more on this).

Data races

Different threads of execution are always allowed to access (read and modify) different memory locations concurrently, with no interference and no synchronization requirements.

Two expression evaluations *conflict* if one of them modifies a memory location or starts/ends the lifetime of an object in a memory location, and the other one reads or modifies the same memory location or starts/ends the lifetime of an object occupying storage that overlaps with the memory location.

A program that has two conflicting evaluations has a *data race* unless

- both evaluations execute on the same thread or in the same signal handler, or
- both conflicting evaluations are atomic operations (see `std::atomic`), or
- one of the conflicting evaluations *happens-before* another (see `std::memory_order`).

If a data race occurs, the behavior of the program is undefined.

Another Example

```
balances : dict<str, int> = {}

fun transfer(source : str, target : str, amount : int):
  src_balance = balances[source]
  target_balance = balances[target]

  balances[source] = src_balance - amount
  balances[target] = target_balance + amount
```

What if we invoke this function twice **in parallel**?

```
parallel (transfer('Cole', 'Edward', 5),
          transfer('Cole', 'Edward', 5))
```

Another Example

`balances['Cole']: 4995` `balances['Edward']: 4010`

Thread 1

```
src_balance = balances['Cole']                  5000

target_balance = balances['Edward']             4000
balances['Cole'] = 5000 - 5                     4995
balances['Edward'] = 4000 + 5                   4005
```

Thread 2

```
src_balance = balances['Cole']                  5000
```

Thread 2 didn't "see" this update.
This is a data race!

```
target_balance = balances['Edward']             4005
balances['Cole'] = 5000 - 5                     4995
balances['Edward'] = 4000 + 5                   4010
```

Data Races

Definition (Data Race): A **data race** occurs when there are two unsynchronized parallel operations on the same memory location, and at least one of the operations is a write.

- There's an important word in this definition…
- **unsynchronized**

Programming languages and **hardware** have tools for operating on shared mutable data without a data race

Synchronization

Synchronization Primitives

- You'll learn about a class of synchronization primitives, **locks/mutexes**, in 15-213

Definition (Lock): A **lock** is a tool that prevents multiple threads from executing particular sections of code concurrently

- Locks guarantee **mutual exclusion**; two pieces of code that require the lock will never run concurrently

Using Locks

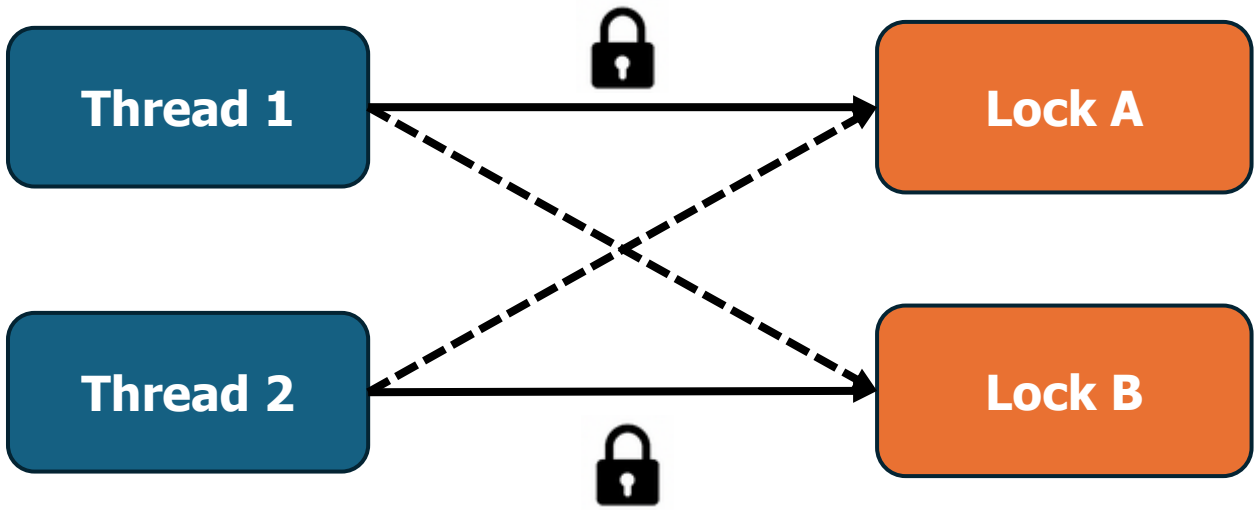
```
balances : dict<str, int> = {}
lock : Lock = /* create a lock */

fun transfer(source : str, target : str, amount : int):
  with lock:
    src_balance = balances[source]
    target_balance = balances[target]

    balances[source] = src_balance - amount
    balances[target] = target_balance + amount
```

The code protected by the lock is called the "**critical section**". At most one thread can be inside the critical section at any moment. Other threads will have to wait

- See 15-213 for information about locks

Deadlock

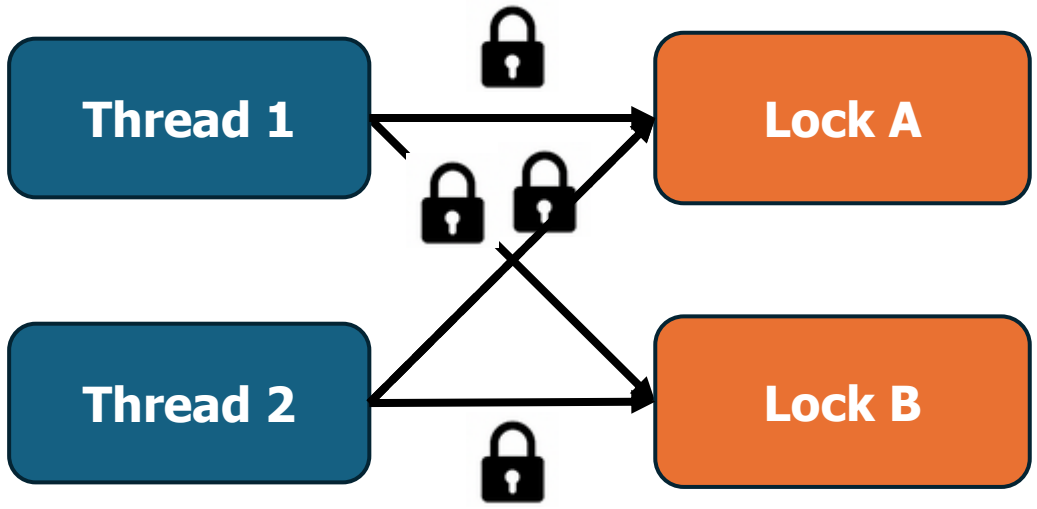
- Locks can be used to fix concurrency bugs, but they can also cause bugs! Especially, when using multiple locks!



Deadlock Avoidance

One way to prevent deadlock is to **impose a strict order** on how locks are acquired.

- **If every program requests locks in the same order, circular waiting can't happen**.
- *Example*: always request locks in alphabetical order.
  - Both threads try to acquire **Lock A**
  - One gets the lock, the other waits and never gets **Lock B** until it's safe.



Downsides of Locks

- Locks can cause deadlock, a serious program bug!
- Locks can also seriously affect performance

Locks prevent concurrency bugs… by preventing concurrency!

- This means that they inhibit parallelism!

Locks are (often) not suitable for highly parallel code

Lock-Free Synchronization

Atomic Operations

- Computer hardware provides certain operations that are safe (i.e, not a data race) to be executed concurrently

Definition (Compare-and-Swap / Compare Exchange): A compare-and-swap operation atomically reads a memory location and replaces the contents with a desired value **if** the current value matches a given value

```
fun cas(T* a, T expected, T desired) -> bool:
  if *a == expected:
    *a ← desired
    return True
  else:
    return False
```

A `cas` has these semantics **but** is executed as a single uninterruptible atomic operation by the hardware

Lock-Free Data Structures

- Most data structures if mutated in parallel would instantly break due to data races
- **Simplest fix**: Use a lock to prevent parallel use of the data structure

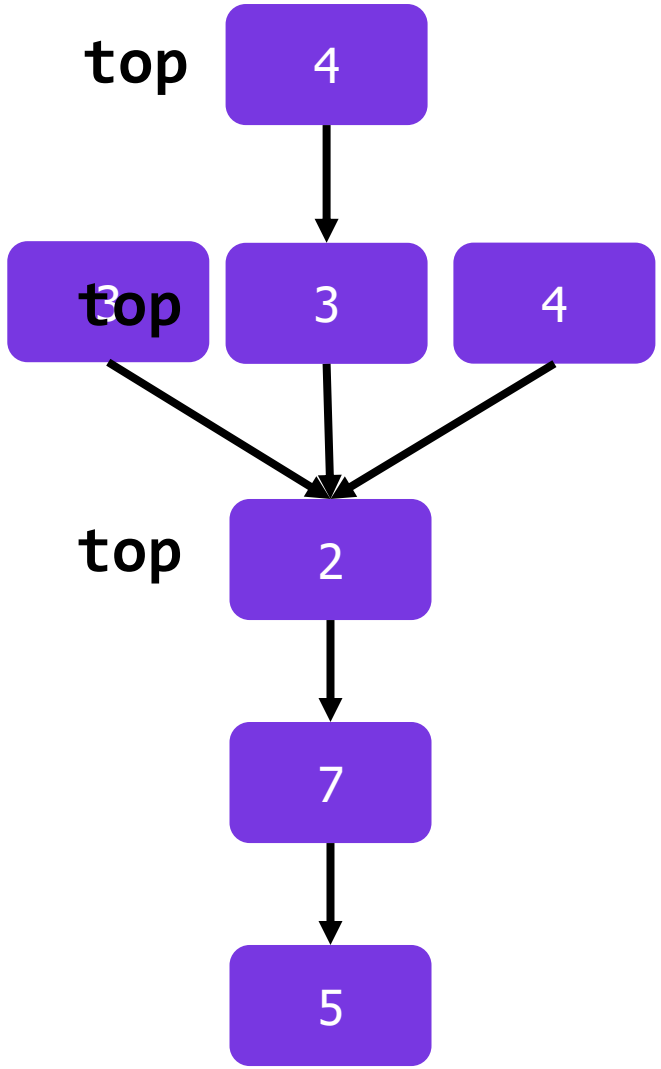
In practice, use a lock if it works and its fast enough…

- **Lock-free data structures** are an alternative. They can be more efficient than locks, especially under high contention
- They are extremely complicated

A Stack

```
type Node  { value, next : Node }
type Stack { top : atomic<Node> }
```

```
fun push(value):
  new_top = Node(value)
  do:
    old_top = top.load()      // read the current top node
    new_top.next ← old_top    // new top points to old one
  while (!top.cas(old_top, new_top))
```



Example: `push(4)` and `push(3)` **concurrently**

- `push(4)` **fails** its compare-and-swap, because the top changed
- It must reload the new top and then try again

`pop` is implemented similarly, using a **compare-and-swap loop**

Concurrency Bugs

- Code with data races may compile and appear to work…
- Incorrect results may occur spuriously, rarely, or catastrophically

Northeast Blackout of 2003: 55 million people in North America lost electricity for an extended period of time.

There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the application processes, they were both able to get write access to a data structure at the same time. And that corruption lead to the alarm event application getting into an infinite loop and spinning.

Engineers pored through one million lines of C and C++ code. It took eight weeks to locate the bug!

Summary

- Parallelism is hard. Concurrency is harder
- **Data races** are **undefined behavior**. Do not write them
- To write correct concurrent code, use synchronization primitives
  - **Locks** are simple to use, but avoid deadlock
  - **Lock-free data structures** using compare-and-swap can be more efficient but are much more complex
- Concurrency bugs are a nightmare