Concurrency

Alastair F. Donaldson

Aims of this lecture

- Cover fundamentals of **concurrent programming**
- Introduce the concepts of **threads**
- Discuss **race conditions** and **data races** between threads
- Introduce **synchronisation** via **locks** to avoid data races
- Discuss the problem of **deadlock** and how to avoid it

Coding demos

https://gitlab.doc.ic.ac.uk/afd/kotlin-2024-concurrency-demos.git

Acknowledgement

These slides are based on material from Antonio Fillieri and Nicolas Wu

What is concurrency?

- Multiple things executing at the same time

Or, more accurately:

- Multiple things executing, in an indeterminate order

Concurrency in computer systems

Distributed applications spanning multiple systems

- The internet
- Social media
- Streaming services

Concurrent processes on a single system

- Tons of stuff happening concurrently on your phone
- Many applications open on your computer

Concurrent threads within a process

- Tabs in a browser working concurrently
- IntelliJ indexing your project in the background

Our focus, but many of the concepts we will study apply more generally

Concurrency and parallelism

Concurrency (*logical parallelism*):

Composition of independently executing units

Could be nondeterministic

Parallelism (*physical parallelism*):

Efficient execution of multiple tasks on multiple processing units

Usually *deterministic*

Our focus

- Concurrency without parallelism is possible – e.g. single core CPUs, JavaScript
- Parallelism without concurrency is possible (e.g. pipelined CPUs, SIMD instructions)

Benefits

There are several reasons to study concurrency:

- **Abstraction** – separating different tasks without ordering execution (e.g. downloading multiple files)
- **Responsiveness** – providing a responsive program with different independent tasks (e.g. type code while it is compiled in the background)
- **Performance**: splitting a large task into multiple units and combining results (e.g. matrix multiplication)

Processes

A **process** is an independent unit of execution (roughly the abstraction of running a single program)

Characterised by:

- Identifier
- Memory space
- One or more **threads** of execution

The operating system schedules processes for execution on the available processor cores

Threads

A process can contain many **threads**

Each thread is characterised by:

- Identifier
- Program counter (the next statement to be executed)
- Local memory (separate for each thread)
- Global memory (shared with other threads)

Processes vs. threads: broad distinction

Processes: executing units that do not share memory

- e.g. IntelliJ, Chrome, Spotify

Threads: executing units that share memory

- e.g. threads controlling players and NPCs in a game

Launching a thread in Kotlin

`Runnable` is actually expressed in Java, but if it were written in Kotlin this is what it would look like

Kotlin (via Java) provides a `Runnable` interface:

```
interface Runnable {
    fun run()
}
```

A class that implements `Runnable` must provide a `run` method

```
class MyFriend : Runnable {
    override fun run() {
        println("Hello!")
    }
}
```

Launching a thread in Kotlin

Kotlin (via Java) has a `Thread` class that can be constructed with a `Runnable`

Creates a thread that, when started, will execute `run` on the provided `MyFriend` instance

```
fun main() {
    val myFirstThread = Thread(MyFriend())
    myFirstThread.start()
    myFirstThread.join()
}
```

Starts the thread

Waits for the thread to finish

Example: chatty threads

Let's write a program that launches several threads that will say (print) some words

We will use `Thread.sleep(…)` to inject some time delays between the threads' print statements

Coding demo

Observations

Our chatty threads program exhibited **nondeterministic** behaviour: the results were different on different executions

Nondeterminism is due to threads being scheduled on the cores of our machine in an order determined by the operating system and the programming language runtime system

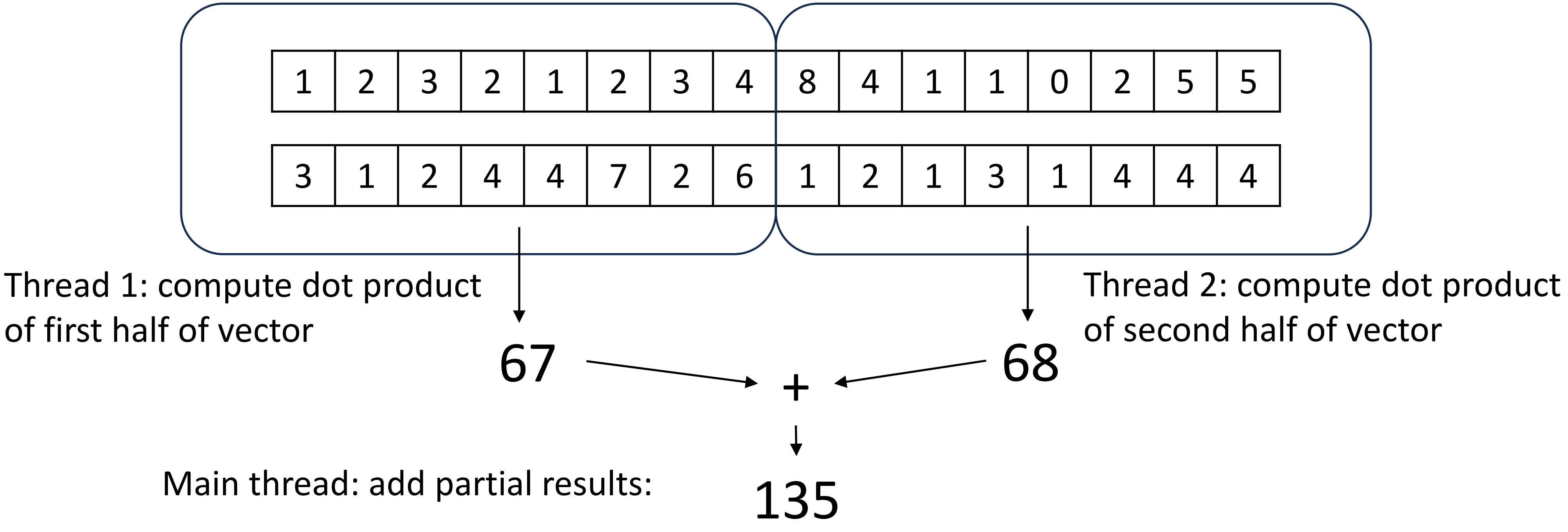
Challenges of concurrent programming:

- Our program must be designed to work correct for **any** interleaving of threads, no matter how rare
- Nondeterminism makes concurrent programs **hard to test**: some bugs only trigger for certain interleavings

Parallel dot product example

Dot product of two vectors: sum of pairwise products of elements

Computing dot product **in parallel**:



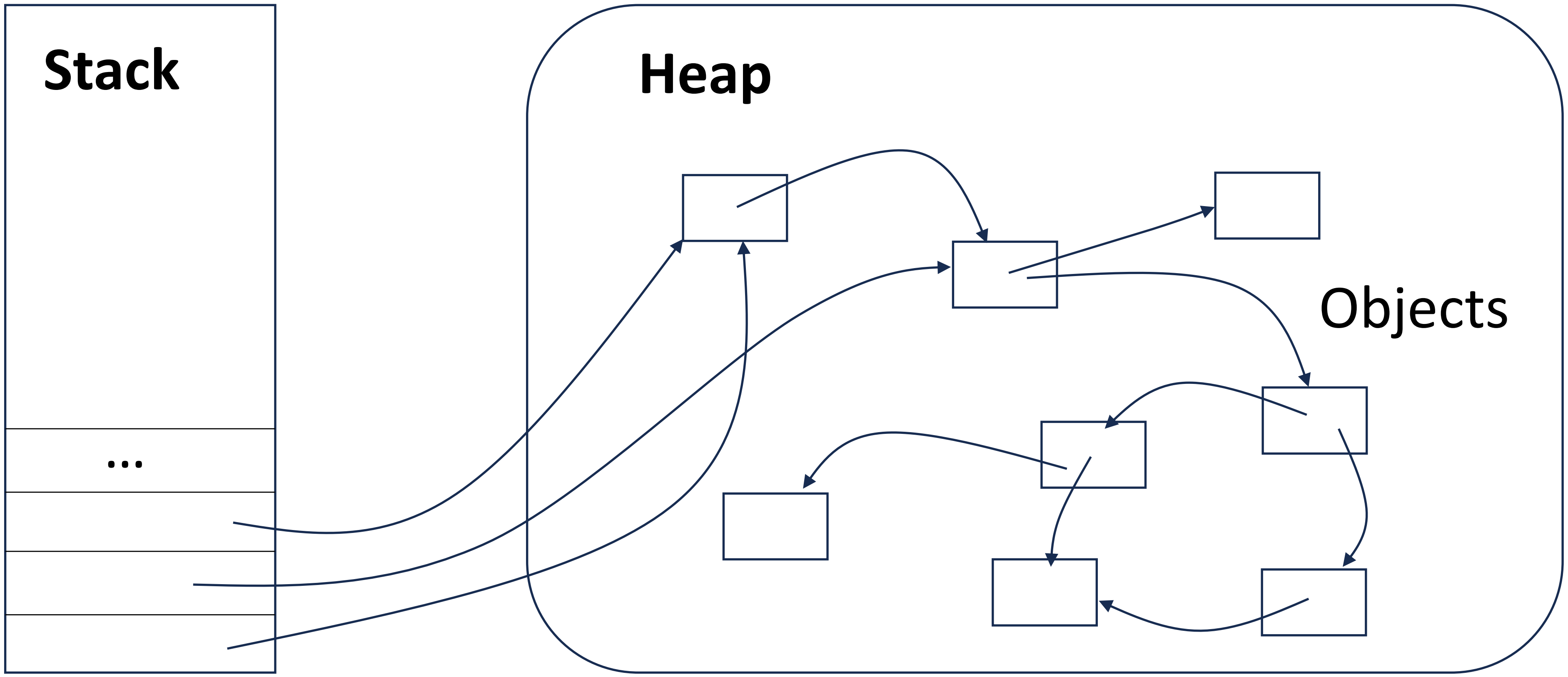
Parallel dot product

Coding demo

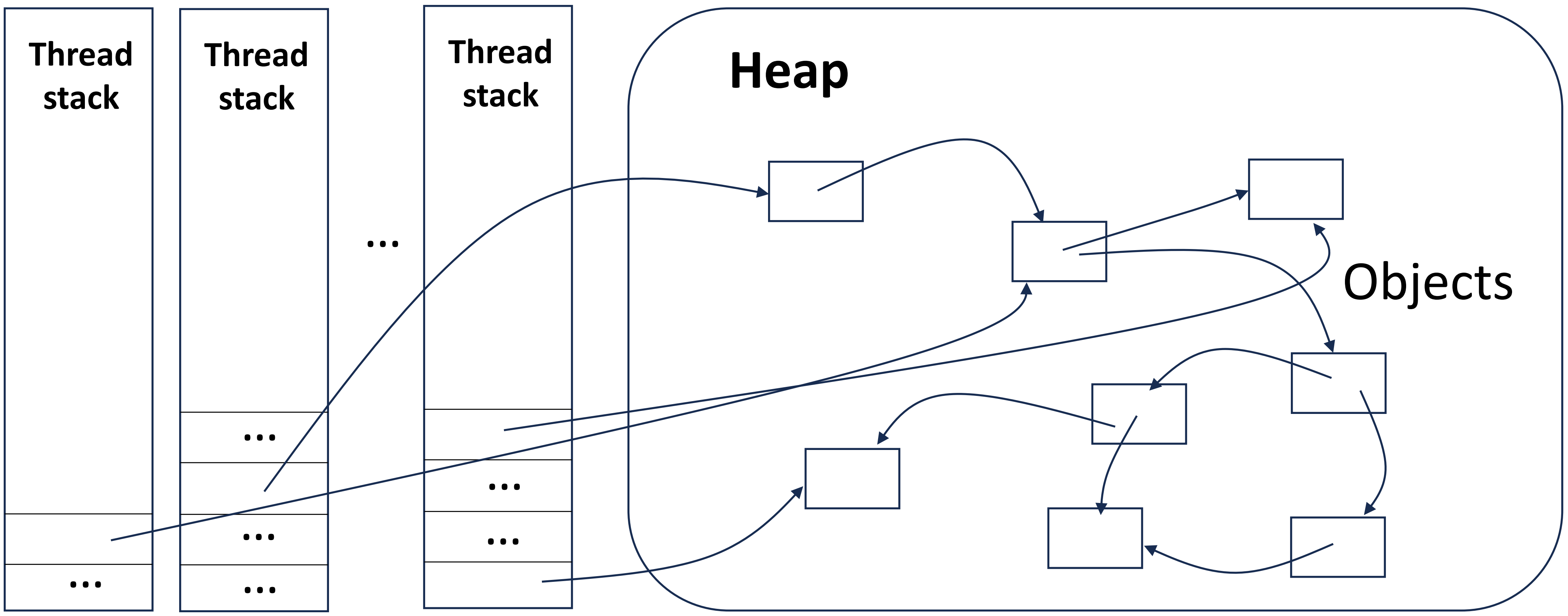
Parallel dot product: observations

- Launching threads requires **boiler-plate code**
- The parallel version gave slightly different results - floating-point arithmetic is not associative!
- Doubling the number of threads doesn’t necessarily double the speedup factor

Remember the stack and heap?



Every thread has its own stack



Race conditions

Concurrent programs are **nondeterministic**

- Multiple executions with the same input may lead to different behaviour
- This is the result of **interleaving** between threads
- The interleaving is decided by the **scheduler**, which you cannot control

Race condition: a situation where the result of a concurrent program depends on the specific execution imposed by the scheduler

Data races

A **data race** occurs when:

- Two threads access the same memory location
- At least one of the access is a write
- The accesses are not ordered by synchronisation

Locks (coming soon) provide a way to synchronise between threads and avoid data races

Data race example: concurrently incrementing a counter

Coding demo

Observations from counter example

```
fun inc() = value++
```

equivalent to

```
fun inc(): Int {
    val result = value
    value = result + 1
    return result
}
```

Multiple threads can be executing the body of `inc` simultaneously

Multiple threads may read the same value from `value`

Multiple threads may write the same value to `value` – leads to **lost increments**

Data races vs. race conditions

Not every race condition is a data race

- Online shop: race to buy last item
- Race conditions can also occur on other resources, e.g. filesystem or network

Data races are usually **unintended** race conditions

However, not every data race is a race condition

- The data race may not affect the result, e.g. two threads writing the same value to a location

Concurrent program for selling concert tickets

Larger coding demo

Observations from the ticket selling example

Some race conditions are natural and intentional

- Someone has to get the last Taylor Swift ticket!
- OK for this to be nondeterministic – down to timing

But the data races are **disastrous**

- Multiple copies of the same Taylor Swift ticket issued
- Exceptions being thrown due to concurrent modifications of mutable sets

Avoiding data races

Concurrent programming introduces:

- The **potential** for parallel execution (faster and better resource use)
- The **possibility** of acceptable race conditions
  - Computer games should be nondeterministic
  - Someone has to get the last concert ticket
  - Think about race conditions in IntellliJ
- The **risk** of unwanted data races

Let's look at how to avoid data races through **synchronisation**

Mutual exclusion

- A fundamental synchronisation problem arises whenever multiple threads have (mutable) access to a shared resource
- A resource’s **critical section** is the part of a program that accesses the shared resource
- The **mutual exclusion property** says that no more than one thread is in a resource’s critical section at any time

Critical section

```
fun inc(): Int {
    val result = value
    value = result + 1
    return result
}
```

Critical section

equivalent to

```
fun inc() = value++
```

Solving mutual exclusion

A fully satisfactory solution achieves these properties for an arbitrary number of threads sharing a resource:

- **Mutual exclusion:** at most one thread is in the critical section at any given time
- **Freedom from deadlocks:** if some threads try to enter the critical section then *some* will eventually succeed
- **Freedom from starvation:** if some threads try to enter the critical section then *all of them* will eventually succeed

Freedom from starvation implies freedom from deadlock

Deadlock

A mutual exclusion protocol provides **exclusive access** to shared resources to one thread at a time

Threads that try to access the resource when it is not available will have to block and wait until the resource becomes free

Deadlock: a situation where a group of threads wait forever because each of them is blocked waiting for one of the others to enter a critical section

Locks

Locks, also called **mutexes**, are special objects that a thread may use to acquire or release exclusive access to a critical section

In Kotlin (via Java) locks implement the interface `java.util.concurrent.locks.Lock`

```
interface Lock {
    fun lock()    // acquire lock
    fun unlock()  // release lock
}
```

Calls to `lock` and `unlock` are **atomic**: they cannot be interrupted

Sharing locks between threads

Several threads can share the same object `lock` (of type `Lock`)

If multiple threads call `lock.lock()` then exactly one thread will acquire the lock

- When some thread *t* calls `lock()` and it returns, then *t* is holding the lock
- All other threads will block on their call to `lock()` until *t* releases the lock
- The lock is released by *t* calling `unlock()`

Reentrant locks

Kotlin (via Java) has several classes that implement `Lock`

The most commonly-used implementation is `ReentrantLock`

A thread holding a `ReentrantLock` can lock it again without causing deadlock

```
val lock: Lock = ReentrantLock()
lock.lock()
lock.lock()
```

Protecting our counter with a lock

Coding demo

Locking a critical section

```
class Counter {
    private val lock: Lock = ReentrantLock()

    var value = 0
        private set

    fun inc(): Int {
        lock.lock()
        val result = value
        value++
        lock.unlock()
        return result
    }
}
```

Critical section is protected by the lock

Lock hygiene

Only the thread that holds a lock can release it

What happens if …

- The programmer makes a stupid error

```
fun inc(): Int {
    lock.lock()
    return value++
    lock.unlock()
}
```

Lock hygiene

Only the thread that holds a lock can release it

What happens if …

- The programmer makes a more subtle error

```
fun somethingComplex() {
    while (true) {
        lock.lock()
        if (true) {
            lock.unlock()
            return
        } else if (true) {
            break
        }
        lock.unlock()
    }
}
```

Lock hygiene

Only the thread that holds a lock can release it

What happens if …

- Code in the critical section throws an exception

```
fun writeData(…) {
    lock.lock()
    file.write(…)
    lock.unlock()
}
```

May throw an `IOException`

The *withLock* extension method

```
fun inc(): Int {
    lock.withLock {
        return value++
    }
}
```

Executes the code in the lambda body

Ensures that `lock` is acquired before execution of the lambda and released after

Ensures this **no matter what** – early returns and exceptions are covered

Protecting our counter using `withLock`

Coding demo

Using locks to make the concert tickets application work

Larger coding demo

Why are ReentrantLocks useful?

```
class ResizingArrayList<T> {
    …
    fun add(element: T) {
        …
    }

    fun add(index: Int, element: T) {
        if (index == size) {
            add(element)
            return
        }
        …
    }
    …
}
```

One `add` overload calls the other

Let’s make this class **thread-safe**

```kotlin
class ThreadSafeResizingArrayList<T> {
    …
    private val lock: Lock = ReentrantLock()

    fun add(element: T) {
        lock.withLock {
            …
        }
    }

    fun add(index: Int, element: T) {
        lock.withLock {
            if (index == size) {
                add(element)
                return
            }
            …
        }
    }
}
```

One `add` overload calls the other, while holding the lock

Without a reentrant lock, this would lead to **deadlock**

Deadlocks: the Coffman Conditions

The **Coffman Conditions** specify necessary conditions for a deadlock

- **Mutual exclusion:** threads may have exclusive access to the shared resource
- **Hold and wait:** a thread may request one resource while holding another
- **No pre-emption:** resources cannot be forcibly taken off threads that hold them
- **Circular wait:** two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding

Deadlock bank

```
class Bank {
    fun transfer(
        fromAccount: Account,
        toAccount: Account,
        amount: Int,
    ) {
        fromAccount.lock.withLock {
            toAccount.lock.withLock {
                if (fromAccount.balance >= amount) {
                    fromAccount.withdraw(amount)
                    toAccount.deposit(amount)
                }
            }
        }
    }
}
```

Breaking the cycle

```kotlin
class Bank {
    fun transfer(
        fromAccount: Account,
        toAccount: Account,
        amount: Int,
    ) {
        val (first, second) = if (
                fromAccount.accountNumber < toAccount.accountNumber
            ) {
            Pair(fromAccount, toAccount)
        } else {
            Pair(toAccount, fromAccount)
        }
        first.lock.withLock {
            second.lock.withLock {
                if (fromAccount.balance >= amount) {
                    fromAccount.withdraw(amount)
                    toAccount.deposit(amount)
                }
            }
        }
    }
}
```