Concurrency

Definition

- Multiple things executing, in an indeterminate order

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

Benefits

- 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 (not precise) 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

Kotlin (via Java) provides a Runnable interface:

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

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

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

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

fun main() {
    val myFirstThread = Thread(MyFriend())
    // Creates a thread that, when started, will execute run on the
provided              MyFriend instance

    myFirstThread.start()
    // Starts the thread
    myFirstThread.join()
    // Waits for the thread to finish
}
```

Example: chatty threads

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

```
class Chatterbox(
    private val id: Int,
    private val words: List<String>,
) : Runnable {
    override fun run() {
        Thread.sleep(1000)
        for (i in 1..200) { words.forEach { println("$id: $it");
Thread.sleep(1) } }
    }
}

fun main() {
    val cb1: Chatterbox = Chatterbox(1, listOf("hello", "I", "am",
"runnable"))
    val cb2: Chatterbox = Chatterbox(2, listOf("Hi", "I", "am",
"not"))
```

```
    val thread1: Thread = Thread(cb1)
    val thread2 = Thread(cb2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
}
```

Observations

Our 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.

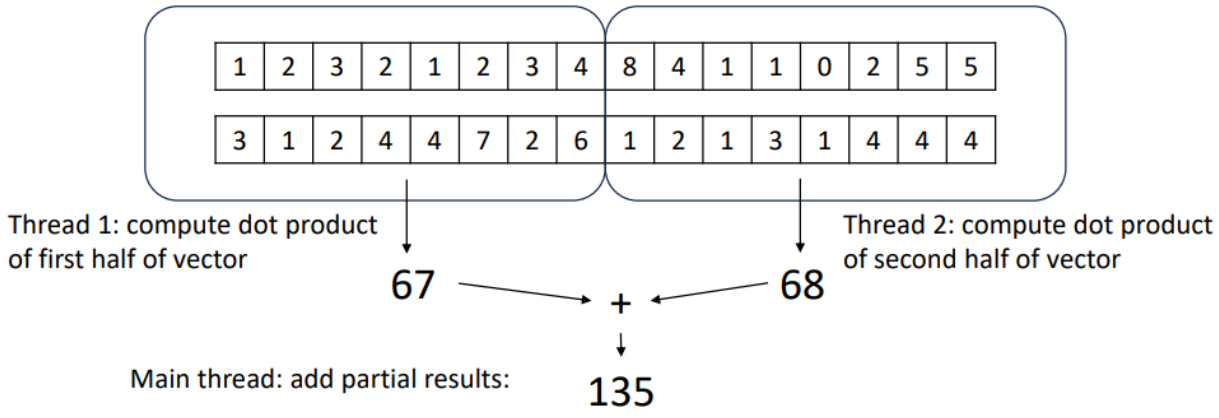
Challenges of concurrent programming:

- 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**:



```
fun sequentialDotProduct(
    array1: DoubleArray,
    array2: DoubleArray,
    ): Double {
    // Assume that array1 and array2 have the same length
    // Should really check this precondition
    var result: Double = 0.0
    for (i in 0..<array1.size) {
        result += array1[i] * array2[i]
    }
    return result
}

class DotProductWorker(
    private val array1: DoubleArray,
    private val array2: DoubleArray,
    private val startIndex: Int,
    private val numElements: Int
) : Runnable {
    var result: Double = 0.0
        private set
    override fun run() {
        for (i in 0..<numElements) {
            result += array1[startIndex + i] * array2[startIndex +
i]
        }
    }
}
fun parallelDotProduct(
    numThreads: Int,
    array1: DoubleArray,
    array2: DoubleArray,
): Double {
    // Assume arrays have the size that is a multiple of numThreads
    // Assume arrays have same size as each other
    val elementsPerThread: Int = array1.size / numThreads
    val workers = (0..<numThreads).map {
        DotProductWorker(array1, array2, it * elementsPerThread,
elementsPerThread)
    }
    val threads = workers.map {
        Thread(it)
    }
```

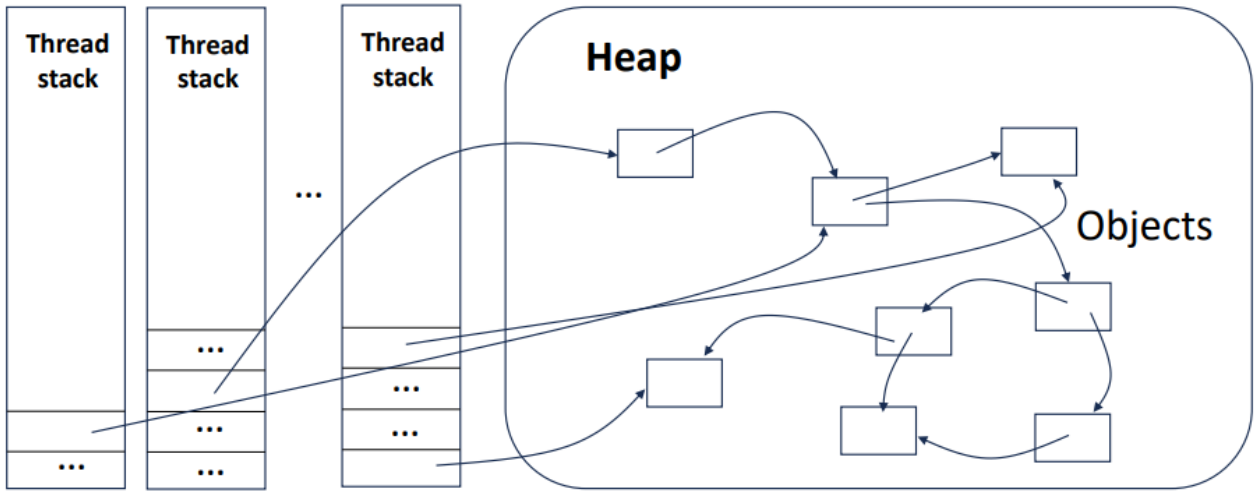
```
    threads.forEach {
        it.start()
    }
    threads.forEach {
        it.join()
    }
    return workers.map {
        it.result
    }.reduce(Double::plus)
}

fun main() {
    val length: Int = 16777216 * 4
    val numThreads: Int = 4
    val array1 = randomDoubleArray(length)
    val array2 = randomDoubleArray(length)
    var dotProductSequential: Double?
    val sequentialTime = measureTime {
         dotProductSequential = sequentialDotProduct(array1,
array2)
    }
    var dotProductParallel: Double?
    val parallelTime = measureTime {
        dotProductParallel = parallelDotProduct(numThreads, array1,
array2)
    }
    println("Sequential result: $dotProductSequential")
    println("Sequential time: $sequentialTime")
    println("Parallel result: $dotProductParallel")
    println("Parallel time: $parallelTime")
}
private fun randomDoubleArray(length: Int) = DoubleArray(length) {
    Random.nextDouble() - 0.5
}
```

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

Floating Point arithmetic is not associative!

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

```
class Counter {
    var value: Int = 0
        private set
    fun inc(): Int = value++
}

class Incrementer(
    private val counter: Counter,
    private val numIncrements: Int,
) : Runnable {
    private val _observedValues: MutableSet<Int> = mutableSetOf()
    val observedValues: Set<Int>
        get() = _observedValues
    override fun run() {
        for (i in 1..numIncrements) {
            _observedValues.add(counter.inc())
        }
    }
}

fun main() {
    val counter = Counter()
    val incrementer1 = Incrementer(counter, 500)
    val incrementer2 = Incrementer(counter, 500)
    val thread1 = Thread(incrementer1)
    val thread2 = Thread(incrementer2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    println(counter.value)
    println(incrementer1.observedValues intersect
incrementer2.observedValues)
}
```

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

```
class MusicFan(
    private val name: String,
    private val favouriteArtist: String,
    private val acceptableVenues: List<String>,
    private var money: Int,
    private val ticketSeller: TicketSeller) : Runnable {
    override fun run() {
        for (attempt in 1..3) {
            println("First attempt by $name")
            for (venue in acceptableVenues) {
                println("$name is looking for tickets to see
$favouriteArtist at $venue")
                val maybeOffer: TicketSeller.Offer? =
                    ticketSeller.requestTicket(
                        customer = name,
```

```
                        artist = favouriteArtist,
                        venue = venue)
                if (maybeOffer == null) {
                    println("No tickets for $name see
$favouriteArtist at $venue")
                    continue
                }
                println("$name got an offer to see $favouriteArtist
at $venue")
                if (!decideToAccept(maybeOffer)) {
                    println("$name rejected offer to see
$favouriteArtist at $venue")
                    maybeOffer.reject()
                    continue
                }
                val maybeTicket = maybeOffer.accept()
                if (maybeTicket != null) {
                    println("$name accepted offer to see
$favouriteArtist at $venue")
                    money -= maybeOffer.price
                    println("$name got a ticket to see
$favouriteArtist at $venue!")
                    return
                }
                println("$name was too late in accepting offer to
see $favouriteArtist at $venue")
            }
        }
    }
    private fun decideToAccept(offer: TicketSeller.Offer): Boolean
{
        if (money < offer.price) {
            return false
        }
        // Could have some decision making logic here.
        // But for simplicity, our music fans will always accept
offers
        Thread.sleep(100)
        return true
    }
}

class Ticket(val id: Int, val artist: String, val venue: String,
val price: Int)
```

```
class TicketSeller {
    inner class Offer(private val customer: String, private val
ticket: Ticket) {
        fun accept(): Ticket? {
            salesRecord.add(Pair(ticket, customer))
            return ticket
        }
        fun reject() {
            availableTickets.add(ticket)
        }
        val price: Int
            get() = ticket.price
    }
    private var nextTicketId: Int = 0
    private val availableTickets: MutableSet<Ticket> =
mutableSetOf()
    // The string component of each pair is the person who bought
the ticket
    private val salesRecord: MutableList<Pair<Ticket, String>> =
mutableListOf()
    fun addTicket(artist: String, venue: String, price: Int) {
        availableTickets.add(Ticket(nextTicketId, artist, venue,
price))
        nextTicketId++
    }
    fun requestTicket(customer: String, artist: String, venue:
String): Offer? {
        val potentialTickets = availableTickets.filter {
            it.artist == artist && it.venue == venue
        }
        if (potentialTickets.isEmpty()) {
            return null
        }
        val ticketToOffer = potentialTickets[0]
        availableTickets.remove(ticketToOffer)
        return Offer(customer, ticketToOffer)
    }
    fun report(artist: String) {
        val numTicketsSoldForArtist = salesRecord.map {
            it.first
        }.map {
            it.artist
        }.filter {
```

```
            it == artist
        }.size
        println("$numTicketsSoldForArtist were sold for $artist")
    }
}

fun main() {
    val seller = TicketSeller()
    for (i in 0..<100) {
        seller.addTicket("Def Leppard", "London Wembley", 100)
        seller.addTicket("Def Leppard", "Birmingham NEC", 80)
        seller.addTicket("Def Leppard", "Glasgow SECC", 60)
    }
    for (i in 0..<10) {
        seller.addTicket("Taylor Swift", "London Wembley", 10000)
        seller.addTicket("Taylor Swift", "Birmingham NEC", 10000)
    }
    val musicFans: MutableList<MusicFan> = mutableListOf()
    musicFans.add(MusicFan("Ally", "Def Leppard",
        listOf("London Wembley", "Glasgow SECC"), 10000,
        seller))
    for (i in 1..100) {
        musicFans.add(MusicFan("Swift fan $i", "Taylor Swift",
            listOf("London Wembley", "Birmingham NEC", "Hyde Park",
"Glasgow SECC"),
            i * 2000,
            seller
            ))
    }
    val threads = musicFans.map {
        Thread(it)
    }
    threads.forEach { it.start() }
    threads.forEach { it.join() }
    seller.report("Def Leppard")
    seller.report("Taylor Swift")
}
```

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

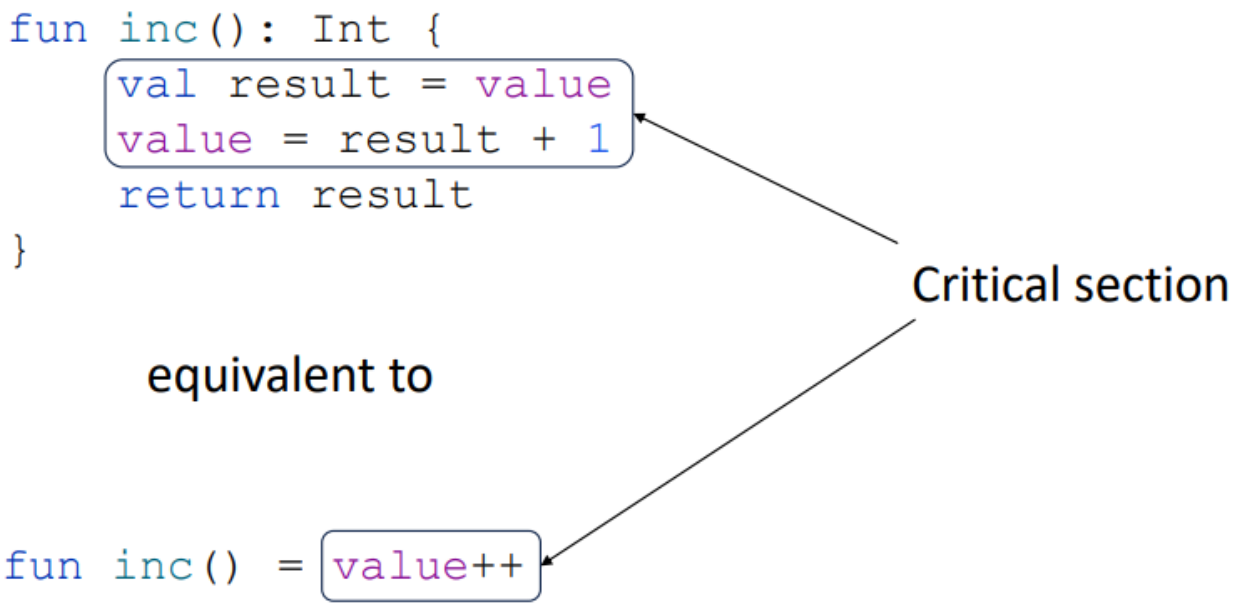
Concurrent programming introduces:

- 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 IntelliJ
- The risk of unwanted data races

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



Solving mutual exclusion

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

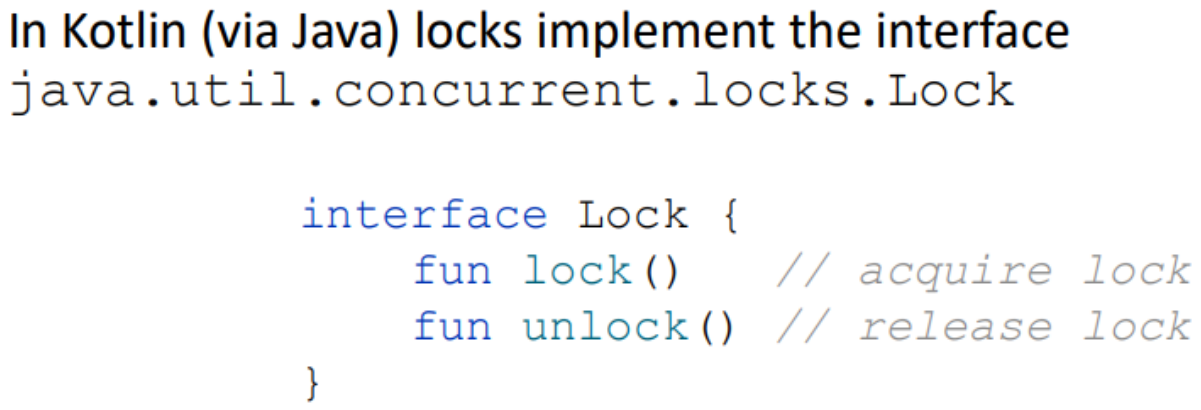
- Mutual exclusion: at most one thread is in the critical section at any given time
- Freedom from deadlocks: if some threads try to enter the critical section then some will eventually succeed
- Freedom from starvation: if some threads try to enter the critical section then all of them will eventually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



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()
```

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.

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`

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

```
    var value: Int = 0
        private set
    fun inc(): Int {
        lock.withLock {
            val result = value
            value++
            return result
        } // The lock is guaranteed to be unlocked, thanks to
withLock
    }
}

class Incrementer(
    private val counter: Counter,
    private val numIncrements: Int,
) : Runnable {
    private val _observedValues: MutableSet<Int> = mutableSetOf()
    val observedValues: Set<Int>
        get() = _observedValues
    override fun run() {
        for (i in 1..numIncrements) {
            _observedValues.add(counter.inc())
        }
    }
}

fun main() {
    val counter = Counter()
    val incrementer1 = Incrementer(counter, 500)
    val incrementer2 = Incrementer(counter, 500)
    val thread1 = Thread(incrementer1)
    val thread2 = Thread(incrementer2)
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    println(counter.value)
    println(incrementer1.observedValues intersect
incrementer2.observedValues)
    // We still have race conditions - these will print
nondeterministic values.
    // But we got rid of the data races!
    println(incrementer1.observedValues.sorted()[0])
```

```
    println(incrementer2.observedValues.sorted()[0])
}
```

Using locks to make the concert tickets application work

```
class MusicFan(
    private val name: String,
    private val favouriteArtist: String,
    private val acceptableVenues: List<String>,
    private var money: Int,
    private val ticketSeller: TicketSeller) : Runnable {
    override fun run() {
        for (attempt in 1..3) {
            println("First attempt by $name")
            for (venue in acceptableVenues) {
                println("$name is looking for tickets to see
$favouriteArtist at $venue")
                val maybeOffer: TicketSeller.Offer? =
                    ticketSeller.requestTicket(
                        customer = name,
                        artist = favouriteArtist,
                        venue = venue)
                if (maybeOffer == null) {
                    println("No tickets for $name see
$favouriteArtist at $venue")
                    continue
                }
                println("$name got an offer to see $favouriteArtist
at $venue")
                if (!decideToAccept(maybeOffer)) {
                    println("$name rejected offer to see
$favouriteArtist at $venue")
                    maybeOffer.reject()
                    continue
                }
                val maybeTicket = maybeOffer.accept()
                if (maybeTicket != null) {
                    println("$name accepted offer to see
$favouriteArtist at $venue")
                    money -= maybeOffer.price
                    println("$name got a ticket to see
```

```
$favouriteArtist at $venue!")
                    return
                }
                println("$name was too late in accepting offer to
see $favouriteArtist at $venue")
            }
        }
    }
    private fun decideToAccept(offer: TicketSeller.Offer): Boolean
{
        if (money < offer.price) {
            return false
        }
        // Could have some decision making logic here.
        // But for simplicity, our music fans will always accept
offers
        Thread.sleep(100)
        return true
    }
}

class Ticket(val id: Int, val artist: String, val venue: String,
val price: Int)

class TicketSeller {
    inner class Offer(private val customer: String, private val
ticket: Ticket) {
        fun accept(): Ticket? {
            lock.withLock {
                salesRecord.add(Pair(ticket, customer))
                return ticket
            }
        }
        fun reject() {
            lock.withLock {
                availableTickets.add(ticket)
            }
        }
        // No need for critical section here as the ticket seller
is
        // not being accessed
        val price: Int
            get() = ticket.price
    }
```

```
    private var nextTicketId: Int = 0
    private val availableTickets: MutableSet<Ticket> =
mutableSetOf()
    // The string component of each pair is the person who bought
the ticket
    private val salesRecord: MutableList<Pair<Ticket, String>> =
mutableListOf()
    private val lock: Lock = ReentrantLock()
    fun addTicket(artist: String, venue: String, price: Int) {
        lock.withLock {
            availableTickets.add(Ticket(nextTicketId, artist,
venue, price))
            nextTicketId++
        }
    }
    fun requestTicket(customer: String, artist: String, venue:
String): Offer? {
        lock.withLock {
            val potentialTickets = availableTickets.filter {
                it.artist == artist && it.venue == venue
            }
            if (potentialTickets.isEmpty()) {
                return null
            }
            val ticketToOffer = potentialTickets[0]
            availableTickets.remove(ticketToOffer)
            return Offer(customer, ticketToOffer)
        }
    }
    fun report(artist: String) {
        // Reporting sales should not be done while shop is open -
no critical section.
        val numTicketsSoldForArtist = salesRecord.map {
            it.first
        }.map {
            it.artist
        }.filter {
            it == artist
        }.size
        println("$numTicketsSoldForArtist were sold for $artist")
    }
}

fun main() {
```

```
    val seller = TicketSeller()
    for (i in 0..<100) {
        seller.addTicket("Def Leppard", "London Wembley", 100)
        seller.addTicket("Def Leppard", "Birmingham NEC", 80)
        seller.addTicket("Def Leppard", "Glasgow SECC", 60)
    }
    for (i in 0..<10) {
        seller.addTicket("Taylor Swift", "London Wembley", 10000)
        seller.addTicket("Taylor Swift", "Birmingham NEC", 10000)
    }
    val musicFans: MutableList<MusicFan> = mutableListOf()
    musicFans.add(MusicFan("Ally", "Def Leppard",
        listOf("London Wembley", "Glasgow SECC"), 10000,
        seller))
    for (i in 1..100) {
        musicFans.add(MusicFan("Swift fan $i", "Taylor Swift",
            listOf("London Wembley", "Birmingham NEC", "Hyde Park",
"Glasgow SECC"),
            i * 2000,
            seller
            ))
    }
    val threads = musicFans.map {
        Thread(it)
    }
    threads.forEach { it.start() }
    threads.forEach { it.join() }
    seller.report("Def Leppard")
    seller.report("Taylor Swift")
}
```

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**

```
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

```
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)
                }
            }
        }
    }
}
```

Breaking the

Addition of Offer Expiry

```
class TicketSeller {
    inner class Offer(private val customer: String,
                      private val ticket: Ticket,
    ) : Runnable {
        init {
            Thread(this).start()
        }
        override fun run() {
            // It would be better to use a constant here, rather
than a
            // magic number.
            Thread.sleep(100)
            reject()
        }
        private var stillAvailable: Boolean = true
        fun accept(): Ticket? {
            lock.withLock {
                if (stillAvailable) {
                    salesRecord.add(Pair(ticket, customer))
                    return ticket
                }
                stillAvailable = false
                return null
            }
        }
        fun reject() {
            lock.withLock {
                if (stillAvailable) {
                    stillAvailable = false
                    availableTickets.add(ticket)
                }
            }
        }
        // No need for critical section here as the ticket seller
is
        // not being accessed
        val price: Int
            get() = ticket.price
    }
    private var nextTicketId: Int = 0
    private val availableTickets: MutableSet<Ticket> =
```

```
mutableSetOf()
    // The string component of each pair is the person who bought
the ticket
    private val salesRecord: MutableList<Pair<Ticket, String>> =
mutableListOf()
    private val lock: Lock = ReentrantLock()
    fun addTicket(artist: String, venue: String, price: Int) {
        lock.withLock {
            availableTickets.add(Ticket(nextTicketId, artist,
venue, price))
            nextTicketId++
        }
    }
    fun requestTicket(customer: String, artist: String, venue:
String): Offer? {
        lock.withLock {
            val potentialTickets = availableTickets.filter {
                it.artist == artist && it.venue == venue
            }
            if (potentialTickets.isEmpty()) {
                return null
            }
            val ticketToOffer = potentialTickets[0]
            availableTickets.remove(ticketToOffer)
            return Offer(customer, ticketToOffer)
        }
    }
    fun report(artist: String) {
        // Reporting sales should not be done while shop is open -
no critical section.
        val numTicketsSoldForArtist = salesRecord.map {
            it.first
        }.map {
            it.artist
        }.filter {
            it == artist
        }.size
        println("$numTicketsSoldForArtist were sold for $artist")
    }
}

class MusicFan(
    private val name: String,
    private val favouriteArtist: String,
```

```
    private val acceptableVenues: List<String>,
    private var money: Int,
    private val ticketSeller: TicketSeller) : Runnable {
    override fun run() {
        for (attempt in 1..3) {
            println("First attempt by $name")
            for (venue in acceptableVenues) {
                println("$name is looking for tickets to see
$favouriteArtist at $venue")
                val maybeOffer: TicketSeller.Offer? =
                    ticketSeller.requestTicket(
                        customer = name,
                        artist = favouriteArtist,
                        venue = venue)
                if (maybeOffer == null) {
                    println("No tickets for $name see
$favouriteArtist at $venue")
                    continue
                }
                println("$name got an offer to see $favouriteArtist
at $venue")
                if (!decideToAccept(maybeOffer)) {
                    println("$name rejected offer to see
$favouriteArtist at $venue")
                    maybeOffer.reject()
                    continue
                }
                val maybeTicket = maybeOffer.accept()
                if (maybeTicket != null) {
                    println("$name accepted offer to see
$favouriteArtist at $venue")
                    money -= maybeOffer.price
                    println("$name got a ticket to see
$favouriteArtist at $venue!")
                    return
                }
                println("$name was too late in accepting offer to
see $favouriteArtist at $venue")
            }
        }
    }
    private fun decideToAccept(offer: TicketSeller.Offer): Boolean
{
        if (money < offer.price) {
```

```
            return false
        }
        // Could have some decision making logic here.
        // But for simplicity, our music fans will always accept
offers
        Thread.sleep(100)
        return true
    }
}

class Ticket(val id: Int, val artist: String, val venue: String,
val price: Int)

fun main() {
    val seller = TicketSeller()
    for (i in 0..<100) {
        seller.addTicket("Def Leppard", "London Wembley", 100)
        seller.addTicket("Def Leppard", "Birmingham NEC", 80)
        seller.addTicket("Def Leppard", "Glasgow SECC", 60)
    }
    for (i in 0..<10) {
        seller.addTicket("Taylor Swift", "London Wembley", 10000)
        seller.addTicket("Taylor Swift", "Birmingham NEC", 10000)
    }
    val musicFans: MutableList<MusicFan> = mutableListOf()
    musicFans.add(MusicFan("Ally", "Def Leppard",
        listOf("London Wembley", "Glasgow SECC"), 10000,
        seller))
    for (i in 1..100) {
        musicFans.add(MusicFan("Swift fan $i", "Taylor Swift",
            listOf("London Wembley", "Birmingham NEC", "Hyde Park",
"Glasgow SECC"),
            i * 2000,
            seller
            ))
    }
    val threads = musicFans.map {
        Thread(it)
    }
    threads.forEach { it.start() }
    threads.forEach { it.join() }
    seller.report("Def Leppard")
    seller.report("Taylor Swift")
}
```