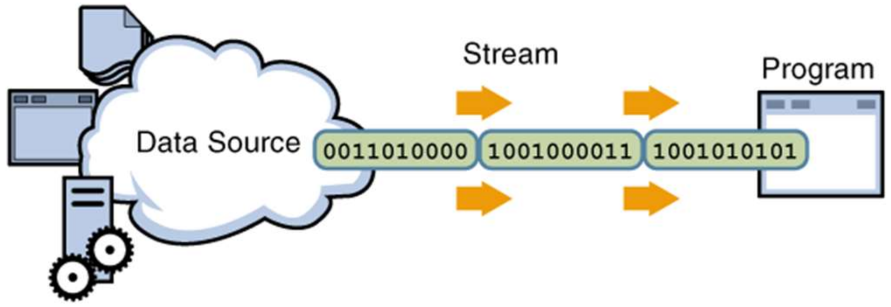
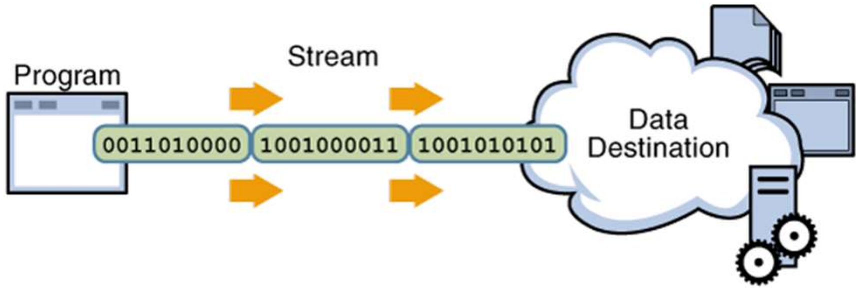
Sensor Based Mobile Applications TX00CK66

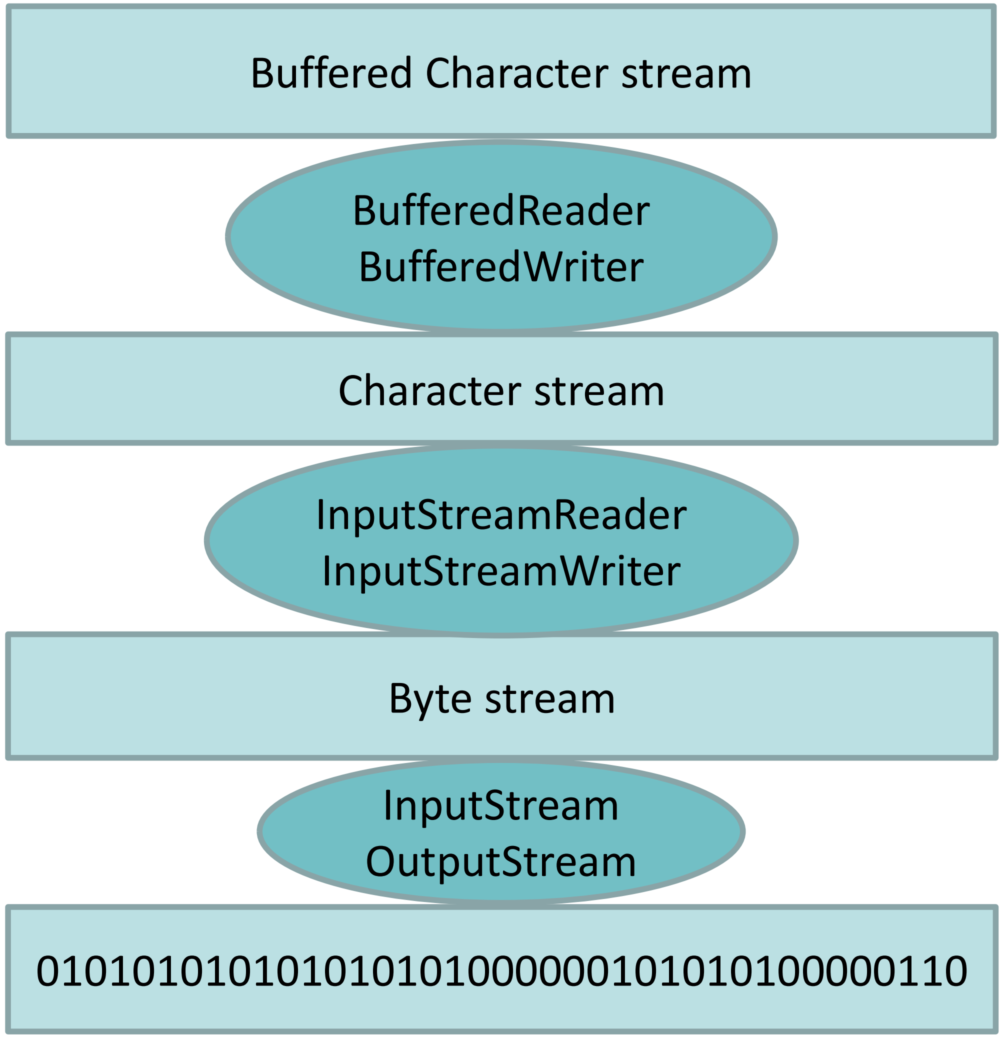
Networking, Multithreading

Jarkko.Vuori@metropolia.fi

Network communication







Using networking capabilities

Create URL object instance

```
val myUrl = URL("https://www.a.com/data.txt")
```

Create URLConnection instance

```
val myConn = myUrl.openConnection()
```

Get the input stream

```
val istream = myConn.getInputStream()
```

Read the stream with reader

```
val allText = istream.bufferedReader().use{it.readText()}
val result = StringBuilder()
result.append(allText)
val str = result.toString()
```

use() executes the given block function on this resource and then closes it down correctly whether an exception is thrown or not

```
<uses-permission android:name="android.permission.INTERNET" />
```

Network Availability

- ConnectivityManager class answers queries about the state of network connectivity. It also notifies applications when network connectivity changes (https://developer.android.com/reference/android/net/ConnectivityManager)
- isDefaultNetworkActive() – method returns Boolean value whether the data network is currently active or not

```
private fun isNetworkAvailable(): Boolean =
    (this.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager).isDefaultNetworkActive
```

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Network Security

- Use TSL/SSL traffic
  - Current TSL version is 1.3
- If your app communicates with a web server that has a certificate issued by a well-known, trusted CA, use HTTPS protocol instead of HTTP
- Since API 23 there has been cleartext attribute (in Manifest):

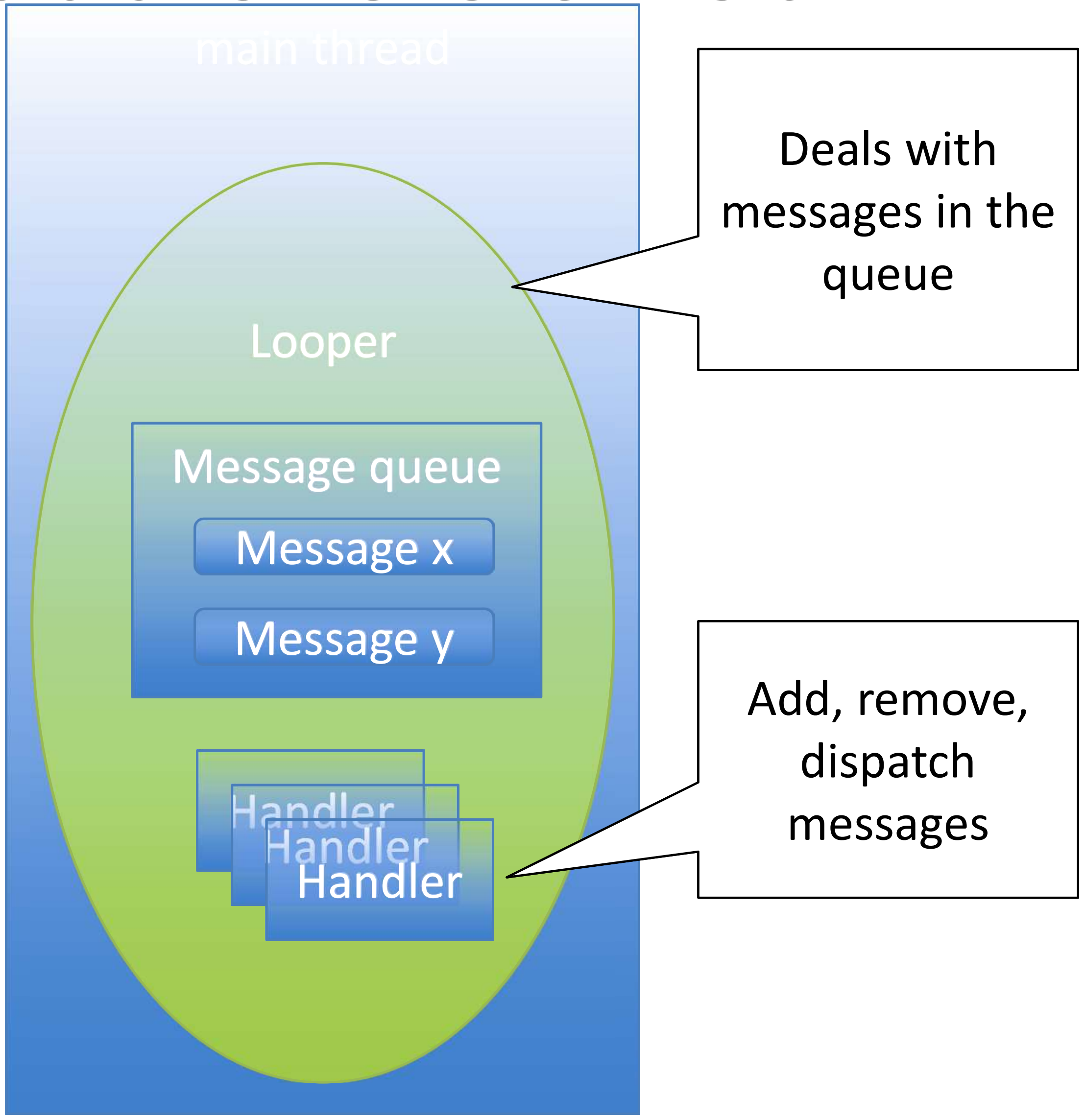
  ```
  android:usesCleartextTraffic="false"
  ```
- The attribute declares whether the app intends to use cleartext network traffic (e.g., HTTP, WebSockets, XMPP, SMTP, IMAP – without TLS or STARTTLS)
- See https://developer.android.com/training/articles/security-ssl

Do not run time consuming operations in the main thread

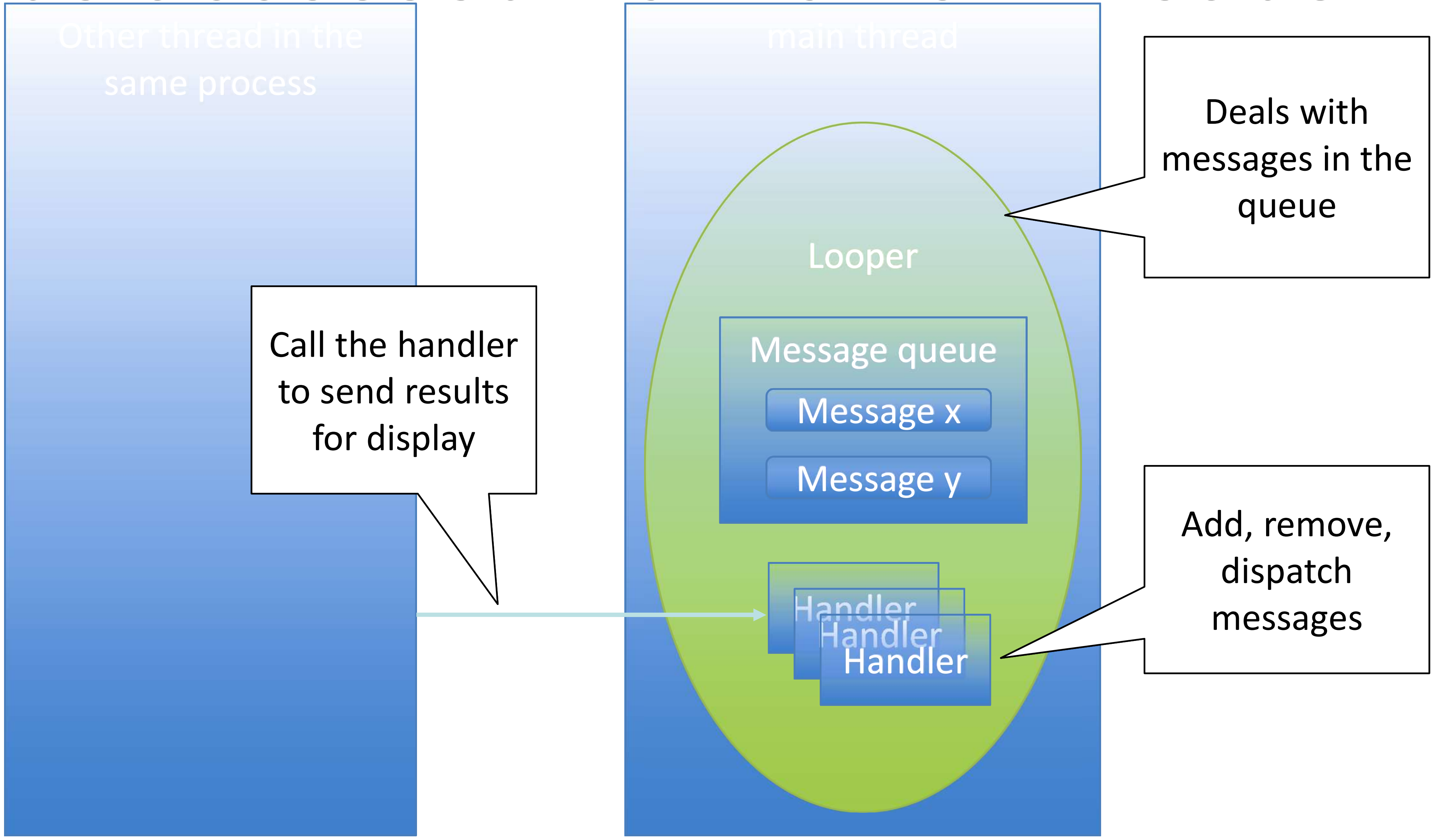
- Network communication is considered to be a slow operation
  - Server responses are normally in the range of 5ms – 2s
- When an application starts, Android operating system (Linux) creates a new process
  - In this way separate applications are totally isolated
  - One Linux process can contain more than one thread, the first thread (main thread) is created for the UI processing
- Never block the main (UI) thread because the application does not then response to the user input, use
  - Multithreading + Handler()
  - Coroutines
  - ~~AsyncTask~~ (deprecated in API level 30)

What happens when Android application module is started?

- If no application components are executing, a new process is created
- Main thread created
- onCreate()-method gets executed (call back)
- Looper with a Message queue is set up

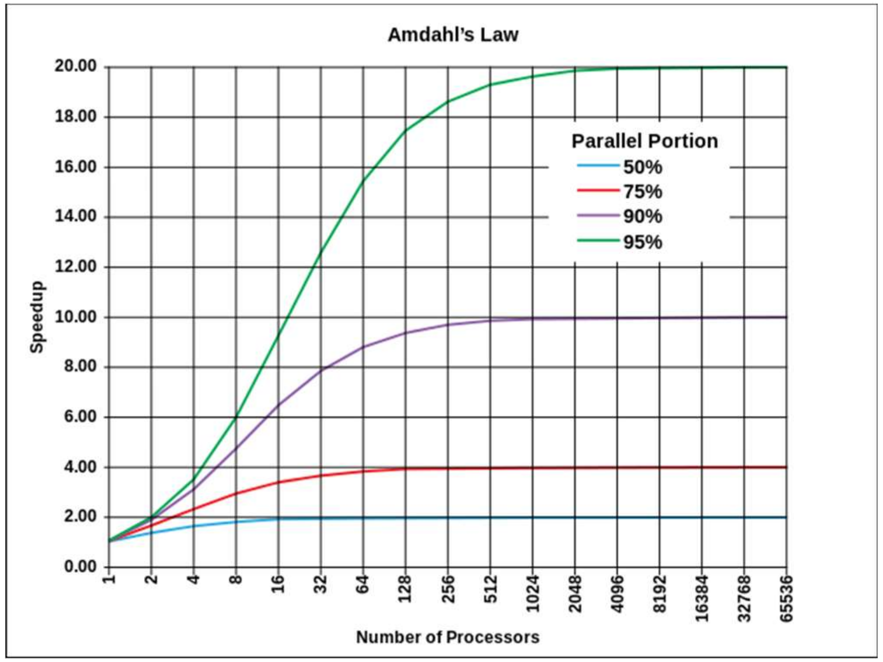


Handler in the main thread can be accessed from other threads


Other thread in the same process
main thread
Deals with messages in the queue
Looper
Call the handler to send results for display
Message queue
Message x
Message y
Handler
Handler
Handler
Add, remove, dispatch messages

Amdahl’s law

- It is possible to calculate the equation y(x) = sin(5x) + cos(9x) + exp(x/3) in parallel
  - if you have more than one calculation unit available, calculating sin(), cos() and exp() in three parallel calculation units speeds up the calculation of the equation y(x)
- It is not always possible to run things in parallel, e.g. the equation y(x) = sin(5cos(9exp(x/3))) cannot run in parallel because terms depend on each other
- Amdahl’s law: In parallelization, if P is the proportion of a system or program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speedup that can be achieved using N number of processors is 1/((1-P)+(P/N))
  - An example: you have two processors (N=2) and all your program can be parallelized (P=1), then the speedup is 1/(0+½)=2
  - An another example: you have four processors (N=4) and half of your program can be parallelized (P=0.5), then the speedup is 1/(0.5+0.5/4)=1.6



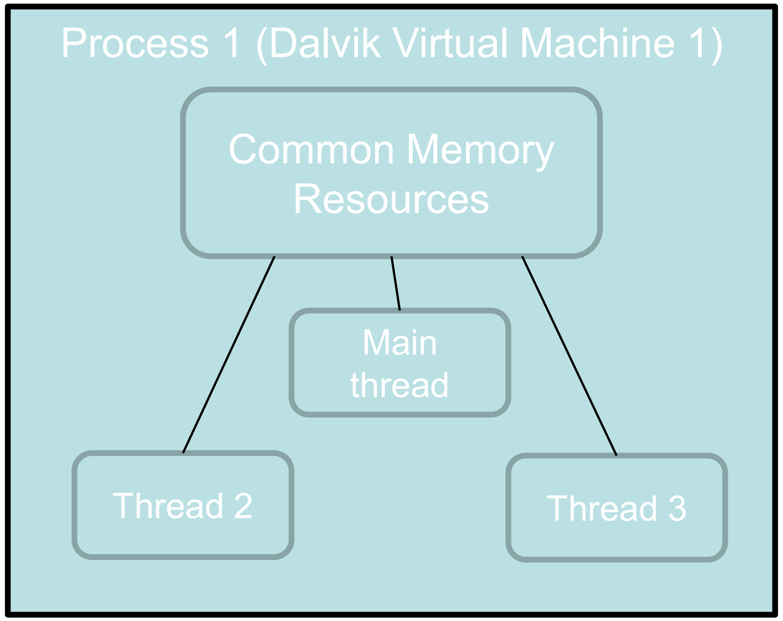
$$\lim_{n\to\infty} S = \frac{1}{1-p}$$

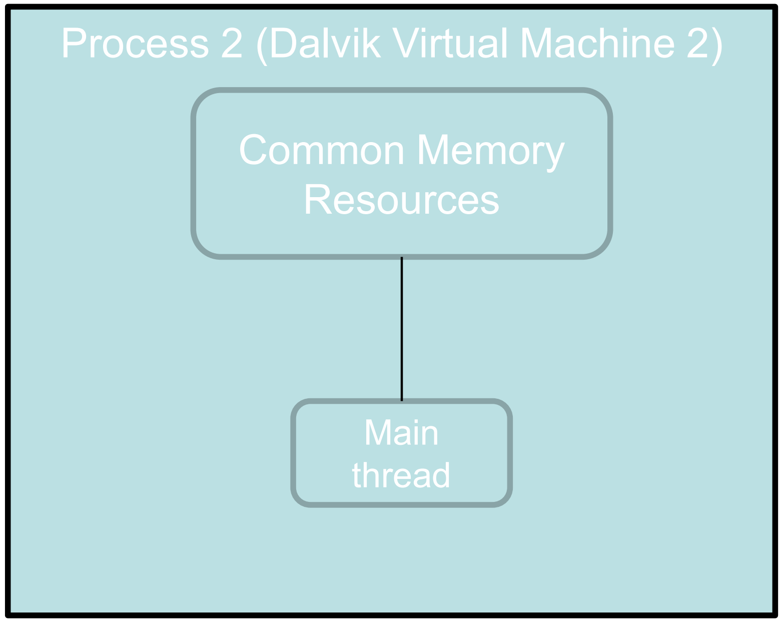
Threads

- Threads are used in creating applications that perform multiple tasks in a manner that appears simultaneous
  - Start a download in browser, continue browsing simultaneously
    - The threads are scheduled so that their execution interleaves, all threads are making some progress all the time
    - So, user does not need to wait for download to complete before starting other tasks
- In most modern computers there is a possibility for a real concurrent execution
  - Processors have multiple core units, sharing a single memory
  - By default, all activity in an application happens in a single thread, i.e., sequentially
  - To utilize multiple core units in a single application one needs to use multiple threads - the scheduler will allocate threads to separate cores when possible (real parallelism)

Process vs thread

- Process with a single thread: an execution sequence and a private memory area
  - (memory can be often shared with other processors, though)
- Thread: an execution sequence without a private memory area (memory is assumed to be shared with other threads)
  - (call stack is not shared, however)





Kotlin threads

- Building blocks for threads in Java:
  - A class that implements Runnable interface. All execution happens in the `run()` method of an instance of this class.
  - A Thread instance to which the runnable object is attached
- Some methods in Thread class
  - `start()` - start the execution of thread, in practice start executing `run()` method in the Runnable instance associated with the thread
  - `sleep()` - cease the execution of the thread for a specified amount of milliseconds. Other threads can be scheduled to be run
  - `join()` - wait for the execution of thread to complete
    - all these possibly inside a try-catch block
- There is also Kotlin extension function for creating threads
  -
    ```
    thread() {
        println("${Thread.currentThread()} has run.")
    }
    ```

Kotlin threads

- Start the execution of the `run()` method of the Runnable object attached to a thread by calling `start()` method of the thread
  - Calling `start()` is asynchronous - call returns as soon as thread execution is started
  - `run()` method is executed in parallel with the thread that called `start()`
  - After `run()` finishes the thread is dead, it is not possible to restart it
- Kotlin extension function creates and calls the run function automatically

Kotlin threads

```
class Account {
    private var amount: Double = 0.0

    fun deposit(amount: Double) {
        val x = this.amount
        Thread.sleep(1)    // simulates 1ms of processing time
        this.amount = x + amount
    }

    fun saldo(): Double {
        return amount
    }
}
```

```
val tili = Account()

val thread1 = thread() {
    for (i in 1..1000)
        tili.deposit(1.0)
}

val thread2 = thread() {
    for (i in 1..1000)
        tili.deposit(1.0)
}

thread1.join(); thread2.join()
Log.i("DBG", ”Account saldo is ${tili.saldo()}")
```

Here we create two threads running simultaneously, both adding one unit of money to the account one thousand times.

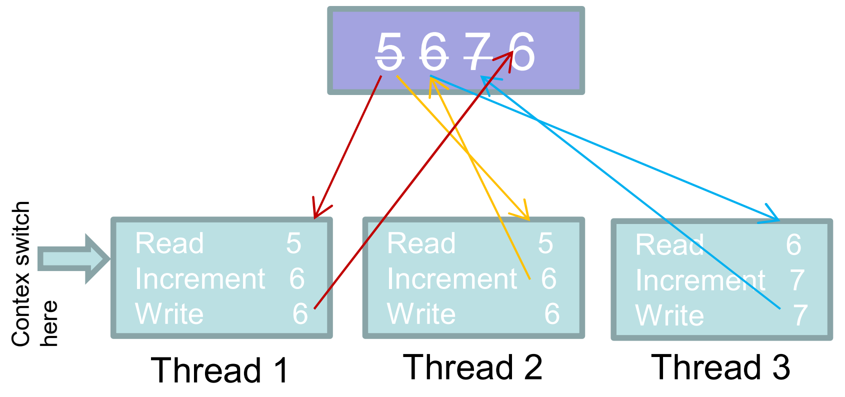
The total value should be 2000.

Synchronization - why?

- When we run the previous program, the result varies every time, like 998, 995, 1002, 1001, etc.
- Why? (It should have been 2000 every time)
  - We have multiple threads running in parallel that are reading and writing same data
    - The execution is interleaved in a non-deterministic way
  - Any operations in progress are visible to other threads - data might be inconsistent
  - Overwriting data updated by another thread

Example of the problem

- Let’s examine more carefully the synchronization problem
  - Three threads increment the same counter
  - In the beginning the counter value is 5
- The following chain of events is possible:
  - Thread 1 reads the counter (5)
  - Context switch happens
  - Thread 2 increments (reads, increments and writes back) the counter (6)
  - Thread 3 increments the counter (7)
  - Turn to run is returned back to the thread1
    - It increments the value 5, it has read already earlier and writes back the value 6



- This means that we loose the effect of the additions in threads 2 and 3 !!
- We need mutex to guarantee that the three steps needed in the additions are done atomically:

```
@Synchronized fun inc(){
        this.count++;
}
```

Synchronized methods

- Mark those methods of a class that may not be executed concurrently with the annotation `@Synchronized`
- Runtime system guarantees (using for example locks) that at most one instance of synchronized methods is executed in parallel
- No conflicts in accessing data
- When a method is annotated with the `@Synchronized` keyword, it can't be run in parallel
  - So don't put the keyword `@Synchronized` unnecessarily to the methods

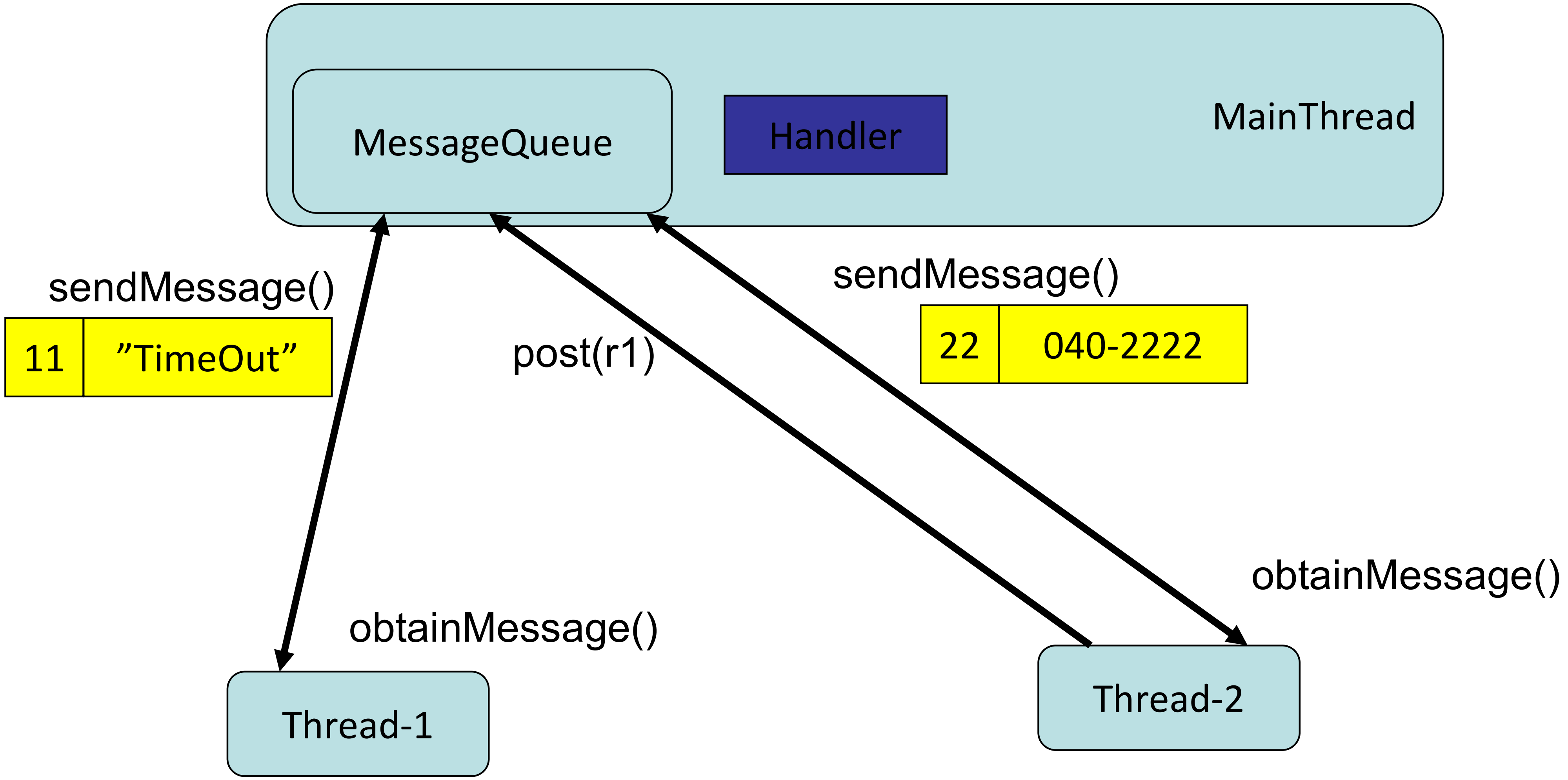
```
class MainActivity : AppCompatActivity() {
    class Account {
        private var amount: Double = 0.0

        @Synchronized fun deposit(amount: Double) {
            val x = this.amount
            Thread.sleep(1)    // simulates processing time
            this.amount = x + amount
        }

        fun saldo(): Double {
            return amount
        }
    }
```

Android Handler

- **to schedule messages and runnables** to be executed at some point in the future
- **to enqueue an action** to be performed on a different thread than your own
- Scheduling messages is accomplished with the **post() and sendMessage()** methods
  - The post methods allow you to enqueue Runnable objects to be called by the message queue when they are received
  - the sendMessage methods allow you to enqueue a Message object containing a bundle of data that will be processed by the Handler's handleMessage(Message) method

When a process is created for your application, its main thread is dedicated to running a message queue that takes care of managing the top-level application objects (activities, broadcast receivers, etc.) and any windows they create.
You can create your own threads, and communicate back with the main application thread through a Handler. This is done by calling Handler's post or sendMessage methods from the new thread. The given Runnable or Message will then be scheduled in the Handler's message queue and processed when appropriate.

MessageQueue



Handler’s MessageQueue

- A secondary thread that wants to communicate with the main thread must request a message token using the **obtainMessage()** method
- Once obtained, the secondary thread can fill data into the message token and attach it to the Handler’s message queue using the **sendMessage()** method
- The Handler uses the **handleMessage()** method to continuously attend new messages arriving to the main thread. A message extracted from the process’ queue can either return some data to the main process or request the execution of runnable objects through the post() method

Message Object

- Defines a message containing a **description (what)** and **arbitrary data object (obj)** that can be sent to a Handler.
  - what - User-defined message code so that the recipient can identify what this message is about
  - obj - An arbitrary object to send to the recipient, for example a string
- While the constructor of Message is public, the best way to get one of these is to call Message.obtain() or call one of the Handler.obtainMessage() methods, which will pull them from a pool of recycled objects

Example

- We have a backend service, which expects two parameters (fname and lname) from http post request
  - Our service will return processed values from those parameters
- Once result is obtained from web server it is displayed in a TextView

Main thread and the worker

```
//MainActivity
private val mHandler: Handler = object :
Handler(Looper.getMainLooper()) {
    override fun handleMessage(inputMessage: Message) {
        if (inputMessage.what == 0) {
            txt_network.text = inputMessage.obj.toString()
        }
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    if (isNetworkAvailable()) {
        val myRunnable = Conn(
            mHandler,
            "John",
            "Doe"
        )
        val myThread = Thread(myRunnable)
        myThread.start()
       }
}

private fun isNetworkAvailable(): Boolean {
    …
}
```

```
class Conn(
    mHand: Handler,
    val fnmae: String,
    val lname: String) : Runnable {

    private val myHandler = mHand

    override fun run() {
        try {
            val myUrl = URL("https://a.com/post.php")
            val myConn = myUrl.openConnection()
                as HttpURLConnection
            myConn.requestMethod = "POST"
            myConn.doOutput = true
            val ostream = myConn.getOutputStream()
            ostream.bufferedWriter().use{
               it.write("fn=${fnmae}&ln=${lname}")
            }

            val istream: InputStream =
                myConn.getInputStream()
            val allText = istream.bufferedReader().use {
                it.readText()
            }
            val result = StringBuilder()
            result.append(allText)
            val str = result.toString()

            val msg = myHandler.obtainMessage()
            msg.what = 0
            msg.obj = str
            myHandler.sendMessage(msg)

        } catch (e: Exception) {
            //...
        }
    }
}
```

Main thread and the worker

```
private val mHandler: Handler = object :
Handler(Looper.getMainLooper()) {
    override fun handleMessage(inputMessage: Message) {
        if (inputMessage.what == 0) {
            txt_network.text = inputMessage.obj.toString()
        }
    }
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    if (isNetworkAvailable()) {
        val myRunnable = Conn(
            mHandler,
            "John",
            "Doe"
        )
        val myThread = Thread(myRunnable)
        myThread.start()
       }
 }

private fun isNetworkAvailable(): Boolean {
    …
}
```

```
class Conn(
    mHand: Handler,
    val fnmae: String,
    val lname: String) : Runnable {

    private val myHandler = mHand

    override fun run() {
        try {
            val myUrl = URL("https://a.com/post.php")
            val myConn = myUrl.openConnection()
                as HttpURLConnection
            myConn.requestMethod = "POST"
            myConn.doOutput = true
            val ostream = myConn.getOutputStream()
            ostream.bufferedWriter().use{
                it.write("fn=${fnmae}&ln=${lname}")
            }

            val istream: InputStream =
                myConn.getInputStream()
            val allText = istream.bufferedReader().use {
                it.readText()
            }
            val result = StringBuilder()
            result.append(allText)
            val str = result.toString()

            val msg = myHandler.obtainMessage()
            msg.what = 0
            msg.obj = str
            myHandler.sendMessage(msg)

        } catch (e: Exception) {
            //…
        }
    }
}
```

Reading list

- Java/Kotlin processes and threads
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html
  - Multithreading and Kotlin. I've been wanting to follow up on my… | by Korhan Bircan | Medium
- Networking
  - http://developer.android.com/training/basics/network-ops/connecting.html
  - (http://developer.android.com/training/basics/network-ops/managing.html)