

Асинхронная работа

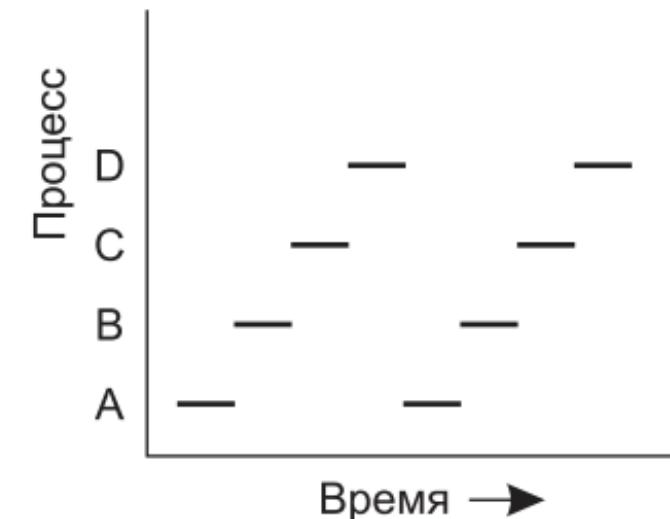
Василевский Елисей

Процессы

«Процесс — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных.»

Современные операционные системы - Э. Таненбаум

В любой многозадачной системе центральный процессор быстро переключается между процессами, предоставляя каждому из них десятки или сотни миллисекунд. При этом хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы.



Создание процесса

События, приводящие к созданию процессов.

1. Инициализация системы.
2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.
3. Запрос пользователя на создание нового процесса

В UNIX существует только один системный вызов для создания нового процесса — fork. Этот вызов создает точную копию вызывающего процесса. После создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами.

В UNIX первоначальное состояние адресного пространства дочернего процесса является копией адресного пространства родительского процесса, но это абсолютно разные адресные пространства — у них нет общей памяти, доступной для записи данных.

Завершение процесса

- Обычный выход (добровольно);
- Выход при возникновении ошибки (добровольно);
- Возникновение фатальной ошибки (принудительно);
- Уничтожение другим процессом (принудительно);

Состояние процесса

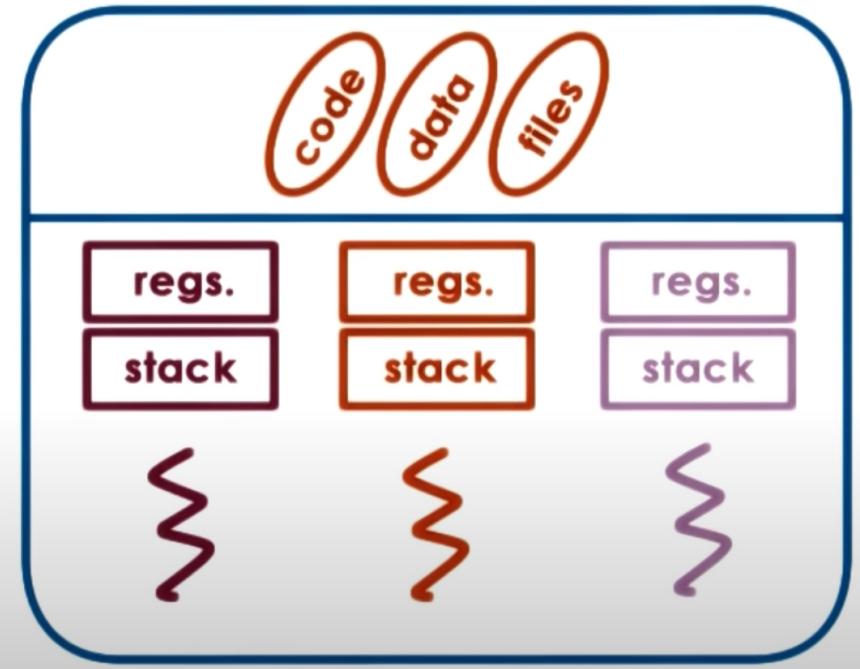
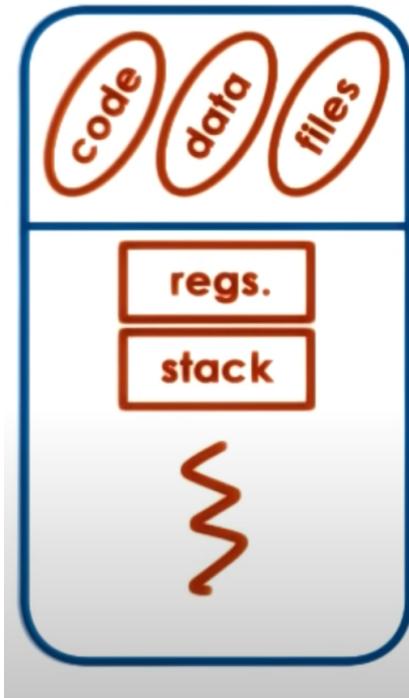


1. Процесс заблокирован в ожидании ввода
2. Диспетчер выбрал другой процесс
3. Диспетчер выбрал данный процесс
4. Входные данные стали доступны

- выполняемый (в данный момент использующий центральный процессор);
- готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);
- заблокированный (неспособный выполнятся, пока не возникнет какое-нибудь внешнее событие).

Потоки

1. Единое адресное пространство и все имеющиеся данные
2. Создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов.



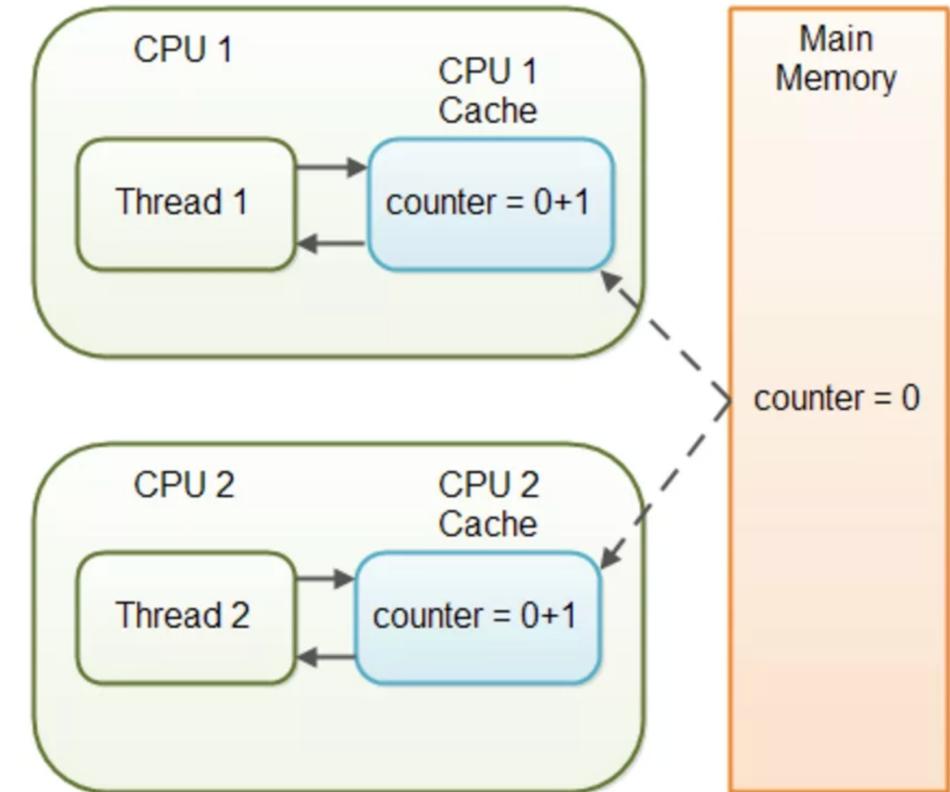
Потоки, visibility

```
fun foo() {  
    var a = 0  
    var b = 0  
    val thread1 = thread {  
        a = 1  
        print(b)  
    }  
    val thread2 = thread {  
        b = 1  
        print(a)  
    }  
    thread1.join()  
    thread2.join()  
}
```

Условия, при которых один поток видит изменения, сделанные другим потоком

Процессоры и кэш

Каждый поток меняет переменную у себя в кэше



Потоки, reordering и optimizations

```
fun foo1() {
    var isDone = false
    var b = 0
    thread {
        b += 2
        isDone = true
    }
    thread {
        while (true) {
            if (isDone) {
                println(b)
                break
            }
        }
    }
}
```

```
fun foo1() {
    var isDone = false
    var b = 0
    thread {
        isDone = true
        b += 2
    }
    thread {
        while (true) {
            if (isDone) {
                println(b)
                break
            }
        }
    }
}
```

```
| fun foo1() {
|     var isDone = false
|     var b = 0
|     thread {
|         isDone = true
|         b += 2
|     }
|     thread {
|         while (true) {
|             if (false) {
|                 println(b)
|                 break
|             }
|         }
|     }
| }
```

Потоки, атомарность

```
fun foo1() {  
    var isDone = false  
    var b = 0  
    thread {  
        b += 2      3 операции  
        isDone = true  
    }  
    thread {  
        while (true) {  
            if (isDone) {  
                println(b)  
                break  
            }  
        }  
    }  
}
```

операция, которая выполняется полностью или не выполняется совсем, частичное выполнение невозможно.

Атомарные операции

Чтение/ запись ссылок и примитивных типов (за исключением long и double)

Чтение/ запись volatile переменных

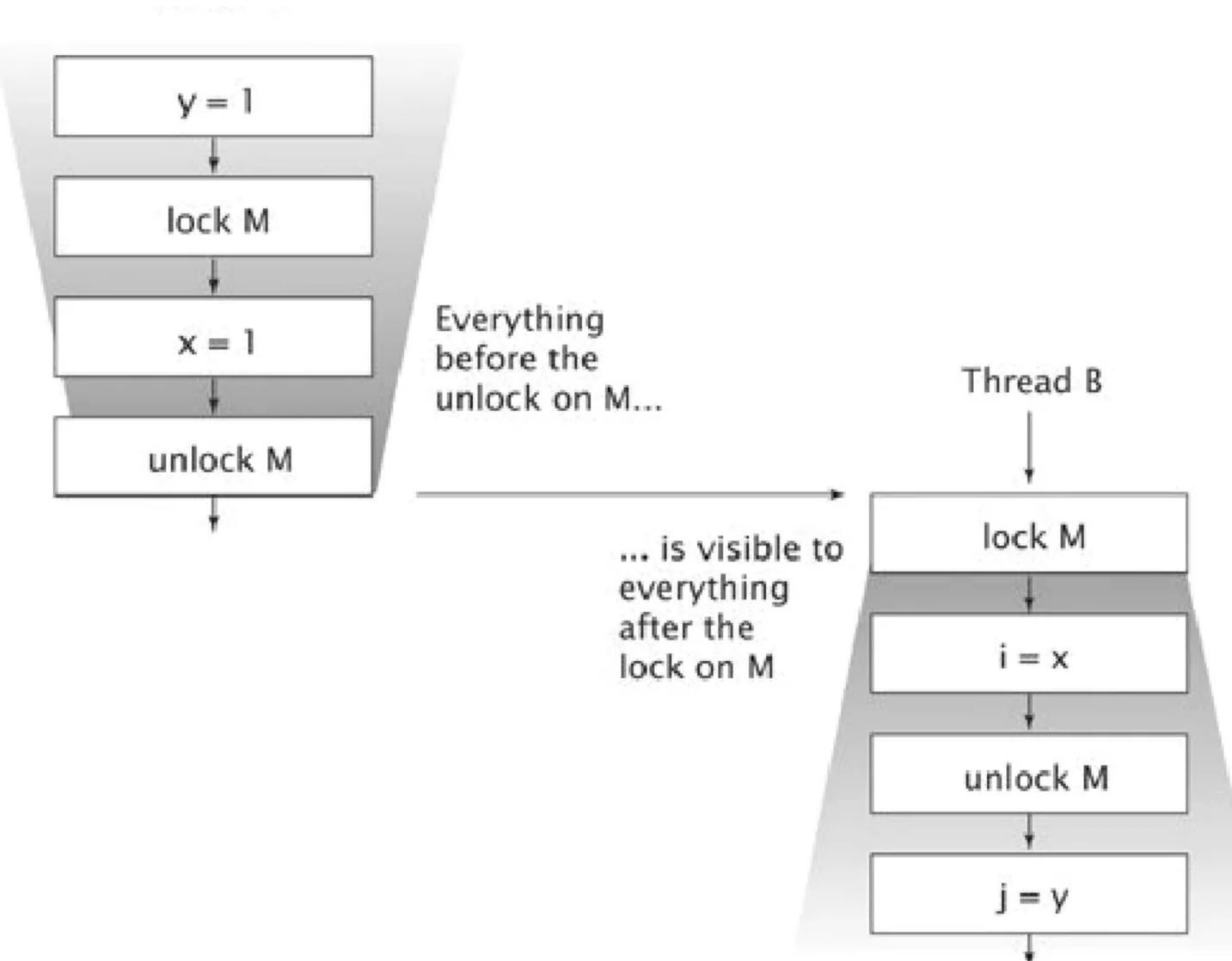
Атомики

Happens-before

X happens-before Y => все операции записи до точки X, видны в любой операции после чтения Y

- **Правило мониторного замка.** Операция unlock на мониторном замке происходит перед каждой последующей операцией lock на том же самом мониторном замке .
- **Правило волатильной переменной.** Запись в волатильное поле происходит перед каждым последующим чтением из этого же поля .
- **Правило запуска потока.** Вызов Thread.start на потоки происходит перед каждым действием в запущенном потоке.
- **Правило терминации потока.** Любое действие в потоке происходит перед тем, как любой другой поток обнаружит, что поток терминировался, успешно вернувшись из Thread.join либо вернув false с помощью Thread.isAlive.
- **Правило прерывания.** Ситуация, когда поток вызывает прерывание на другом потоке, происходит перед тем, как прерванный поток обнаружит прерывание (в результате выдачи исключения InterruptedException либо активации isInterrupted или interrupted).
- **Транзитивность.** Если A происходит перед B и B происходит перед C, то A происходит перед C.

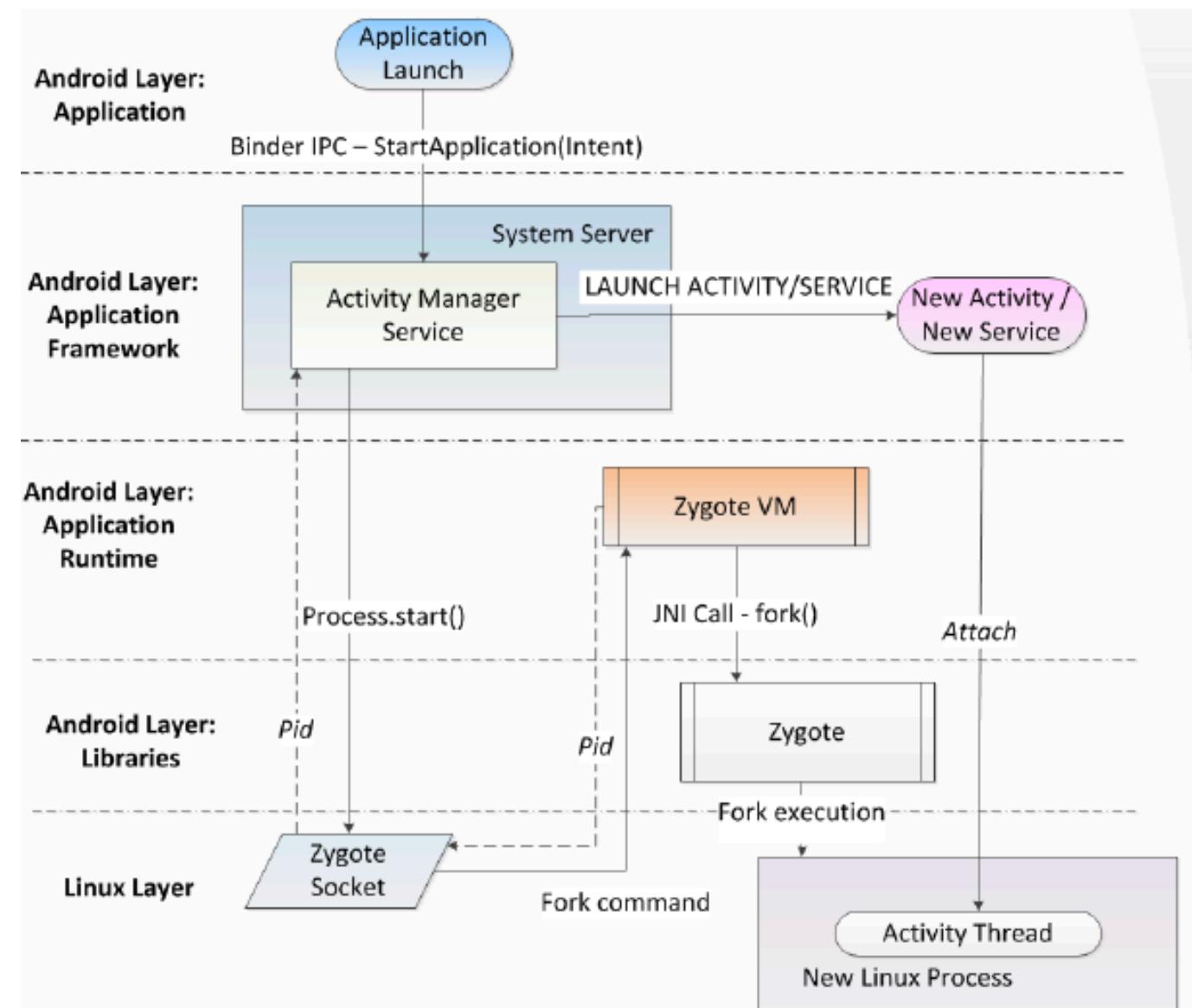
Happens-before



Процессы в Android

Каждое приложение в своем процессе, свое адресное пространство

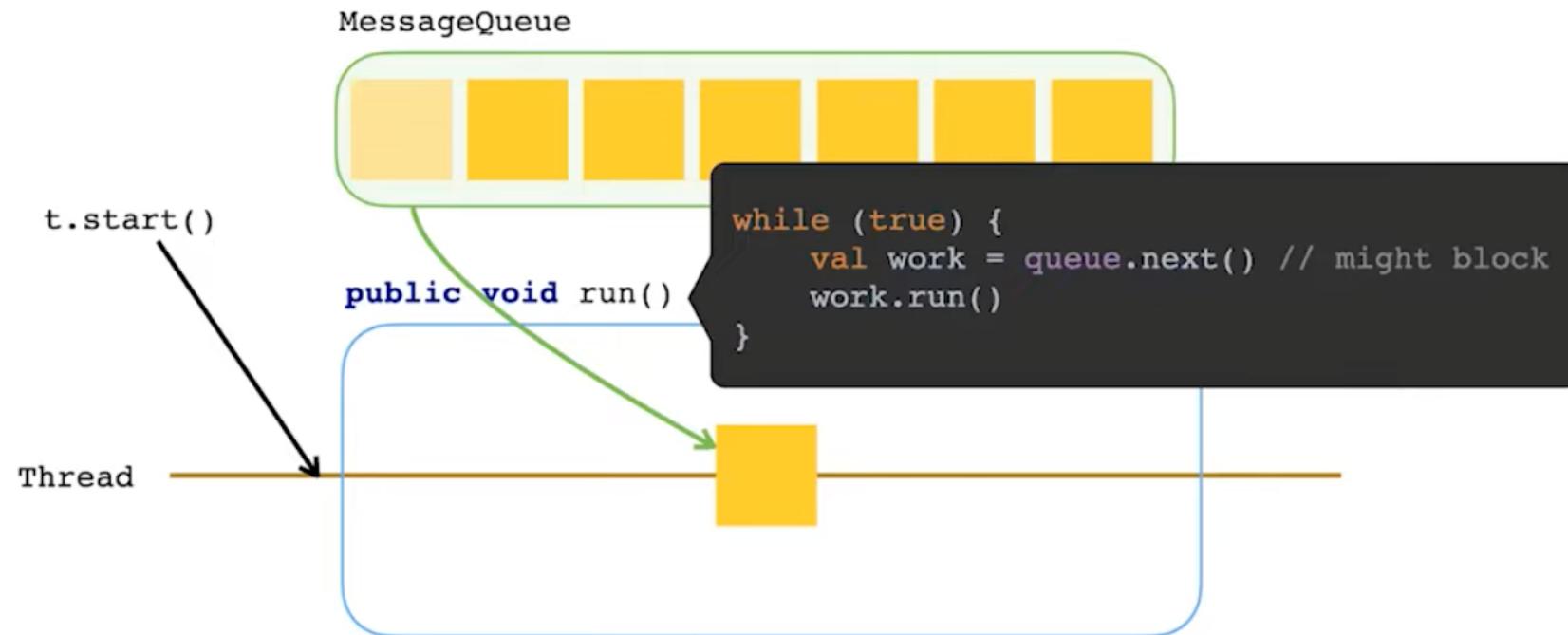
Если есть процесс приложения то открывается он, если нет то обращается к ядру , то там есть корневой процесс zugote который дублируется



Процессы в Android

1. Каждое приложение в своем процессе
2. Одно приложение может создать несколько процессов
Например: браузер
3. Процесс может быть убит/пересоздан в любое время

ГлавНЫЙ ПОТОК



Главный поток

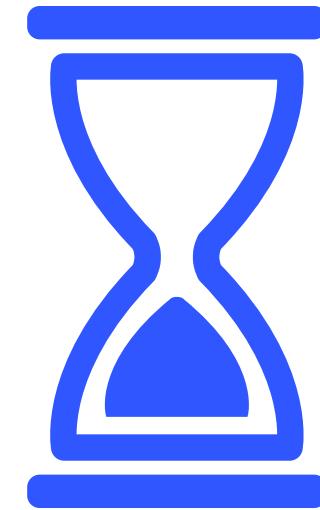
Весь UI приложения работает в рамках главного потока. Все вызовы к UI должны перенаправляться в этот поток.

Все действия в контексте главного потока должны выполняться в пределах около 16 мс (для обеспечения частоты обновления интерфейса в пределах 60 кадров в секунду).

В случае выполнения тяжелой работы на главном потоке, могут возникать артефакты, задержки при обработке пользовательского ввода и т.д. Android считает главный поток заблокированным, если обработка события занимает несколько секунд (около 5), в это случае генерируется исключение (ANR).

Что нельзя делать в главном потоке

- Операции с файлами
- Сетевые запросы
- Операции с БД
- Длительные операции
- Явно выполнять вызовы ожидания (Thread.sleep)



MessageQueue

MessageQueue - низкоуровневый интерфейс, реализующий очередь сообщений в Android предлагает.

MessageQueue работает с экземплярами класса Message, внутри которых можно положить данные любого типа или Runnable.

Android автоматически создаёт пул классов Message, поэтому рекомендуется использовать статический метод Message#obtain, вместо вызова конструктора.

```
val message = Message.obtain()  
message.obj = SomeObject()
```

Looper

Looper - бесконечный цикл выборки сообщений, выполняющийся в потоке к которому он добавлен, отвечает за доставку сообщений из MessageQueue.

- Looper.prepare() - создаёт цикл выборки сообщений и добавляет его к потоку, в котором этот метод вызывается
- Looper.loop() - блокирующий метод, стартует цикл выборки сообщений
- Looper.quit() - завершает выполнение метода Looper.loop()

Looper

- По умолчанию, поток не имеет цикла выборки сообщений
- К потоку можно добавить не более одного цикла выборки сообщений
- Looper.getThread() -вернёт поток, к которому Looper был добавлен или null
- Проверка является ли текущий поток главным Looper.getMainLooper() == Looper.myLooper()

Handler

Класс Handler предназначен для отправки Message и для обработки в потоке, к которому добавлен Looper

Конструктор принимает на вход экземпляр Looper или неявно вызывает внутри себя Looper.myLooper().

```
val handler: Handler = object : Handler(looper) {
    // Этот метод будет выполнен в потоке, к которому добавлен looper
    override fun handleMessage(msg: Message) {
        // handle incoming message
    }
}
```

Отправка сообщения

```
handler.obtainMessage(what: 0, SomeObject()).sendToTarget()
```

HandlerThread

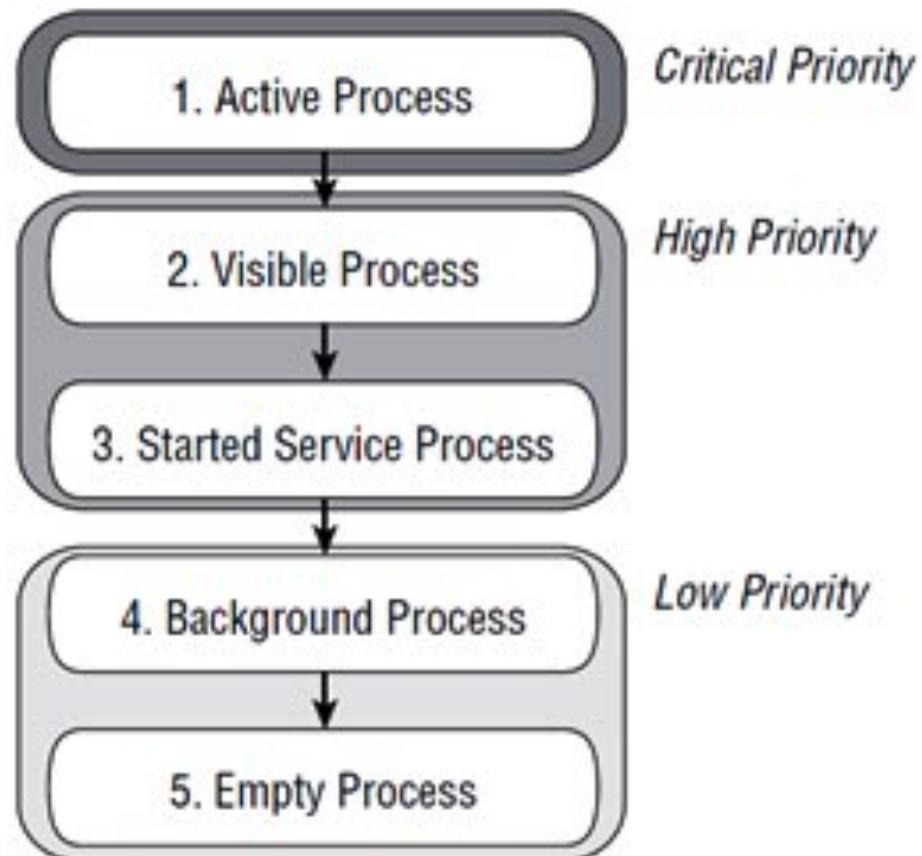
```
public class HandlerThread extends Thread {  
    Looper mLooper;  
    private Handler mHandler;  
  
    public HandlerThread(String name) {  
        super(name);  
    }  
  
    public HandlerThread(String name, int priority) {  
        super(name);  
    }  
  
    @Override  
    public void run() {  
        Looper.prepare();  
        synchronized (this) {  
            mLooper = Looper.myLooper();  
            notifyAll();  
        }  
        Looper.loop();  
    }  
}
```

Service

Один из основных компонентов Android приложения, предназначен для выполнения операций в фоне. Например: проигрывание музыки, синхронизация данных приложения с сервером, длительные сетевые запросы и т.д. Сервис непосредственно не связан с пользовательским интерфейсом приложения и его жизненным циклом.

Как и у Activity жизненный цикл сервиса 'привязан' к главному потоку (onStartCommand). Для выполнения своих задач, сервис должен использовать фоновые потоки.

Процессы



Чем ниже приоритет тем выше вероятность,
что система убьет процесс

Service

Объявление в манифесте

```
<service  
    android:name=".MyService" />
```

Запуск

```
val intent = Intent(context, MyService::class.java)  
context.startService(intent);
```

Некоторые методы

```
fun onCreate()
```

```
fun onStartCommand(intent: Intent)
```

```
fun onDestroy()
```

Service

Остановка

- `stopService()`
- в `startService()` передать специальный Intent, по которому сервис вызовет `stopSelf()`

Виды сервисов

- **Foreground** - в случае длительных действий, нужно уведомлять пользователя с помощью нотификаций
- **Background** - если делать фоновую работу и не уведомлять пользователя, то работа будет быстро завершена

Worker

Гарантированное выполнение фоновой задачи

- Worker
- WorkRequest
- WorkManager

Был добавлен недавно, но доступен с 14 апи (под капотом разные технологии на разных версиях)

Worker

```
class MyWorker(context: Context, workerParams: WorkerParameters) : Worker(context, workerParams) {  
    override fun doWork(): Result {  
        //do work  
        return Result.success()  
    }  
  
    val someWork = OneTimeWorkRequest.Builder(MyWorker::class.java).build()  
    WorkManager.getInstance(baseContext).enqueue(someWork);
```

Корутины

"concurrency design pattern that you can use on Android to simplify code that executes asynchronously"

- **Облегченный**
- **Меньше утечек памяти**
- **Встроенная поддержка отмены**

Корутины

Suspend функции могут быть вызваны только из корутин или suspend функций

```
suspend fun doSmth(){  
}
```

Coroutine Builders

```
val scope: CoroutineScope = ...
```

```
scope.launch{...}
```

```
scope.async{...}
```

```
runBlocking{...}
```

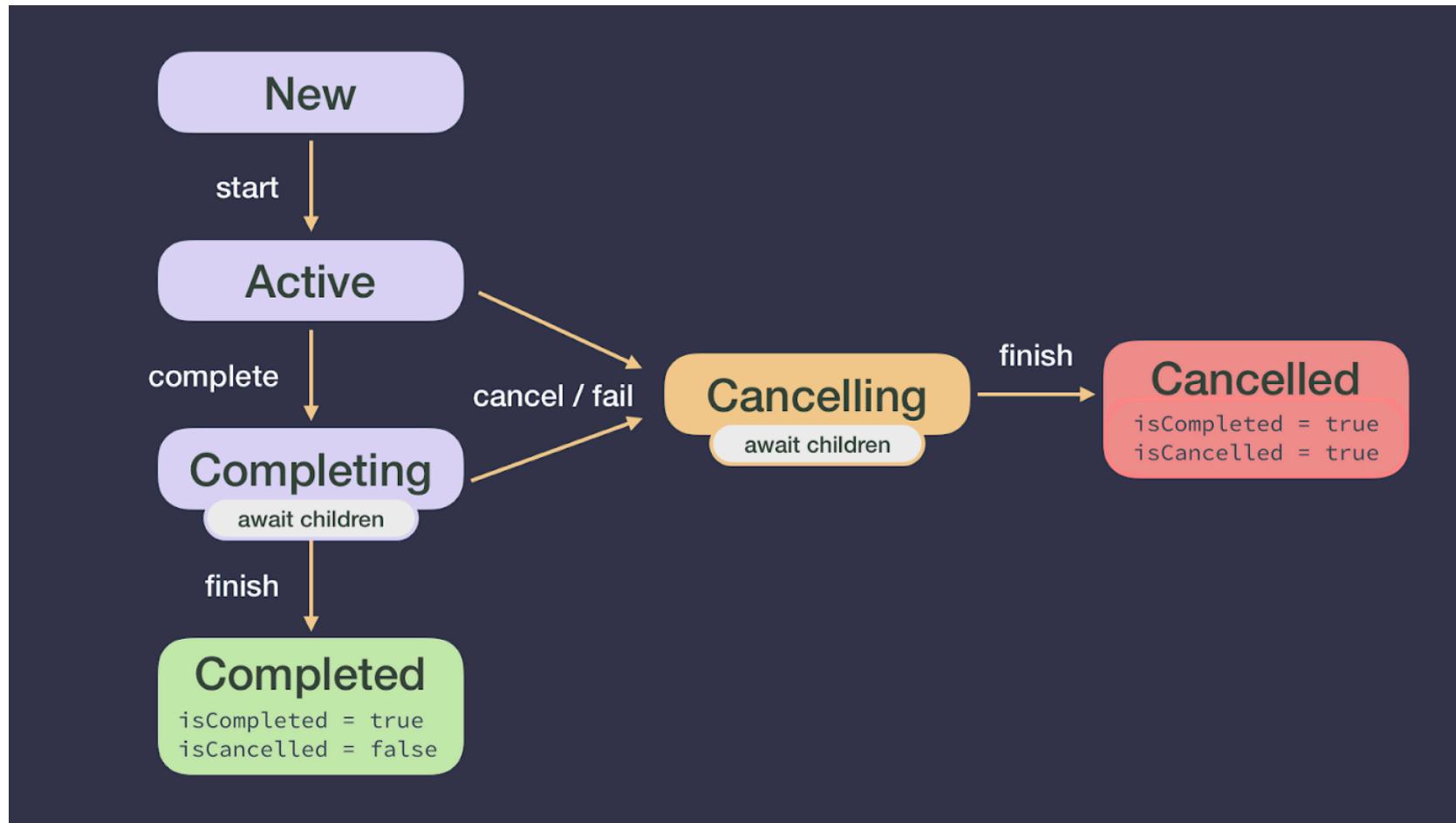
Корутины

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:x.x.x' -
нужно именно для андроида , т.к. для разных фреймворков
используются
разные реализации диспетчеров

Dispatchers

Dispatchers.Main	MessageQueue, Handler
Dispatchers.Default	Пулл потоков
Dispatchers.IO	Расширяемый Пулл потоков
Dispatchers.Unconfined	Без разницы где выполнять
.asCoroutineDispatcher()	Можем написать свой с помощью экстеншен функции

Корутины



Жизненный цикл Job

Корутины

```
public interface CoroutineExceptionHandler : CoroutineContext.Element {  
  
    Handles uncaught exception in the given context. It is invoked if coroutine has an uncaught  
    exception.  
  
    public fun handleException(context: CoroutineContext, exception: Throwable)  
}
```

```
val handler1 = CoroutineExceptionHandler {...}  
val handler2 = CoroutineExceptionHandler {...}  
val handler3 = CoroutineExceptionHandler {...}  
scope.launch(handler1) { this: CoroutineScope  
    launch(handler2) { this: CoroutineScope  
        launch(handler3) { this: CoroutineScope  
            Выброшено исключение  
        }  
    }  
}  
}
```

Сработает только самый верхнеуровневый хендлер,
т.к. ошибка прокидывается вверх к родителю, и он
уже отменяет дочерние корутины

Если хендлер не указан у родителя то исключение
будет выброшено дальше

Можно указать хендлер у всего скоупа

Корутины

```
scope.async { this: CoroutineScope
    launch(handler2) { this: CoroutineScope
        }
    launch(handler2) { this: CoroutineScope
        launch(handler3) { this: CoroutineScope
            }
    }
}
```

В случае `async` ошибка не произойдет, даже если не указать хендлер

Хендлер не будет вызван, даже если указать у скопа

Но будет выброшено исключение при вызове `await`

Спасибо за внимание!

