



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение  
высшего образования

«МИРЭА – Российский технологический университет»

**РТУ МИРЭА**

---

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных  
технологий

## **Отчет по практической работе №8**

по дисциплине «Разработка мобильных приложений»

**Выполнил:**

Студент группы ИКБО-20-23

Комисарик М.А.

**Проверил:**

Старший преподаватель кафедры  
МОСИТ

Шешуков Л.С.

Москва 2025 г.

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ .....	3
1.1 Введение.....	3
1.2 Thread, Runnable .....	4
1.3 Handler .....	9
1.4 WorkManager.....	23
2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ .....	44
2.1 Разметка.....	44
2.2 Реализация.....	47
2.2.1 Последовательные и параллельные потоки.....	47
2.2.2 Работа с Rest API.....	49
2.3 Тестирование .....	51
ЗАКЛЮЧЕНИЕ .....	53

# 1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

## 1.1 Введение

С появлением многоядерных процессоров возникло понятие параллелизма — одновременного выполнения нескольких вычислений. Ранее все задачи процессор выполнял строго последовательно.

Безусловно, даже в эпоху одноядерных процессоров пользователи могли одновременно просматривать сайты в браузере, слушать музыку и копировать файлы. Однако эта "одновременность" была иллюзорной — на самом деле единственный процессор быстро переключался между задачами. Для создания эффекта параллельной работы процессы (программы) разделялись на потоки. Процессор поочерёдно выполнял небольшие фрагменты разных потоков, создавая видимость параллельной работы. Такой подход называется псевдопараллелизмом.

Поток (Thread) можно представить как последовательность команд программы, которая претендует на использование процессора вычислительной системы для своего выполнения. Потоки одной и той же программы работают в общем адресном пространстве и, тем самым, разделяют (совместно используют) данные программы.

Разработка многопоточных программ — непростая задача. При планировании многопоточной программы оптимальным вариантом является разделение последовательности вычислений на потоки, число которых равно количеству процессоров. Если сделать много потоков, то это не ускорит выполнение программы, а затормозит ее, потому что переключение между потоками требует некоторого дополнительного времени.

В Android-приложениях на Java доступно несколько инструментов для работы с потоками, которые мы рассмотрим далее.

## 1.2 Thread, Runnable

Поток, с которого начинается выполнение программы, называется главным (основным). В Java выполнение главного потока начинается с метода `main()` после создания процесса. Затем, в соответствии с логикой программы, при выполнении определённых условий могут запускаться дополнительные (фоновые) потоки.

В Java предусмотрены два универсальных способа создания потоков (Threads), которые работают как в обычных Java-приложениях, так и в Android:

- наследование от класса `Thread`: создаётся класс-наследник от `Thread` с переопределённым методом `run()`, содержащим код для выполнения в отдельном потоке,
- реализация интерфейса `Runnable`: создаётся объект класса, реализующего интерфейс `Runnable`, т.е. переопределяющего его единственный метод `run()`, который затем передаётся в конструктор `Thread`.

Класс `Thread` предоставляет набор методов для управления потоками:

- `getName()` — возвращает имя потока,
- `getPriority()` — возвращает приоритет потока (значение от 1 до 10),
- `isAlive()` — проверяет, активен ли поток (возвращает `true`, если поток запущен и ещё не завершён),
- `join()` — приостанавливает выполнение текущего потока до завершения потока, метод `join()` которого был вызван,
- `run()` — содержит код, который будет выполняться в потоке,
- `sleep(long millis)` — приостанавливает выполнение потока на указанное количество миллисекунд,
- `start()` — запускает новый поток, вызывая его метод `run()`.

Реализуем простое Android-приложение, демонстрирующее возникновение ошибки ANR (Application Not Responding) при блокировке основного потока. В классе `MainActivity` напомним обработку двух кнопок:

первая будет вызывать метод с длительной операцией, вторая — выводить сообщение в лог.

Для начала создадим разметку `activity_main.xml`, где разместим две кнопки с различными идентификаторами. Затем в методе `onCreate()` `MainActivity` инициализируем эти кнопки и установим для них обработчики нажатий. Для первой кнопки реализуем вызов метода `doSlow()`, а для второй — простой вывод в лог.

В методе `doSlow()` специально напишем код, который приостанавливает выполнение потока на 20 секунд с помощью `Thread.sleep()`. Такой подход искусственно создаёт ситуацию, когда основной поток приложения блокируется на длительное время (Рисунок 1).

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button btn1 = findViewById(R.id.button);
        Button btn2 = findViewById(R.id.button2);

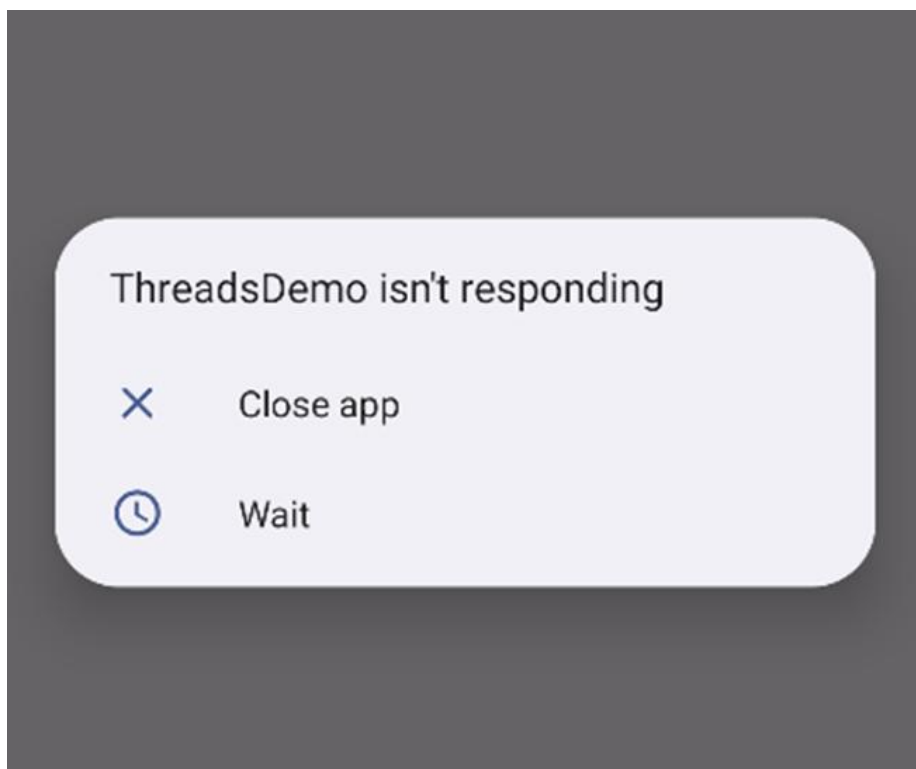
        btn1.setOnClickListener(v -> doSlow());
        btn2.setOnClickListener(v -> Log.d( tag: "DDD", msg: "Нажали на вторую кнопку!"));
    }

    public void doSlow() {
        try {
            // Приостанавливаем поток на 20 секунд
            Thread.sleep( millis: 20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

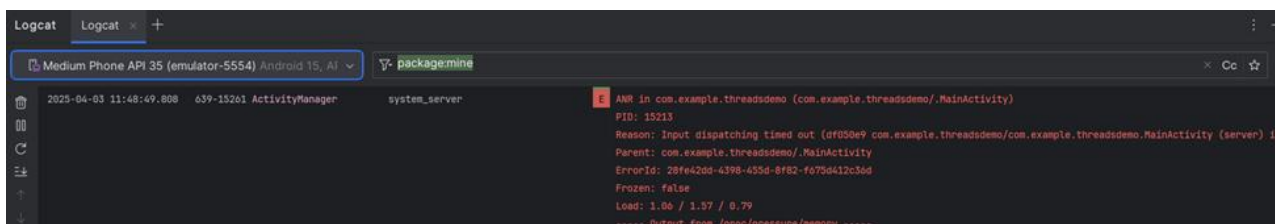
**Рисунок 1 – Содержание проблематичного приложения**

При запуске приложения и нажатии на первую кнопку наблюдаем, что интерфейс перестаёт реагировать на любые действия пользователя. Через 5 секунд система Android выводит сообщение о том, что приложение не отвечает, предлагая закрыть приложение или подождать. При этом попытка нажать вторую кнопку во время "зависания" не даёт результата — сообщение в лог не

выводится, так как основной поток занят выполнением длительной операции (Рисунки 2-3).



**Рисунок 2 – Демонстрация ANR ошибки**



**Рисунок 3 – Сообщение об ANR ошибке в Logcat**

Данный пример наглядно демонстрирует важное правило разработки Android-приложений: все длительные операции необходимо выносить в фоновые потоки, чтобы избежать блокировки основного потока и предотвратить появление ошибок ANR.

ANR возникает в вашем приложении, когда выполняется одно из следующих условий:

- тайм-аут обработки ввода: ваше приложение не ответило на входное событие (например, нажатие клавиши или касание экрана) в течение 5 секунд,

- выполняющийся сервис: если сервис, объявленный вашим приложением, не завершил выполнение `Service.onCreate()`, `Service.onStartCommand()` или `Service.onBind()` в течение нескольких секунд,
- `service.startForeground()` не вызван: если ваше приложение использует `Context.startForegroundService()` для запуска нового фонового сервиса, но этот сервис затем не вызывает `startForeground()` в течение 5 секунд,
- передача интента: если `BroadcastReceiver` не завершил выполнение в установленное время. Если в приложении есть `Activity` на переднем плане, этот тайм-аут составляет 5 секунд,
- взаимодействие с `JobScheduler`: если `JobService` не возвращает управление из `JobService.onStartJob()` или `JobService.onStopJob()` в течение нескольких секунд, либо если пользовательский `job` запускается, а ваше приложение не вызывает `JobService.setNotification()` в течение нескольких секунд после вызова `JobService.onStartJob()`. Для приложений, ориентированных на Android 13 и ниже, такие ANR являются "тихими" и не сообщаются приложению. Для приложений, ориентированных на Android 14 и выше, ANR становятся явными и сообщаются приложению.

Как избежать ошибок ANR:

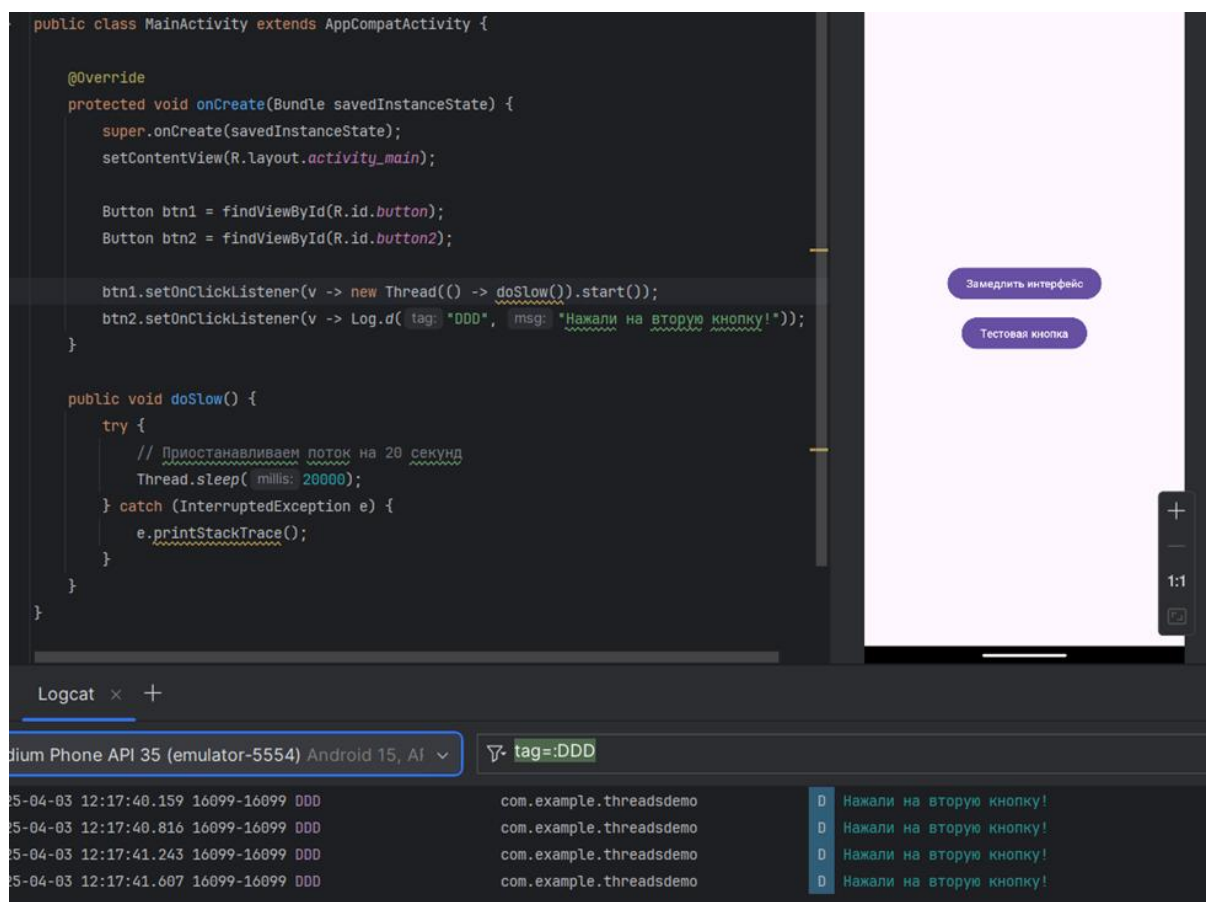
- главный поток пользовательского интерфейса (UI Thread) должен выполнять только операции, связанные с обновлением интерфейса,
- все ресурсоемкие операции (работу с базой данных, операции ввода-вывода, сетевые запросы) необходимо выполнять в отдельных потоках (Thread),
- для взаимодействия между UI-поток и фоновыми потоками используйте `Handler`,
- для обработки асинхронных операций применяйте современные решения: `RxJava`, `Kotlin Coroutines` или `WorkManager`.

Для решения проблемы ANR в нашем приложении реализуем вызов длительной операции в отдельном потоке. При нажатии на кнопку создаём

новый поток, используя класс Thread. В конструктор Thread передаём реализацию интерфейса Runnable, которую оформляем в виде лаконичного лямбда-выражения. Это лямбда-выражение содержит вызов нашего метода doSlow(), выполняющего длительную операцию.

После создания объекта Thread вызываем его метод start(), который запускает новый поток выполнения. Такой подход позволяет основному потоку приложения продолжать обрабатывать пользовательский интерфейс, в то время как длительная операция выполняется в фоне.

Такой подход гарантирует, что интерфейс приложения остаётся отзывчивым, а система Android не будет считать приложение зависшим, даже при выполнении длительных операций. Вторую кнопку можно нажимать в любой момент, и её обработчик будет немедленно срабатывать, так как основной поток больше не блокируется операцией sleep() (Рисунок 4).



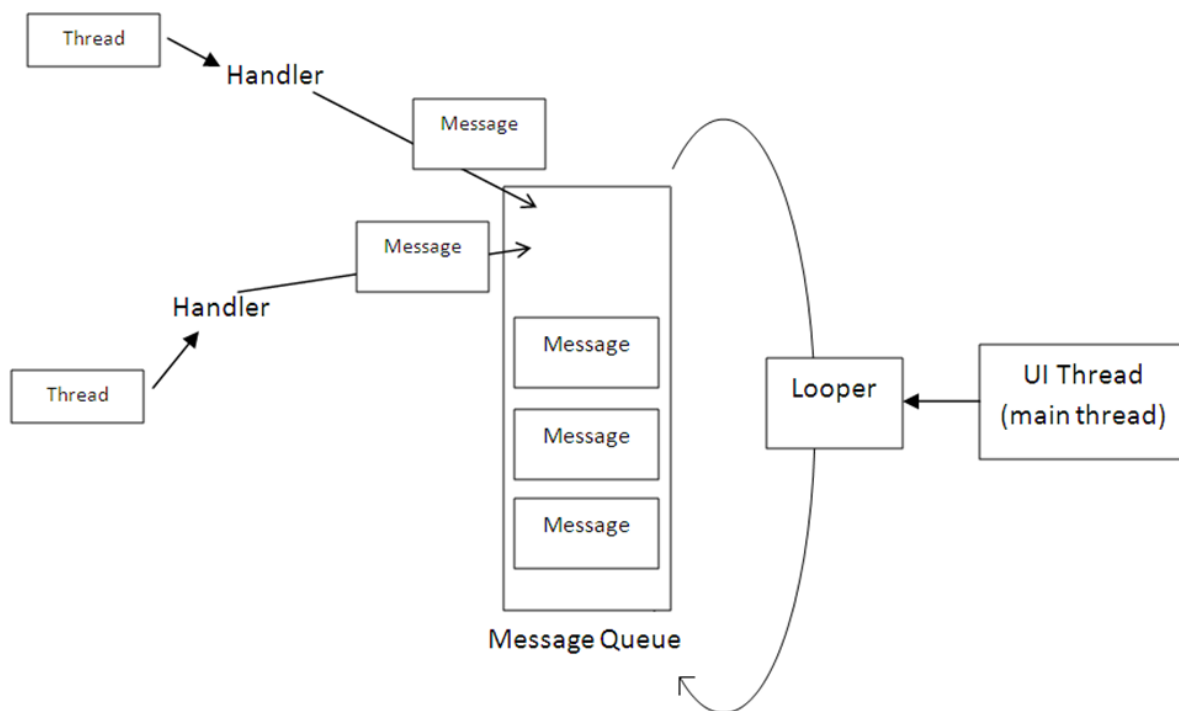
**Рисунок 4 – Предотвращение ошибки ANR посредством запуска длительной операции в отдельном потоке**



## 1.3 Handler

Класс `android.os.Handler` представляет собой важный механизм для работы с многопоточностью в Android, который значительно упрощает взаимодействие между потоками. Основная идея `Handler` заключается в том, что он позволяет планировать выполнение кода как в текущем, так и в других потоках, обеспечивая безопасную передачу задач. При создании `Handler` автоматически связывается с `Looper`'ом — специальным компонентом, который непрерывно обрабатывает очередь сообщений (`MessageQueue`) в потоке. Это позволяет `Handler`'у получать и выполнять как `Runnable`-задачи, так и объекты `Message`.

Объект `Message` в Android представляет собой контейнер для данных, который может передаваться между потоками. Он может содержать различную информацию: числовые значения, ссылки на объекты или другие данные. Все сообщения попадают в `MessageQueue` — специальную очередь, связанную с конкретным потоком. `Looper` постоянно проверяет эту очередь и передает сообщения соответствующему `Handler`'у для обработки. Именно такая архитектура обеспечивает безопасное взаимодействие между потоками (Рисунок 5).



**Рисунок 5 – Взаимодействие Handler с другими компонентами для обеспечения многопоточности**

Handler находит применение в двух основных сценариях. Во-первых, он позволяет планировать выполнение кода на будущее — например, запускать задачи с определенной задержкой с помощью метода `postDelayed`. Во-вторых, он обеспечивает возможность выполнения кода в другом потоке, что особенно важно при обновлении пользовательского интерфейса из фоновых потоков. Когда фоновый поток завершает сложные вычисления, он может через Handler отправить результат в главный поток для отображения, избегая при этом проблем с блокировкой UI.

Опишем некоторые наиболее используемые методы класса Handler:

- `post(Runnable r)` — добавляет Runnable в очередь сообщений для немедленного выполнения в потоке, связанном с Handler'ом,
- `postDelayed(Runnable r, long delayMillis)` — добавляет Runnable в очередь сообщений для выполнения через указанное количество миллисекунд (задержка отсчитывается с момента вызова метода),

- `sendMessage(Message msg)` — помещает `Message` в очередь сообщений для немедленной обработки в потоке `Handler`'а,
- `sendMessageDelayed(Message msg, long delayMillis)` — помещает `Message` в очередь сообщений для обработки через указанное количество миллисекунд (задержка отсчитывается с момента отправки),
- `removeCallbacks(Runnable r)` — удаляет из очереди все ожидающие выполнения экземпляры указанного `Runnable`.

Реализуем простое Android-приложение, демонстрирующее работу `Handler` для межпоточного взаимодействия. Создадим базовую разметку `Activity` с использованием вертикального `LinearLayout`. Внутри контейнера размещаем два основных элемента: `TextView` для отображения статуса выполнения и `Button` для запуска операции (Рисунок 6).

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp">

    <TextView
        android:id="@+id/result_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Нажмите кнопку для начала загрузки"
        android:textSize="18sp"
        android:layout_marginBottom="24dp"/>

    <Button
        android:id="@+id/start_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Начать загрузку"/>

</LinearLayout>
```

Рисунок 6 – Разметка для `MainActivity`

При реализации многопоточного взаимодействия в Android создаем объект `Handler`, который будет обеспечивать связь между потоками. В конструктор `Handler` передаем объект `Looper`, получаемый через статический метод

getMainLooper(). Этот метод возвращает Looper главного (UI) потока приложения, что позволяет Handler'у отправлять и обрабатывать сообщения именно в контексте основного потока.

Механизм работы основан на том, что каждый поток с Looper'ом имеет собственную очередь сообщений (MessageQueue). Когда мы создаем Handler с Looper.getMainLooper(), он привязывается к очереди сообщений главного потока. Это означает, что все Runnable задачи или Message, отправленные через этот Handler, будут автоматически выполняться в UI-потоке, даже если отправка происходит из фонового потока.

Такой подход обеспечивает безопасное обновление пользовательского интерфейса из любых фоновых потоков. При этом не требуется самостоятельно синхронизировать потоки или проверять, в каком потоке находится код — Handler автоматически доставляет задачи в нужный поток благодаря привязке к его Looper'у (Рисунок 7).

```
public class MainActivity extends AppCompatActivity {
    private TextView resultText;
    private Button startButton;
    private Handler uiHandler;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        resultText = findViewById(R.id.result_text);
        startButton = findViewById(R.id.start_button);

        // Создаем Handler, привязанный к главному потоку
        uiHandler = new Handler(Looper.getMainLooper());
    }
}
```

Рисунок 7 – Создание Handler и привязка его к UI потоку

Реализуем обработку нажатия кнопки, которая инициирует выполнение фоновой задачи с последующим обновлением интерфейса. При нажатии на

кнопку сразу же изменяем текст в `TextView`, уведомляя пользователя о начале загрузки. Затем создаем и запускаем новый поток, в котором выполняем имитацию длительной операции с помощью `Thread.sleep()`.

Внутри фонового потока после завершения операции используем ранее созданный `uiHandler` для безопасного обновления пользовательского интерфейса. Вызываем метод `post()` у `Handler`'а, передавая ему `Runnable` с кодом, который должен выполняться в главном потоке. В данном случае этот код изменяет текст `TextView`, сообщая о завершении загрузки.

Особенность данной реализации заключается в том, что хотя длительная операция выполняется в фоновом потоке, все манипуляции с элементами интерфейса происходят через `Handler`, что гарантирует их выполнение в главном потоке. Такой подход предотвращает возможные ошибки и исключает блокировку пользовательского интерфейса во время выполнения ресурсоемких задач.

При этом важно отметить, что метод `post()` `Handler`'а обеспечивает асинхронное выполнение переданного кода, не блокируя при этом ни фоновый поток, откуда происходит вызов, ни главный поток, в котором выполняется обновление интерфейса (Рисунок 8).

```

startButton.setOnClickListener(v -> {
    resultText.setText("Загрузка началась...");

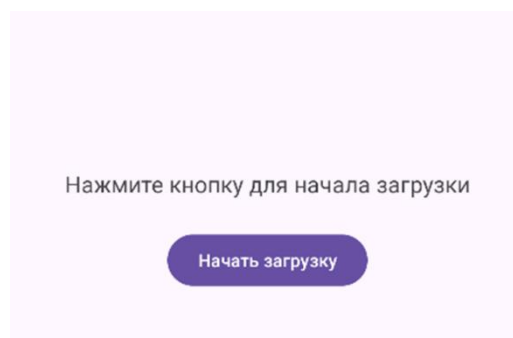
    // Запускаем фоновый поток
    new Thread(() -> {
        // Имитируем длительную операцию (3 секунды)
        try {
            Thread.sleep( millis: 3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Через Handler обновляем UI из фонового потока
        uiHandler.post(() -> {
            resultText.setText("Загрузка завершена!");
        });
    }).start();
});
}
}

```

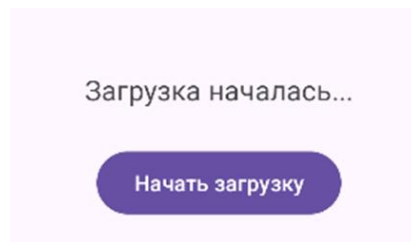
**Рисунок 8 – Обновление UI через Handler**

При запуске приложения на экране отображается начальное состояние с текстом "Нажмите кнопку для начала загрузки" и кнопкой "Начать загрузку" (Рисунок 9).



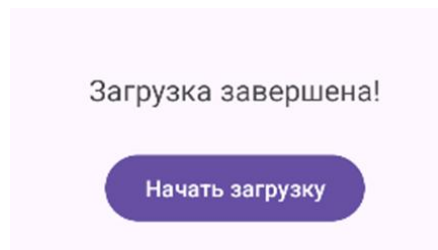
**Рисунок 9 – Начальное состояние приложения**

При нажатии на кнопку текст сразу изменяется на "Загрузка началась...", что подтверждает мгновенную реакцию интерфейса на действие пользователя. В течение следующих 3 секунд, пока выполняется фоновая операция, интерфейс остается полностью отзывчивым — кнопку можно нажимать повторно, хотя в текущей реализации это не приводит к новым действиям (Рисунок 10).



**Рисунок 10 – Интерфейс во время выполнения фоновой операции**

После завершения фоновой задачи текст автоматически обновляется до "Загрузка завершена!", демонстрируя корректную работу механизма Handler для межпоточного взаимодействия (Рисунок 11).



**Рисунок 11 – Обновление UI после выполнения операции**

Приведём пример создания приложения, демонстрирующего использование Handler для передачи сообщений между потоками с помощью объектов типа Message.

Класс Message представляет собой контейнер для передачи данных между потоками через Handler. Основные поля и их назначение:

- what (int) — идентификатор типа сообщения, позволяет определить его назначение (используются заранее объявленные константы),
- arg1 (int) — первое целочисленное значение для передачи простых данных (например, прогресса),
- arg2 (int) — второе целочисленное значение (дополнительный параметр),
- obj (Object) — произвольный объект для передачи данных,
- replyTo (Messenger) — ссылка на объект типа Messenger для получения ответных сообщений,
- sendingUid (int) — UID отправителя.

В данном примере реализуем фоновую задачу, которая периодически отправляет сообщения в главный поток для обновления пользовательского интерфейса.

Создадим разметку Activity, содержащую кнопку для запуска фоновой задачи и текстовые поля для отображения статуса выполнения и счетчика прогресса (Рисунок 12).

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/startButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Start Background Task" />

    <TextView
        android:id="@+id/statusText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Status: Ready" />

    <TextView
        android:id="@+id/counterText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Counter: 0" />

</LinearLayout>
```

**Рисунок 12 – Измененная разметка MainActivity**

В классе MainActivity объявляем необходимые компоненты интерфейса и механизмы для работы с потоками. Создаем поля для текстовых элементов (statusText и counterText), которые будут отображать статус выполнения и текущий прогресс операции. Добавляем кнопку startButton для инициации фоновой задачи.

Определяем набор констант для идентификации типов сообщений:



- MSG\_UPDATE\_STATUS (значение 1) — для обновления текста статуса,
- MSG\_UPDATE\_COUNTER (значение 2) — для передачи текущего значения счетчика,
- MSG\_TASK\_COMPLETED (значение 3) — для уведомления о завершении задачи.

Добавим поле `mainHandler` типа `Handler`, которое будет обеспечивать связь между потоками. Это позволит безопасно обновлять пользовательский интерфейс из фонового потока. Также объявляем поле `backgroundThread` для хранения экземпляра нашего фонового потока (Рисунок 13).

```
public class MainActivity extends AppCompatActivity {  
  
    private TextView statusText;  
    private TextView counterText;  
    private Button startButton;  
  
    private Handler mainHandler;  
    private BackgroundThread backgroundThread;  
  
    // Константы для идентификации сообщений  
    private static final int MSG_UPDATE_STATUS = 1;  
    private static final int MSG_UPDATE_COUNTER = 2;  
    private static final int MSG_TASK_COMPLETED = 3;
```

Рисунок 13 – Поля класса MainActivity

В классе `MainActivity` создаём внутренний класс `BackgroundThread`, который наследуется от `Thread`. Определяем поле `mainHandler` типа `Handler`, которое будет использоваться для отправки сообщений в главный поток. Реализуем конструктор класса, принимающий параметр `mainHandler` и сохраняющий его в соответствующее поле. Такой подход позволяет передать экземпляр `Handler` из основного потока в фоновый, обеспечивая возможность межпоточного взаимодействия через механизм сообщений (Рисунок 14).

```

public class MainActivity extends AppCompatActivity {

    // Внутренний класс для фонового потока
    private static class BackgroundThread extends Thread {
        private final Handler mainHandler;

        public BackgroundThread(Handler mainHandler) {
            this.mainHandler = mainHandler;
        }
    }
}

```

**Рисунок 14 – Внутренний класс BackgroundThread, часть 1**

Реализуем логику фонового потока в методе `run()` класса `BackgroundThread`. Внутри метода организуем обработку длительной операции с использованием механизма сообщений для взаимодействия с главным потоком.

В начале работы потока создаем первое сообщение с помощью статического метода `Message.obtain()`, который возвращает пустой объект `Message` из пула переиспользуемых объектов. Заполняем поле `what` константой `MSG_UPDATE_STATUS`, идентифицирующей тип сообщения, а в поле `obj` помещаем строку `"Initializing..."` для отображения статуса. Отправляем сообщение через метод `sendMessage()` нашего `Handler`'а.

В основном рабочем цикле реализуем пошаговое выполнение фоновой задачи с периодической отправкой сообщений о прогрессе. На каждой итерации цикла сначала приостанавливаем выполнение потока на 500 миллисекунд с помощью `Thread.sleep()`, имитируя таким образом обработку данных или выполнение ресурсоемкой операции. Затем создаем новый объект сообщения через метод `Message.obtain()`, что является оптимальным способом получения экземпляра `Message` из пула переиспользуемых объектов. В созданном сообщении устанавливаем поле `what` равным константе `MSG_UPDATE_COUNTER`, что позволяет идентифицировать это сообщение как обновление счетчика прогресса. Текущее значение счетчика `i` помещаем в поле `arg1` сообщения, которое специально предназначено для передачи целочисленных значений. Завершаем обработку итерации отправкой

подготовленного сообщения в главный поток через метод `sendMessage()` нашего Handler'a (Рисунок 15).

```
public class MainActivity extends AppCompatActivity {
    private static class BackgroundThread extends Thread {

        public void run() {
            try {
                // Отправляем сообщение о начале работы
                Message statusMsg = Message.obtain();
                statusMsg.what = MSG_UPDATE_STATUS;
                statusMsg.obj = "Initializing...";
                mainHandler.sendMessage(statusMsg);

                // Имитируем долгую задачу (например, загрузку или обработку)
                for (int i = 1; i <= 10; i++) {
                    Thread.sleep(500); // Имитация работы

                    // Отправляем сообщение с прогрессом
                    Message progressMsg = Message.obtain();
                    progressMsg.what = MSG_UPDATE_COUNTER;
                    progressMsg.arg1 = i;
                    mainHandler.sendMessage(progressMsg);

                    // Проверяем, не был ли поток прерван
                    if (isInterrupted()) {
                        return;
                    }
                }
            }
        }
    }
}
```

Рисунок 15 – Отправка сообщений о начале работы и о прогрессе ее выполнения

После завершения цикла отправляем финальное сообщение `MSG_TASK_COMPLETED` без дополнительных данных (Рисунок 16).

```
// Отправляем сообщение о завершении
Message completedMsg = Message.obtain();
completedMsg.what = MSG_TASK_COMPLETED;
mainHandler.sendMessage(completedMsg);
```

Рисунок 16 – Отправка сообщения о завершении

В блоке `catch` перехватываем `InterruptedException` и отправляем сообщение о прерывании задачи с соответствующим статусом (Рисунок 17).

```
} catch (InterruptedException e) {  
    // Поток был прерван  
    Message interruptedMsg = Message.obtain();  
    interruptedMsg.what = MSG_UPDATE_STATUS;  
    interruptedMsg.obj = "Task interrupted";  
    mainHandler.sendMessage(interruptedMsg);  
}
```

Рисунок 17 – Отправка сообщения при прерывании работы потока

В методе onCreate() активности MainActivity выполняем инициализацию пользовательского интерфейса и настройку механизма обработки сообщений. Сначала связываем переменные statusText, counterText и startButton с соответствующими элементами разметки через метод findViewById(). Затем создаем экземпляр Handler, привязанный к главному потоку приложения через Looper.getMainLooper(), и переопределяем его метод handleMessage() для обработки входящих сообщений.

В переопределенном handleMessage() реализуем логику обработки разных типов сообщений с использованием оператора switch по полю msg.what. Для сообщений типа MSG\_UPDATE\_STATUS обновляем текст статуса, используя данные из поля msg.obj. При получении сообщений MSG\_UPDATE\_COUNTER выводим значение счетчика из msg.arg1. В случае сообщения MSG\_TASK\_COMPLETED обновляем статус и активируем кнопку startButton.

Для кнопки startButton устанавливаем обработчик клика, который сначала деактивирует кнопку и устанавливает начальный статус выполнения задачи. Затем создаем и запускаем экземпляр фонового потока BackgroundThread, передавая ему наш Handler для возможности отправки сообщений в главный поток (Рисунок 18).

```

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        statusText = findViewById(R.id.statusText);
        counterText = findViewById(R.id.counterText);
        startButton = findViewById(R.id.startButton);

        // Создаем Handler для главного потока
        mainHandler = new Handler(Looper.getMainLooper()) {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    case MSG_UPDATE_STATUS:
                        statusText.setText("Status: " + msg.obj);
                        break;
                    case MSG_UPDATE_COUNTER:
                        counterText.setText("Counter: " + msg.arg1);
                        break;
                    case MSG_TASK_COMPLETED:
                        statusText.setText("Status: Task completed");
                        startButton.setEnabled(true);
                        break;
                }
            }
        };

        startButton.setOnClickListener(v -> {
            startButton.setEnabled(false);
            statusText.setText("Status: Task running...");

            // Создаем и запускаем фоновый поток
            backgroundThread = new BackgroundThread(mainHandler);
            backgroundThread.start();
        });
    }
}

```

Рисунок 18 – Создание Handler для главного потока и запуск фонового потока

В методе `onDestroy()` реализуем корректное завершение работы фонового потока при уничтожении активности. Вызываем метод `interrupt()` для `backgroundThread`, если он был инициализирован, что позволяет безопасно прервать выполнение фоновой задачи и избежать утечек ресурсов (Рисунок 19).

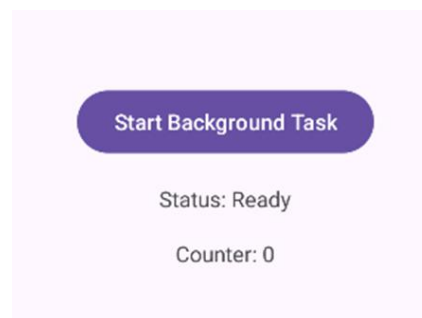
```

@Override
protected void onDestroy() {
    super.onDestroy();
    // Останавливаем фоновый поток при уничтожении активности
    if (backgroundThread != null) {
        backgroundThread.interrupt();
    }
}

```

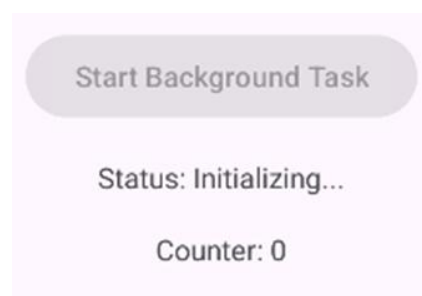
**Рисунок 19 – Остановка потока при уничтожении Activity**

Запустим приложение и убедимся, что изначальный интерфейс со статусом "Ready" и нулевым счётчиком свидетельствует о том, что главный поток свободен и готов к работе, а фоновые операции ещё не запущены (Рисунок 20).

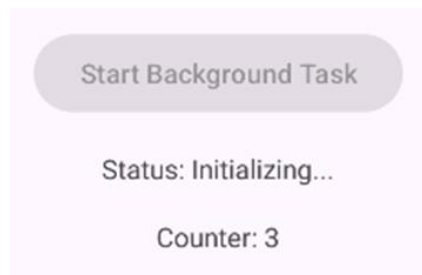


**Рисунок 20 – Демонстрация запуска приложения**

Нажмём кнопку и увидим мгновенное обновление статуса на "Initializing...", что демонстрирует работу Handler'a — сообщение успешно доставлено из фонового потока в главный через очередь сообщений. Плавное увеличение счётчика каждые 500 мс показывает, как фоновый поток последовательно отправляет сообщения через Handler, а главный поток асинхронно их обрабатывает, не блокируя интерфейс (Рисунки 21-22).

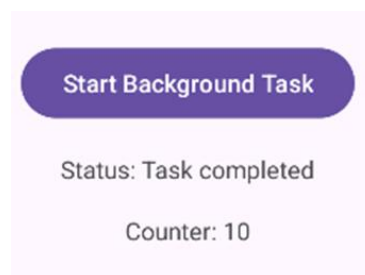


**Рисунок 21 – Запуск фонового потока**



**Рисунок 22 – Обновление UI во время выполнения работы в фоновом потоке**

Когда счётчик достигает 10 и статус меняется на "Task completed", это подтверждает корректное завершение работы фонового потока и успешную доставку финального сообщения (Рисунок 23).



**Рисунок 23 – Обновление UI по завершении выполнения работы в потоке**

## 1.4 WorkManager

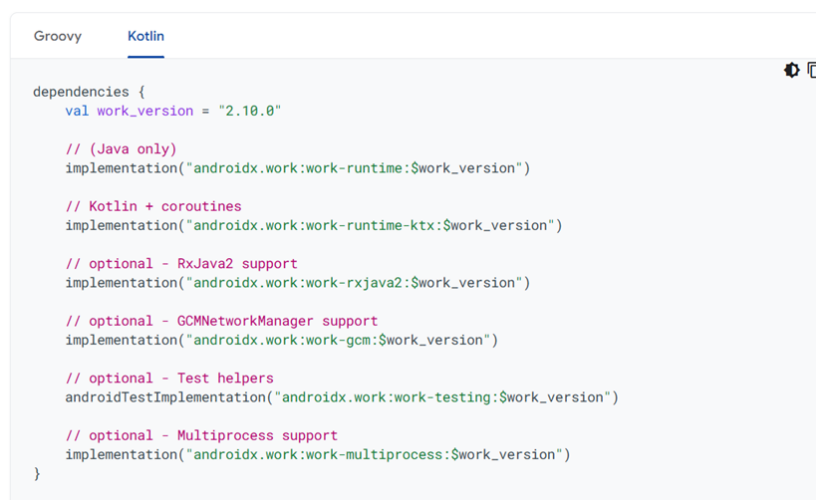
Одним из современных инструментов организации параллельного выполнения потоков в Android является класс WorkManager. С его помощью можно запускать последовательные и параллельные фоновые задачи, получать из них результат, передавать в них данные, отслеживать статус их выполнения и т.п.

WorkManager – довольно простая, при этом достаточно гибкая библиотека, обладающая множеством преимуществ, к которым можно отнести:

- поддержка разовых и периодических асинхронных задач,
- выполнение фоновых задач с учетом ограничений (например, ограничений состояния сети, наличия памяти для хранения данных, статуса зарядки и др.),
- возможность объединения сложных рабочих запросов в единую цепочку,

- возможность использования выходных данных одной задачи в качестве входных данных для следующей,
- поддержка LiveData для удобного отображения состояния рабочего запроса в пользовательском интерфейсе.

Для использования библиотеки WorkManager в раздел dependencies файла build.gradle необходимо добавить зависимости представленные на рисунке 24.



```
dependencies {
    val work_version = "2.10.0"

    // (Java only)
    implementation("androidx.work:work-runtime:$work_version")

    // Kotlin + coroutines
    implementation("androidx.work:work-runtime-ktx:$work_version")

    // optional - RxJava2 support
    implementation("androidx.work:work-rxjava2:$work_version")

    // optional - GCMNetworkManager support
    implementation("androidx.work:work-gcm:$work_version")

    // optional - Test helpers
    androidTestImplementation("androidx.work:work-testing:$work_version")

    // optional - Multiprocess support
    implementation("androidx.work:work-multiprocess:$work_version")
}
```

**Рисунок 24 – Добавление зависимостей**

После добавления зависимостей и синхронизации Gradle-проекта следующим шагом будет определение задач для выполнения.

Задачи определяются с использованием класса Worker. Метод doWork() выполняется асинхронно в фоновом потоке, предоставляемом WorkManager.

Для создания задачи нужно расширить класс Worker и переопределить метод doWork(). Например, создание Worker, который загружает изображения представлено на рисунке 25.



```

public class UploadWorker extends Worker {
    public UploadWorker(
        @NonNull Context context,
        @NonNull WorkerParameters params) {
        super(context, params);
    }

    @Override
    public Result doWork() {

        // Do the work here--in this case, upload the images.
        uploadImages();

        // Indicate whether the work finished successfully with the Result
        return Result.success();
    }
}

```

**Рисунок 25 – Создание задачи расширением класса Worker**

Результат (Result), возвращаемый из doWork(), информирует сервис WorkManager об успешности выполнения задачи и, в случае неудачи, нужно ли повторять попытку. Используются значения результата, возвращаемые следующими методами:

- Result.success(): задача успешно завершена,
- Result.failure(): задача завершилась неудачно,
- Result.retry(): задача не выполнена и должна быть повторена позже,

согласно политике повтора.

После определения задачи (Worker) её необходимо запланировать через сервис WorkManager для выполнения. WorkManager предоставляет гибкие возможности планирования задач: можно настроить периодическое выполнение через определённые интервалы времени или однократный запуск.

Независимо от выбранного способа планирования, всегда используется WorkRequest. Если Worker определяет саму задачу, то WorkRequest (и его подклассы) задают параметры выполнения: когда и как она должна запускаться. В простейшем случае можно использовать OneTimeWorkRequest, как показано на рисунке 26.

```
WorkRequest uploadWorkRequest =  
    new OneTimeWorkRequest.Builder(UploadWorker.class)  
        .build();
```

**Рисунок 26 – Создание WorkRequest**

Наконец, вам нужно отправить ваш WorkRequest в WorkManager, используя метод enqueue() (Рисунок 27).

```
WorkManager  
    .getInstance(myContext)  
    .enqueue(uploadWorkRequest);
```

**Рисунок 27 – Планирование экземпляра WorkRequest с помощью WorkManager**

Точное время выполнения Worker зависит от Constraints, указанных в WorkRequest, и оптимизаций системы. WorkManager разработан для обеспечения наилучшего поведения в рамках этих ограничений.

Объект WorkRequest содержит всю необходимую информацию для планирования и выполнения задачи в WorkManager. Он может включать в себя:

- constraints — условия, которые должны быть выполнены для запуска задачи,
- параметры планирования (задержки или интервалы повторения),
- настройки повторных попыток,
- входные данные (если задача их требует).

WorkRequest является абстрактным базовым классом. Существует две реализации:

- OneTimeWorkRequest — для разовых задач,
- PeriodicWorkRequest — для периодически повторяющихся задач.

Для базовых задач, не требующих дополнительной настройки, используйте статический метод from (Рисунок 28).

```
WorkRequest myWorkRequest = OneTimeWorkRequest.from(MyWork.class);
```

**Рисунок 28 – Использование метода from() для создания экземпляра WorkRequest**

Для более сложных задач применяйте билдер (Рисунок 29).

```
WorkRequest uploadWorkRequest =  
    new OneTimeWorkRequest.Builder(MyWork.class)  
        // Additional configuration  
        .build();
```

**Рисунок 29 – Использование билдера для создания экземпляра WorkRequest**

В вашем приложении может возникнуть необходимость периодического выполнения определенных задач. Например, вам может потребоваться регулярное создание резервных копий данных, загрузка нового контента или отправка логов на сервер.

Приведём пример использования `PeriodicWorkRequest` для создания `WorkRequest`, который выполняется периодически (Рисунок 30).

```
PeriodicWorkRequest saveRequest =  
    new PeriodicWorkRequest.Builder(SaveImageToFileWorker.class, 1, TimeUnit.HOURS)  
        // Constraints  
        .build();
```

**Рисунок 30 – Создание экземпляра PeriodicWorkRequest**

В данном примере работа запланирована с интервалом в один час.

Интервал определяется как минимальное время между повторениями. Точное время выполнения `worker` зависит от `constraints`, используемых в вашем объекте `WorkRequest`, и от оптимизаций, выполняемых системой.

Минимальный интервал повторения, который можно задать, составляет 15 минут.

Вы можете применять `constraints` к периодическим задачам. Например, можно добавить `constraint` к `work request`, чтобы задача выполнялась только при зарядке устройства. В этом случае, даже если заданный интервал повторения истёк, `PeriodicWorkRequest` не запустится, пока не будет выполнено это условие. Это может привести к задержке конкретного запуска задачи или даже его пропуску, если условия не выполняются в течение интервала выполнения.

`Constraints` обеспечивают выполнение задач только при соблюдении оптимальных условий. `WorkManager` поддерживает следующие типы ограничений:

- `NetworkType` — определяет тип сети, необходимый для выполнения работы. Например, Wi-Fi (`UNMETERED`),
- `BatteryNotLow` — если установлено значение `true`, работа не будет выполняться при низком уровне заряда батареи,
- `RequiresCharging` — при значении `true` задача запустится только во время зарядки устройства,
- `DeviceIdle` — если `true`, требует, чтобы устройство находилось в режиме простоя, что полезно для пакетных операций, которые могут повлиять на производительность других приложений,
- `StorageNotLow` — при `true` предотвращает выполнение работы при недостаточном объёме памяти на устройстве.

Для создания набора ограничений и их связывания с задачей создаем экземпляр `Constraints` с помощью `Constraints.Builder()` и назначаем его `WorkRequest.Builder()`.

Приведём пример кода, в котором создаётся запрос на выполнение работы, который запускается только при одновременном выполнении двух условий: устройство заряжается и подключено к Wi-Fi (Рисунок 31).

```
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.UNMETERED)
    .setRequiresCharging(true)
    .build();

WorkRequest myWorkRequest =
    new OneTimeWorkRequest.Builder(MyWork.class)
        .setConstraints(constraints)
        .build();
```

**Рисунок 31 – Добавление ограничений на запрос выполнения задачи**

При указании нескольких ограничений задача будет выполняться только когда соблюдены все условия.

Если в процессе выполнения работы какое-либо ограничение перестает соблюдаться, `WorkManager` останавливает `worker`. Повторная попытка выполнения произойдет только после восстановления всех требуемых условий.

Если ваша задача не имеет ограничений (constraints) или все ограничения уже выполнены на момент постановки в очередь (enqueued), система может начать выполнение работы немедленно. Если необходимо избежать немедленного запуска, вы можете указать минимальную начальную задержку перед выполнением.

Приведём пример кода по настройке запуска задачи не ранее чем через 10 минут после постановки в очередь (Рисунок 32).

```
WorkRequest myWorkRequest =  
    new OneTimeWorkRequest.Builder(MyWork.class)  
        .setInitialDelay(10, TimeUnit.MINUTES)  
        .build();
```

**Рисунок 32 – Добавление задержки перед выполнением**

Чтобы WorkManager повторно выполнил вашу задачу, необходимо вернуть Result.retry() из worker. В этом случае задача будет перепланирована с учетом backoff delay (задержки перед повтором) и backoff policy (политики повторных попыток).

Backoff delay определяет минимальное время ожидания перед повторной попыткой после первой неудачи. Это значение не может быть меньше 10 секунд (MIN\_BACKOFF\_MILLIS).

Backoff policy определяет, как будет увеличиваться задержка при последующих попытках. WorkManager поддерживает две политики: LINEAR (линейную) и EXPONENTIAL (экспоненциальную).

Каждый WorkRequest имеет свою backoff policy и backoff delay. По умолчанию используется политика EXPONENTIAL с задержкой 30 секунд, но эти параметры можно переопределить в конфигурации WorkRequest.

Приведём пример настройки пользовательских backoff delay и policy (Рисунок 33).

```
WorkRequest myWorkRequest =  
    new OneTimeWorkRequest.Builder(MyWork.class)  
        .setBackoffCriteria(  
            BackoffPolicy.LINEAR,  
            OneTimeWorkRequest.MIN_BACKOFF_MILLIS,  
            TimeUnit.MILLISECONDS)  
        .build();
```

**Рисунок 33 – Настройка перепланирования задачи**

В данном примере минимальная задержка установлена на минимально допустимое значение — 10 секунд. При политике LINEAR интервал между попытками будет увеличиваться примерно на 10 секунд с каждой новой попыткой. Например, первый возврат Result.retry() приведет к повтору через 10 секунд, затем через 20, 30, 40 и так далее, если задача продолжает возвращать Result.retry(). При политике EXPONENTIAL последовательность задержек будет примерно 20, 40, 80 секунд.

Каждый WorkRequest обладает уникальным идентификатором, позволяющим впоследствии идентифицировать задачу — для её отмены или наблюдения за прогрессом выполнения. Если у вас есть группа логически связанных задач, вы можете пометить их тегами (tags), что обеспечивает удобное управление всей группой WorkRequest как единым целым.

Например, метод WorkManager.cancelAllWorkByTag(String) отменяет все WorkRequest с указанным тегом, а WorkManager.getWorkInfosByTag(String) возвращает список объектов WorkInfo для отслеживания текущего состояния задач.

В следующем примере кода показано, как добавить тег "cleanup" к WorkRequest (Рисунок 34).

```
WorkRequest myWorkRequest =  
    new OneTimeWorkRequest.Builder(MyWork.class)  
        .addTag("cleanup")  
        .build();
```

**Рисунок 34 – Добавление тега**

К одному `WorkRequest` можно добавить несколько тегов — внутренне они хранятся как множество строк. Для получения тегов, связанных с `WorkRequest`, используйте метод `WorkInfo.getTags()`. Кроме того, из класса `Worker` можно получить набор тегов через метод `ListenableWorker.getTags()`.

Ваша задача может требовать входных данных для выполнения работы. Например, задача по загрузке изображения может нуждаться в URI этого изображения в качестве входного параметра.

Входные значения хранятся в виде пар "ключ-значение" в объекте `Data` и могут быть установлены в `WorkRequest`. `WorkManager` передаст входные данные (input Data) вашей задаче при её выполнении. Класс `Worker` получает доступ к этим аргументам через метод `Worker.getInputData()`.

Следующий пример кода демонстрирует создание экземпляра `Worker`, требующего входных данных, и способ их передачи в `WorkRequest` (Рисунок 35).

```
// Define the Worker requiring input
public class UploadWork extends Worker {

    public UploadWork(Context appContext, WorkerParameters workerParams) {
        super(appContext, workerParams);
    }

    @NonNull
    @Override
    public Result doWork() {
        String imageUriInput = getInputData().getString("IMAGE_URI");
        if(imageUriInput == null) {
            return Result.failure();
        }

        uploadFile(imageUriInput);
        return Result.success();
    }
    ...
}

// Create a WorkRequest for your Worker and sending it input
WorkRequest myUploadWork =
    new OneTimeWorkRequest.Builder(UploadWork.class)
        .setInputData(
            new Data.Builder()
                .putString("IMAGE_URI", "http://...")
                .build()
        )
        .build();
```

**Рисунок 35 – Работа с входными данными**

Аналогично, класс `Data` может использоваться для возврата результата выполнения задачи.

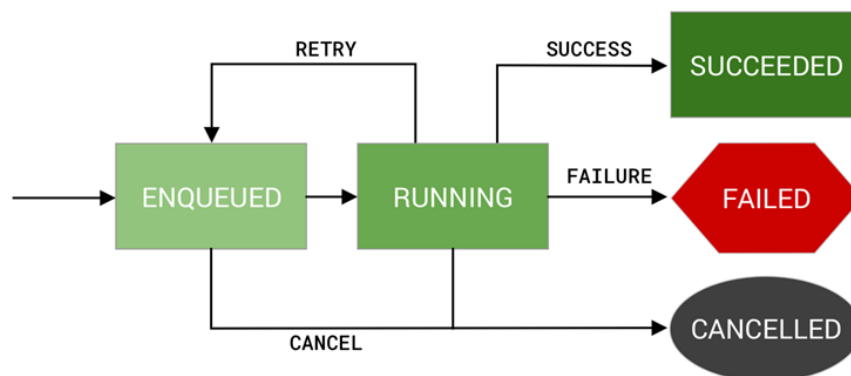


В методе `doWork()` вашего `Worker` вы можете создать объект `Data` с выходными значениями и вернуть его через `Result.success(Data)` или `Result.failure(Data)`. Это позволяет передавать результаты выполнения задачи (например, статус операции или обработанные данные) другим компонентам приложения.

`WorkManager` сохраняет эти выходные данные (output Data), и они могут быть получены через `WorkInfo.getOutputData()` при наблюдении за состоянием задачи.

В течение своего жизненного цикла задача проходит через последовательность состояний (State).

Для `OneTimeWorkRequest` работа начинается в состоянии `ENQUEUED`. В этом состоянии задача становится готовой к выполнению, как только будут соблюдены все `Constraints` и требования к начальной задержке. Затем она переходит в состояние `RUNNING`, а после завершения — в `SUCCEEDED`, `FAILED` или снова в `ENQUEUED` (если возвращен `Result.retry()`). В любой момент задачу можно отменить, переведя её в состояние `CANCELLED` (Рисунок 36).



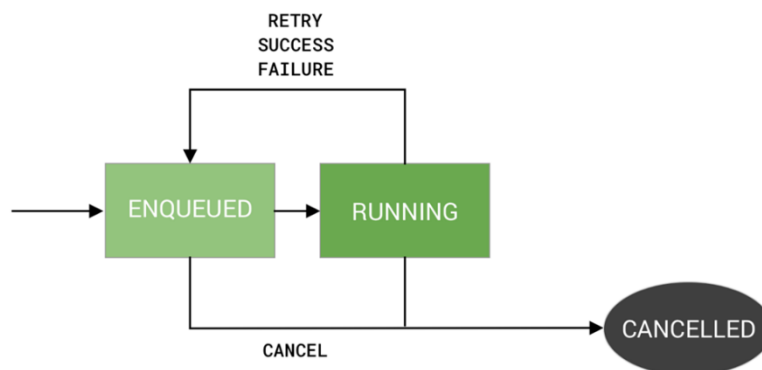
**Рисунок 36 – Состояния жизненного цикла `OneTimeWorkRequest`**

Состояния `SUCCEEDED`, `FAILED` и `CANCELLED` являются терминальными — для них `WorkInfo.State.isFinished()` возвращает `true`.

Успешное и неудачное завершение применимы только к разовым и связанным задачам. Для периодических задач (`PeriodicWorkRequest`) единственным терминальным состоянием является `CANCELLED`, так как такие



задачи никогда не завершаются окончательно — после каждого выполнения они перепланируются, независимо от результата (Рисунок 37).



**Рисунок 37 – Состояния жизненного цикла `PeriodicWorkRequest`**

Отдельно стоит состояние `BLOCKED`, которое применяется к задачам, организованным в цепочки.

После определения `Worker` и `WorkRequest` финальным шагом является постановка задачи в очередь с помощью метода `enqueue()` в `WorkManager`, куда передается созданный `WorkRequest` (Рисунок 38).

```
WorkRequest myWork = // ... OneTime or PeriodicWork
WorkManager.getInstance(requireContext()).enqueue(myWork);
```

**Рисунок 38 – Постановка задачи в очередь**

Важно соблюдать осторожность при постановке задач, чтобы избежать дублирования. Например, если приложение пытается выгружать логи в сервис каждые 24 часа, некорректная реализация может привести к многократному добавлению одинаковых задач. Для предотвращения этого можно использовать уникальные задачи (`unique work`).

Уникальные задачи (`unique work`) — это мощный механизм, гарантирующий существование только одного экземпляра задачи с определённым именем в любой момент времени. В отличие от ID, уникальные имена задаются разработчиком в читаемом формате, а не генерируются `WorkManager` автоматически. В отличие от тегов (`tags`), уникальное имя может быть связано только с одним экземпляром задачи.

Unique work применяется как к разовым (one-time), так и к периодическим (periodic) задачам. Для создания уникальной последовательности задач используются следующие методы в зависимости от типа:

- `WorkManager.enqueueUniqueWork()` — для разовых задач,
- `WorkManager.enqueueUniquePeriodicWork()` — для периодических задач.

Оба метода принимают три аргумента:

- `uniqueWorkName` — строка для уникальной идентификации задачи,
- `existingWorkPolicy` — перечисление (enum), определяющее поведение `WorkManager` при обнаружении существующей незавершённой задачи с таким же именем,
- `work` — объект `WorkRequest` для планирования.

Использование unique work решает проблему дублирования задач, описанную ранее (Рисунок 39).

```
PeriodicWorkRequest sendLogsWorkRequest = new
    PeriodicWorkRequest.Builder(SendLogsWorker.class, 24, TimeUnit.HOURS)
        .setConstraints(new Constraints.Builder()
            .setRequiresCharging(true)
            .build()
        )
        .build();
WorkManager.getInstance(this).enqueueUniquePeriodicWork(
    "sendLogs",
    ExistingPeriodicWorkPolicy.KEEP,
    sendLogsWorkRequest);
```

**Рисунок 39 – Использование уникальных задач**

При планировании уникальных задач (unique work) необходимо указать `WorkManager`, как действовать в случае конфликта. Это делается путём передачи перечисления (enum) при постановке задачи в очередь.

Для разовых задач (one-time work) используется `ExistingWorkPolicy`, предлагающий 4 варианта обработки конфликтов:

- `REPLACE` — заменяет существующую задачу новой. Текущая задача отменяется,

- KEEP — сохраняет существующую задачу и игнорирует новую,
- APPEND — добавляет новую задачу в конец существующей, создавая цепочку. Новая задача запустится только после успешного завершения текущей. Если существующая задача отменена (CANCELLED) или завершилась неудачно (FAILED), новая задача также отменяется,
- APPEND\_OR\_REPLACE — аналогично APPEND, но не зависит от статуса предыдущей задачи. Даже если существующая задача отменена или провалена, новая всё равно выполнится.

Для периодических задач (periodic work) используется ExistingPeriodicWorkPolicy, где оба варианта — REPLACE и KEEP — аналогичны одноимённым политикам для разовых задач.

В любой момент после постановки задачи в очередь вы можете проверить её статус, запросив WorkManager по имени задачи, её id или связанному тегу (Рисунок 40).

```
// by id
workManager.getWorkInfoById(syncWorker.id); // ListenableFuture<WorkInfo>

// by name
workManager.getWorkInfosForUniqueWork("sync"); // ListenableFuture<List<WorkInfo>>

// by tag
workManager.getWorkInfosByTag("syncTag"); // ListenableFuture<List<WorkInfo>>
```

**Рисунок 40 – Запрос данных о поставленной в очередь задаче**

Запрос возвращает ListenableFuture<WorkInfo>, содержащий информацию о задаче: её id, теги, текущее состояние (State) и выходные данные (output data), если они были установлены через Result.success(outputData).

Каждый из этих методов также имеет вариант с LiveData, позволяющий отслеживать изменения WorkInfo путём регистрации наблюдателя. Например, если необходимо показать пользователю сообщение об успешном завершении задачи, это можно настроить следующим образом (Рисунок 41).

```
workManager.getWorkInfoByIdLiveData(syncWorker.id)
    .observe(getViewLifecycleOwner(), workInfo -> {
        if (workInfo.getState() != null &&
            workInfo.getState() == WorkInfo.State.SUCCEEDED) {
            Snackbar.make(requireView(),
                R.string.work_completed, Snackbar.LENGTH_SHORT)
                .show();
        }
    });
```

**Рисунок 41 – Отслеживание статуса задачи**

Если вам больше не требуется выполнение ранее добавленной в очередь задачи, вы можете запросить её отмену. Отмена возможна по имени задачи, её идентификатору или связанному тегу (Рисунок 42).

```
// by id
workManager.cancelWorkById(syncWorker.id);

// by name
workManager.cancelUniqueWork("sync");

// by tag
workManager.cancelAllWorkByTag("syncTag");
```

**Рисунок 42 – Запрос отмены задачи**

На уровне реализации WorkManager проверяет текущее состояние (State) задачи. Если задача уже завершена, никаких действий не происходит. В противном случае её состояние меняется на CANCELLED, и задача больше не будет выполняться. Все зависимые WorkRequest, связанные с этой задачей, также будут отменены.

Для задач, находящихся в состоянии RUNNING, вызывается метод ListenableWorker.onStopped(). Переопределите этот метод, чтобы реализовать необходимые операции очистки при принудительной остановке задачи.

Существует несколько причин, по которым WorkManager может остановить выполняющийся Worker:

- явный запрос на отмену, например, через метод WorkManager.cancelWorkById(UUID),
- для уникальных задач — при добавлении нового WorkRequest с политикой ExistingWorkPolicy.REPLACE, старый WorkRequest немедленно считается отменённым,

- нарушение условий Constraints для данной задачи,
- системное требование остановить работу (например, при превышении 10-минутного лимита выполнения). В этом случае задача будет перепланирована на более позднее время.

В этом случае вам следует корректно прервать выполняемую работу и освободить все ресурсы, которые использует ваш Worker. Например, необходимо закрыть открытые подключения к базам данных и файлам в этот момент.

WorkManager позволяет создавать и ставить в очередь цепочки задач, где можно определить несколько зависимых операций и порядок их выполнения. Эта возможность особенно полезна, когда требуется выполнить несколько задач в строгой последовательности.

Для создания цепочки используйте `WorkManager.beginWith(OneTimeWorkRequest)` или `WorkManager.beginWith(List<OneTimeWorkRequest>)`, которые возвращают экземпляр `WorkContinuation`.

Затем с помощью `WorkContinuation` можно добавлять зависимые `OneTimeWorkRequest`, используя методы `then(OneTimeWorkRequest)` или `then(List<OneTimeWorkRequest>)`. Каждый вызов `WorkContinuation.then(...)` создаёт новый экземпляр `WorkContinuation`. Если передаётся список `OneTimeWorkRequest`, эти задачи могут выполняться параллельно.

Финальным шагом цепочка задач ставится в очередь методом `WorkContinuation.enqueue()`.

Рассмотрим пример. Три разных Worker настраиваются для (потенциально параллельного) выполнения. Их результаты объединяются и передаются Worker для кэширования, после чего итоговые данные загружаются на удалённый сервер через Worker для выгрузки (Рисунок 43).

```
WorkManager.getInstance(myContext)
    // Candidates to run in parallel
    .beginWith(Arrays.asList(plantName1, plantName2, plantName3))
    // Dependent work (only runs after all previous work in chain)
    .then(cache)
    .then(upload)
    // Call enqueue to kick things off
    .enqueue();
```

**Рисунок 43 – Создание цепочки задач**

Цепочки `OneTimeWorkRequest` выполняются последовательно при условии успешного завершения каждой задачи (то есть, когда возвращается `Result.success()`). Если какая-либо задача завершается с ошибкой (`Result.failure()`) или отменяется, это влияет на выполнение всех зависимых задач в цепочке.

При создании цепочек из `OneTimeWorkRequest` выходные данные (output) родительских задач автоматически передаются как входные данные (input) дочерним задачам. Например, в описанном сценарии результаты `plantName1`, `plantName2` и `plantName3` становятся входными данными для задачи кэширования (cache request).

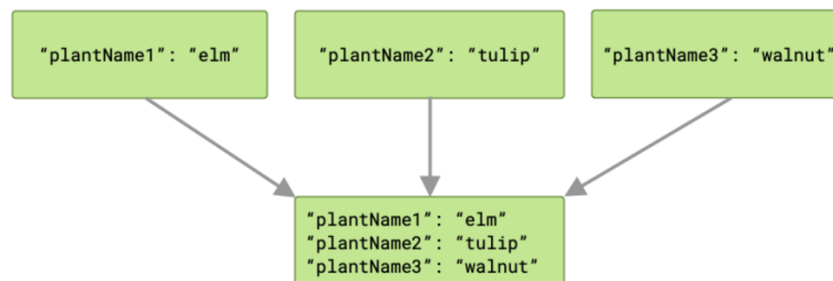
Для объединения входных данных от нескольких родительских задач `WorkManager` использует `InputMerger`. Доступно два стандартных типа:

- `OverwritingInputMerger`: добавляет все ключи из всех входных данных в результат. При конфликтах ключей перезаписывает предыдущие значения новыми,
- `ArrayCreatingInputMerger`: объединяет входные данные, автоматически создавая массивы для совпадающих ключей.

Для нестандартных сценариев можно создать собственный `InputMerger`, унаследовав его от базового класса.

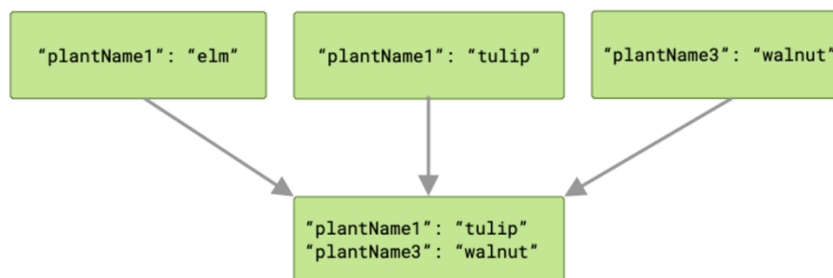
`OverwritingInputMerger` используется по умолчанию для слияния входных данных в `WorkManager`. При обнаружении одинаковых ключей в выходных данных родительских задач последнее записанное значение перезаписывает предыдущие.

На примере задач `plantName1`, `plantName2` и `plantName3`: если каждая задача возвращает данные с уникальными ключами (например, `"plantName1"`, `"plantName2"`, `"plantName3"`), то результат слияния будет содержать все три пары ключ-значение (Рисунок 44).



**Рисунок 44 – Работа `OverwritingInputMerger` при уникальности ключей**

Однако при совпадении ключей сохранится только значение из последней завершённой задачи — порядок выполнения параллельных задач не гарантирован, поэтому результат может варьироваться (Рисунок 45).



**Рисунок 45 – Работа `OverwritingInputMerger`**

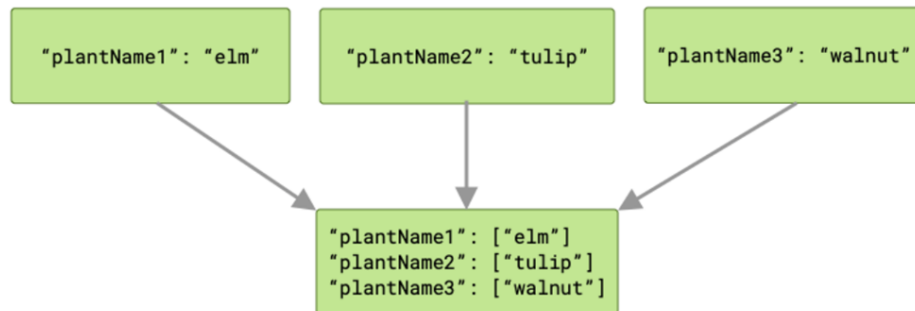
Если требуется сохранить все значения при конфликте ключей, вместо `OverwritingInputMerger` следует использовать `ArrayCreatingInputMerger`, который автоматически объединяет данные в массивы.

Для рассматриваемого примера, где необходимо сохранить выходные данные всех `Workers` (`plantName1`, `plantName2` и `plantName3`), следует использовать `ArrayCreatingInputMerger` (Рисунок 46).

```
OneTimeWorkRequest cache = new OneTimeWorkRequest.Builder(PlantWorker.class)
    .setInputMerger(ArrayCreatingInputMerger.class)
    .setConstraints(constraints)
    .build();
```

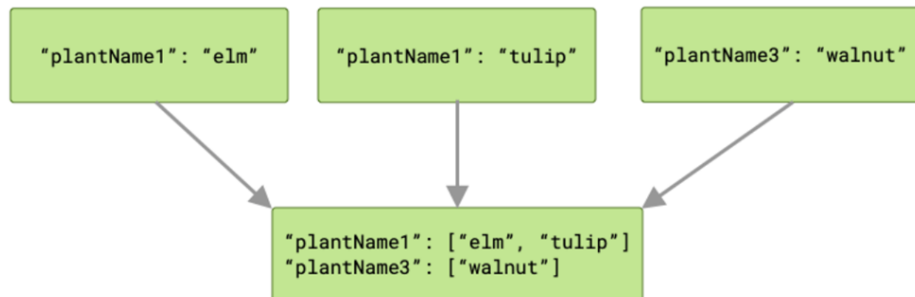
**Рисунок 46 – Использование `ArrayCreatingInputMerger`**

ArrayCreatingInputMerger связывает каждый ключ с массивом значений. Если все ключи уникальны, результатом будет набор массивов, каждый из которых содержит один элемент (Рисунок 47).



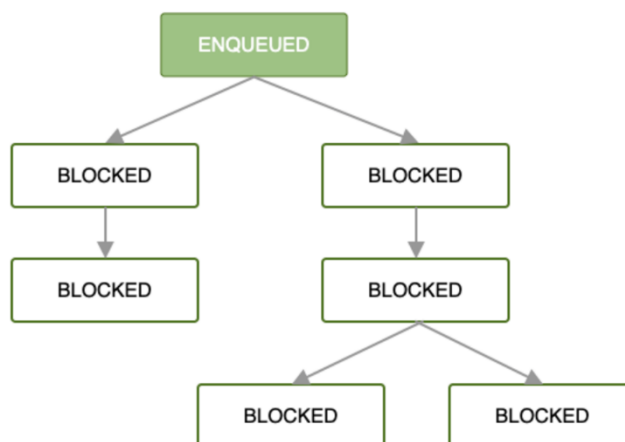
**Рисунок 47 – Работа ArrayCreatingInputMerger при уникальности ключей**

В случае совпадения ключей соответствующие значения группируются в массив, что позволяет сохранить все выходные данные без потерь (Рисунок 48).



**Рисунок 48 – Работа ArrayCreatingInputMerger при повторении ключей**

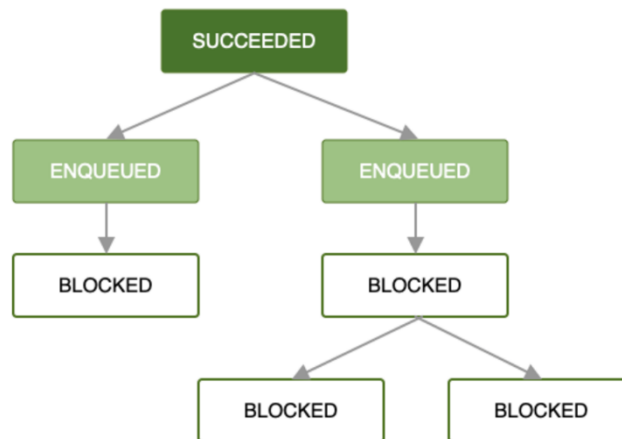
Когда первый OneTimeWorkRequest ставится в очередь, все последующие задачи блокируются до его завершения. WorkManager автоматически управляет этой зависимостью, гарантируя соблюдение порядка выполнения (Рисунок 49).



**Рисунок 49 – Блокировка зависимых задач при постановке корневой задачи в очередь**



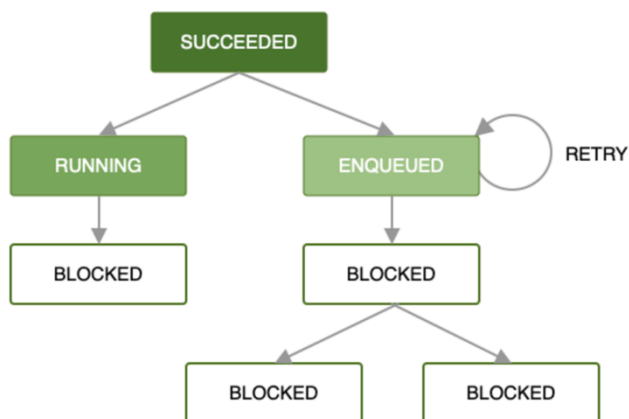
После постановки в очередь и выполнения всех условий (Constraints) запускается первый WorkRequest. Если корневая задача (OneTimeWorkRequest) или список задач (List<OneTimeWorkRequest>) завершается успешно (возвращая Result.success()), WorkManager автоматически ставит в очередь следующий набор зависимых задач (Рисунок 50).



**Рисунок 50 – Автоматическая постановка в очередь зависимых задач после успешного выполнения текущей**

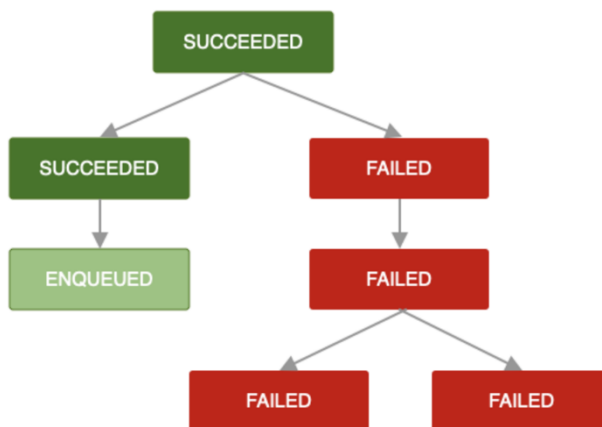
Такой порядок выполнения сохраняется на протяжении всей цепочки задач — каждая последующая WorkRequest ставится в очередь только после успешного завершения (Result.success()) предыдущей. Этот процесс продолжается, пока все задачи в цепочке не будут выполнены. Хотя такой сценарий является базовым, обработка ошибок не менее важна.

Если при выполнении WorkRequest возникает ошибка, система может повторить попытку согласно заданной политике повтора (backoff policy). Повтор затрагивает только конкретную задачу, используя исходные входные данные, и не влияет на параллельно выполняемые задачи в цепочке (Рисунок 51).



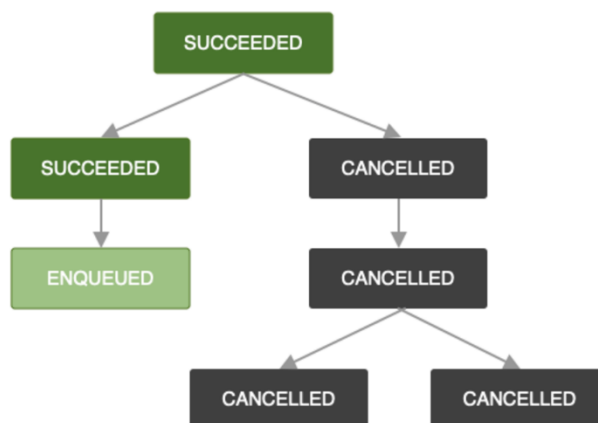
**Рисунок 51 – Повторная попытка выполнения задачи в цепочке**

Если политика повторных попыток (retry policy) не определена, исчерпана или если `OneTimeWorkRequest` возвращает `Result.failure()`, то данный `WorkRequest` и все зависимые задачи в цепочке помечаются как `FAILED` (Рисунок 52).



**Рисунок 52 – Ошибка выполнения задачи и ее влияние на цепочку**

Аналогичная логика применяется при отмене `OneTimeWorkRequest`: все зависимые `WorkRequest` автоматически помечаются как `CANCELLED`, и их выполнение не производится (Рисунок 53).



**Рисунок 53 – Отмена выполнения задачи и ее влияние на очередь**

Важно учитывать, что при добавлении новых `WorkRequest` к очереди, содержащей `FAILED` или `CANCELLED` задачи, эти новые задачи автоматически получают соответствующий статус (`FAILED` или `CANCELLED`). Для расширения уже существующей очереди порой имеет смысл использовать политику `APPEND_OR_REPLACE` из `ExistingWorkPolicy`, которая позволяет добавить задачи независимо от состояния предыдущих.

При создании очередей задач рекомендуется всегда определять политики повторных попыток (`retry policies`) для зависимых `WorkRequest`.

## 2 ПРАКТИЧЕСКОЕ ЗАДАНИЕ

### 2.1 Разметка

После создания проекта заполним файл разметки activity\_main.xml необходимыми элементами.

Для тестирования потоков было создано 3 кнопки (Рисунок 54).

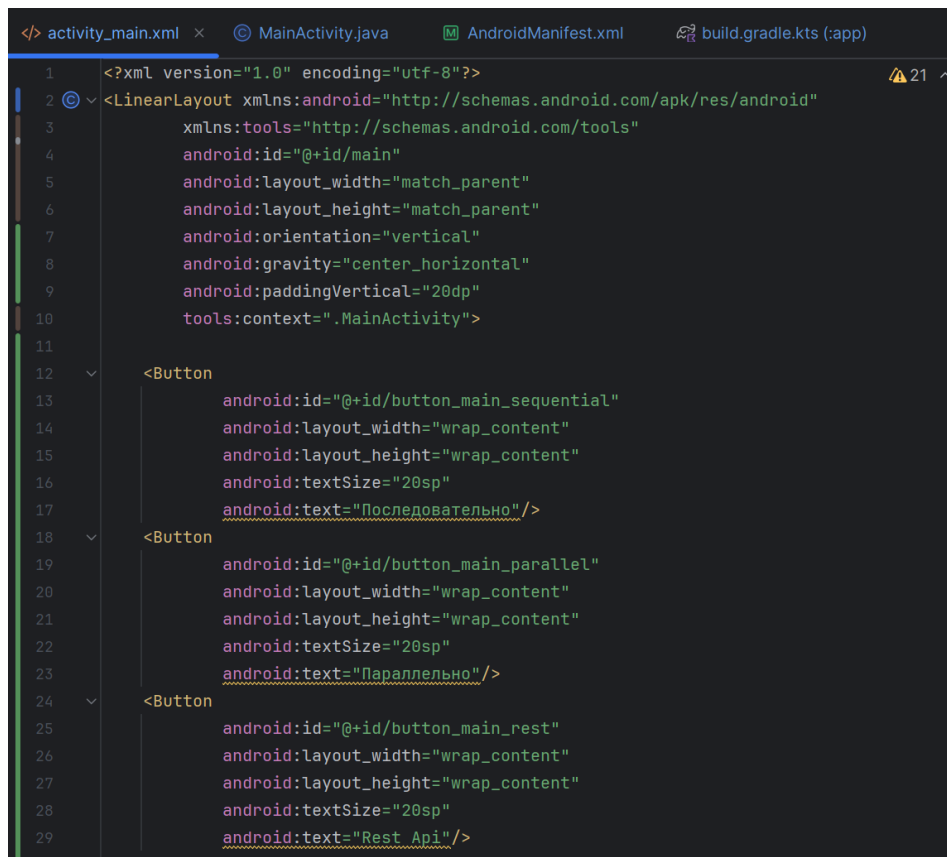


Рисунок 54 – Кнопки в файле разметки activity\_main.xml

Для последовательных потоков были добавлены таблички с временем начала и конца работы каждого из потоков (Рисунки 55-56).

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="25sp"
    android:text="Последовательные задачи"/>
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="*">
    <TableRow>
        <TextView
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Время начала"/>
        <TextView
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Время окончания"/>
    </TableRow>
    <TableRow>
        <TextView
            android:id="@+id/text_main_seq1Start"
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="He началась"/>
        <TextView
            android:id="@+id/text_main_seq1End"
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="He закончилась"/>
    </TableRow>
</TableLayout>

```

**Рисунок 55 – Таблица временных меток последовательных потоков, часть 1**

```

<TableRow>
    <TextView
        android:id="@+id/text_main_seq2Start"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="He началась"/>
    <TextView
        android:id="@+id/text_main_seq2End"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="He закончилась"/>
</TableRow>
<TableRow>
    <TextView
        android:id="@+id/text_main_seq3Start"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="He началась"/>
    <TextView
        android:id="@+id/text_main_seq3End"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="He закончилась"/>
</TableRow>
</TableLayout>

```

**Рисунок 56 – Таблица временных меток последовательных потоков, часть 2**

Аналогичная таблица была создана для параллельных потоков (Рисунки 57-58).

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="25sp"
    android:text="Параллельные задачи"/>
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="*>
    <TableRow>
        <TextView
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Время начала"/>
        <TextView
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Время окончания"/>
    </TableRow>
    <TableRow>
        <TextView
            android:id="@+id/text_main_par1Start"
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Не началась"/>
        <TextView
            android:id="@+id/text_main_par1End"
            android:layout_width="0dp"
            android:gravity="center"
            android:textSize="20sp"
            android:text="Не закончилась"/>
    </TableRow>
</TableLayout>

```

Рисунок 57 – Таблица временных меток параллельных потоков, часть 1

```

<TableRow>
    <TextView
        android:id="@+id/text_main_par2Start"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="Не началась"/>
    <TextView
        android:id="@+id/text_main_par2End"
        android:layout_width="0dp"
        android:gravity="center"
        android:textSize="20sp"
        android:text="Не закончилась"/>
</TableRow>
</TableLayout>

```

Рисунок 58 – Таблица временных меток параллельных потоков, часть 2

Для картинки, получаемой с помощью Rest API было выделено место внизу страницы (Рисунок 59).

```

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="25sp"
    android:text="Rest API"/>
<ImageView
    android:id="@+id/image_main_restApi"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</LinearLayout>

```

Рисунок 59 – Место для картинки, получаемой с помощью Rest API

## 2.2 Реализация

### 2.2.1 Последовательные и параллельные потоки

Для реализации многопоточности был использован механизм Handler. Для его использования был реализован класс MyThread, в который передается экземпляр класса Handler при его создании (Рисунок 60).

```
10 usages
public class MyThread extends Thread
{
    2 usages
    private final Handler mainHandler;
    2 usages
    private Runnable onFinish;
    2 usages
    private final long duration;

    5 usages
    public MyThread(Handler mainHandler, long duration)
    {
        this.mainHandler = mainHandler;
        this.duration = duration;
    }

    5 usages
    public void setOnFinished(Runnable onFinish)
    {
        this.onFinish = onFinish;
    }

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(duration);
            mainHandler.post(onFinish);
        }
        catch (InterruptedException ignored) { }
    }
}
```

Рисунок 60 – Реализация класса MyThread

Данный класс эмулирует долгую работу с помощью входного параметра `duration`, который передается в `thread.sleep()`. Также в класс передается метод, который требуется выполнить при завершении работы потока.

Для запуска последовательных и параллельных потоков были использованы методы `startSequentialThreads()` и `startParallelThreads()` соответственно (Рисунок 61). В методах также используется метод `setCurrentTime()`, который устанавливает текст элемента `TextView` по данному `id` в текущее время.

```
1 usage
private void startSequentialThreads(Handler mainHandler)
{
    MyThread sequentialThread1 = new MyThread(mainHandler, duration: 1000);
    MyThread sequentialThread2 = new MyThread(mainHandler, duration: 2000);
    MyThread sequentialThread3 = new MyThread(mainHandler, duration: 3000);
    sequentialThread1.setOnFinished(() ->
    {
        setCurrentTime(R.id.text_main_seq1End);
        setCurrentTime(R.id.text_main_seq2Start);
        sequentialThread2.start();
    });
    sequentialThread2.setOnFinished(() ->
    {
        setCurrentTime(R.id.text_main_seq2End);
        setCurrentTime(R.id.text_main_seq3Start);
        sequentialThread3.start();
    });
    sequentialThread3.setOnFinished(() ->
    {
        setCurrentTime(R.id.text_main_seq3End);
    });
    sequentialThread1.start();
}

1 usage
private void startParallelThreads(Handler mainHandler)
{
    MyThread parallelThread1 = new MyThread(mainHandler, duration: 2000);
    MyThread parallelThread2 = new MyThread(mainHandler, duration: 4000);
    parallelThread1.setOnFinished(() -> setCurrentTime(R.id.text_main_par1End));
    parallelThread2.setOnFinished(() -> setCurrentTime(R.id.text_main_par2End));
    setCurrentTime(R.id.text_main_par1Start);
    setCurrentTime(R.id.text_main_par2Start);
    parallelThread1.start();
    parallelThread2.start();
}
```

Рисунок 61 – Описание методов `startSequentialThreads()` и `startParallelThreads()`

В методе `onCreate()` класса `MainActivity()` происходит инициализация функционала кнопок для запуска потоков (Рисунок 62).



```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    EdgeToEdge.enable( $this$enableEdgeToEdge: this);
    setContentView(R.layout.activity_main);

    Handler mainHandler = new Handler(Looper.getMainLooper());
    Button buttonSequential = findViewById(R.id.button_main_sequential);
    buttonSequential.setOnClickListener(v ->
    {
        setCurrentTime(R.id.text_main_seq1Start);
        startSequentialThreads(mainHandler);
    });
    Button buttonParallel = findViewById(R.id.button_main_parallel);
    buttonParallel.setOnClickListener(v ->
    {
        startParallelThreads(mainHandler);
    });
    Button buttonRest = findViewById(R.id.button_main_rest);
    buttonRest.setOnClickListener(v ->
    {
        RestThread imageThread = new RestThread(mainHandler);
        imageThread.start();
    });
}

10 usages
private void setCurrentTime(int textViewId)
{
    TextView text = findViewById(textViewId);
    String time = DateTimeFormatter.ofPattern("HH:mm:ss").format(LocalTime.now());
    text.setText(time);
}

```

Рисунок 62 – Метод onCreate() класса MainActivity

### 2.2.2 Работа с Rest API

Для работы с Rest API был реализован класс потока RestThread.

В методе run() с помощью метода getImage() получается картинка и передается в mainHandler для установки в соответствующий элемент ImageView (Рисунок 63).

```

2 usages
public class RestThread extends Thread
{
    3 usages
    private final Handler mainHandler;

    1 usage
    public RestThread(Handler mainHandler)
    {
        this.mainHandler = mainHandler;
    }

    @Override
    public void run()
    {
        Bitmap image = getImage();
        if (image == null)
        {
            mainHandler.post() ->
            Toast.makeText(context: MainActivity.this,
                text: "При загрузке картинки произошла ошибка",
                Toast.LENGTH_SHORT).show());
            return;
        }
        mainHandler.post() ->
        {
            ImageView imageView = findViewById(R.id.image_main_restApi);
            imageView.setImageBitmap(image);
        });
    }
}

```

**Рисунок 63 – Реализация метода run() класса RestThread**

В методе getImage() происходит получение ссылки на картинку и загрузка Bitmap с применением java классов URL, HttpURLConnection и InputStream (Рисунок 64).

```

1 usage
private Bitmap getImage()
{
    try
    {
        URL imageUrl = new URL(getStringByURL("https://random.dog/woof.json"));
        HttpURLConnection imageConnection = (HttpURLConnection) imageUrl.openConnection();
        imageConnection.setDoInput(true);
        imageConnection.connect();
        InputStream input = imageConnection.getInputStream();
        return BitmapFactory.decodeStream(input);
    }
    catch (IOException e)
    {
        return null;
    }
}

```

**Рисунок 64 – Реализация метода getImage() класса RestThread**

В методе также используется метод `getStringByUrl()`, реализация которого представлена на рисунке 65.

```
1 usage
private String getStringByUrl(String url) throws IOException
{
    URL resourceUrl = new URL(url);
    HttpURLConnection resourceConnection = (HttpURLConnection) resourceUrl.openConnection();
    if (resourceConnection.getResponseCode() == 200)
    {
        InputStream response = resourceConnection.getInputStream();
        InputStreamReader responseReader = new InputStreamReader(response, charsetName: "UTF-8");
        JsonReader jsonReader = new JsonReader(responseReader);
        jsonReader.beginObject();
        while (jsonReader.hasNext())
        {
            String name = jsonReader洗洗Name();
            if (!name.equals("url"))
            {
                jsonReader.skipValue();
                continue;
            }
            return jsonReader.nextString();
        }
        return null;
    }
    return null;
}
```

Рисунок 65 – Реализация метода `getStringByUrl()` класса `RestThread`

## 2.3 Тестирование

Откроем приложение (Рисунок 66).

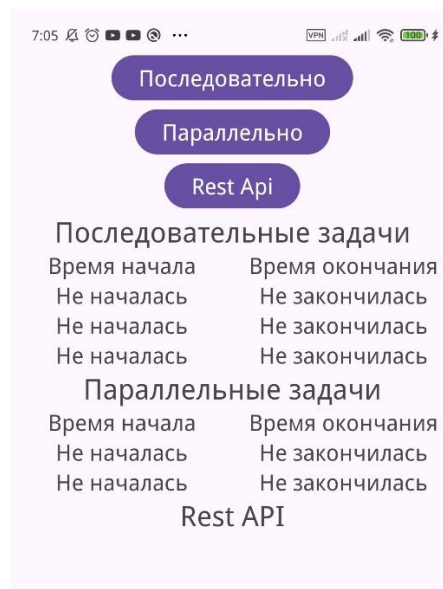
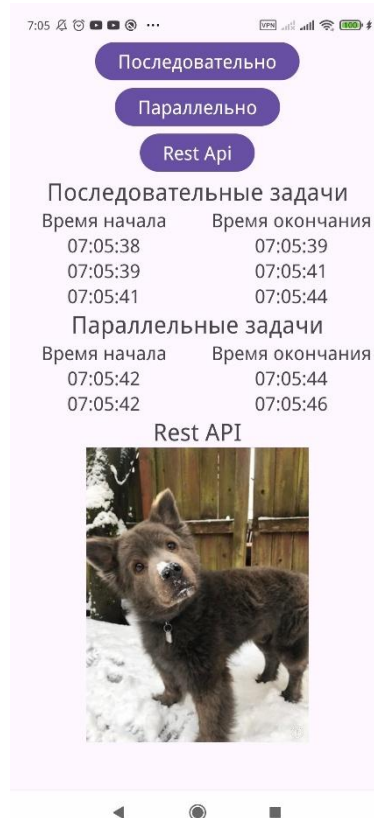


Рисунок 66 – Начальный экран приложения

Запустим каждый из потоков посредством нажатия кнопок (Рисунок 67).



**Рисунок 67 – Результат запуска потоков**

Все потоки работают успешно, а поток пользовательского интерфейса не блокируется при выполнении запросов или других долгих действий.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения работы были успешно реализованы механизмы последовательного и параллельного выполнения задач с использованием Handler и Thread, что позволило на практике изучить особенности организации многопоточности. Дополнительно была реализована загрузка и отображение данных, полученных через обращение к REST API.