Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерный наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий.

**Отчёт по лабораторной работе №6**

**Дисциплина**: Проектирование мобильных приложений

**Тема**: Многопоточные Android приложения. Вариант 13

Выполнил студент гр. 3530901/80201: ­­­­­­­­­­­­­ ­ В.Д. Петров

(подпись)

Преподаватель: ­А.Н. Кузнецов

(подпись)

Санкт-Петербург

2020

**СОДЕРЖАНИЕ**

Введение…………………………………………………………………………….3

Задача №1. Альтернативное решение задачи “не секундомер”…………………5

Задача №2. Загрузка картинки в фоновом потоке (AsyncTask)………………...13

Задача №3. Загрузка картинки в фоновом потоке (Kotlin Coroutines)………….16

Задача №4. Использование сторонних библиотек……………………………....18

Заключение……………………………………………………………………..…19

Список использованных источников………………………………………...…..20

**ВВЕДЕНИЕ**

Целями данной работы являются получение практически навыков разработки многопоточных приложений – организация обработки длительных операций в background (worker) thread и публикация данных из background (worker) thread в main (ui) thread, а также освоение 3 основных групп API для разработки многопоточных приложений: Kotlin Coroutines, AsyncTask и Java Threads.

В Android важно избегать блокировки основного потока – в нем происходит отрисовка UI, вызываются все обработчики кликов и других событий, поэтому он должен работать бесперебойно. Обновление UI происходит каждые 16 ms (60 кадров в секунду), многие операции занимают больше времени, из-за чего использование этих операций в основном потоке приведет к задержкам и фризам приложения.

Класс AsyncTask предлагает простой и удобный механизм для перемещения трудоёмких операций в фоновый поток. Благодаря ему вы получаете удобство синхронизации обработчиков событий с графическим потоком, что позволяет обновлять элементы пользовательского интерфейса для отчета о ходе выполнения задачи или для вывода результатов, когда задача завершена. У AsyncTask есть несколько основных методов. Обязательным является метод doInBackground(), остальные используются исходя из логики приложения.

* doInBackground() – основной метод, который выполняется в новом потоке. Не имеет доступа к UI. Именно в этом методе должен находиться код для тяжёлых задач. Принимает набор параметров тех типов, которые определены в реализации вашего класса. Этот метод выполняется в фоновом потоке, поэтому в нем не должно быть никакого взаимодействия с элементами пользовательского интерфейса. Размещайте здесь трудоёмкий код, используя метод publishProgress(), который позволит обработчику onProgressUpdate() передавать изменения в пользовательский интерфейс. Когда фоновая задача завершена, данный метод возвращает конечный результат для обработчика onPostExecute(), который сообщит о нём в поток пользовательского интерфейса.
* onPreExecute() – выполняется перед doInBackground(). Имеет доступ к UI
* onPostExecute() – выполняется после doInBackground() (может не вызываться, если AsyncTask был отменен). Имеет доступ к UI. Используйте его для обновления пользовательского интерфейса, как только ваша фоновая задача завершена. Данный обработчик при вызове синхронизируется с потоком GUI, поэтому внутри него вы можете безопасно изменять элементы пользовательского интерфейса.
* onProgressUpdate(). Имеет доступ к UI. Переопределите этот обработчик для публикации промежуточных обновлений в пользовательский интерфейс. При вызове он синхронизируется с потоком GUI, поэтому в нём вы можете безопасно изменять элементы пользовательского интерфейса.
* publishProgress() - можно вызвать в doInBackground() для показа промежуточных результатов в onProgressUpdate()
* cancel() - отмена задачи
* onCancelled() - Имеет доступ к UI. Задача была отменена. Имеются две перегруженные версии.

Корутины – инструмент Котлина, который преобразует асинхронные вызовы в последовательный код. Вызовы используются для выполнения длительных операций без блокирования основного потока, операция запускается в background потоке, а когда операция завершается, вызовы используются для информирования о результатах выполнения в основном потоке. Однако у такого подхода есть несколько недостатков – код становится тяжелым для чтения и понимания, также вызовы не позволяют использовать различные инструменты языка, как, например, исключения. Корутины позволяют переписать вызовы в последовательный код: его будет легче читать и будет возможность использовать все инструменты языка.

**ЗАДАЧА №1. АЛЬТЕРНАТИВНОЕ РЕШЕНИЕ ЗАДАЧИ “НЕ СЕКУНДОМЕР”**

Исходный код для следующих заданий представлен по ссылке:

<https://github.com/ADsty/labs-android_labs/tree/master/lab6>

Для данного и следующих заданий было разработано приложение, в котором будут представлены 3 решения для “не секундомера” и 3 решения для скачивания картинки. Перемещение между “не секундомером” и скачиванием картинки будет организовано с помощью bottomNavigation, выбор одного из 3 решений задачи – с помощью кнопок.

Для задания с использованием JavaThread был использован код из лабораторной работы №2. Поток работает до тех пор, пока флаг stopped равен false, данный флаг изменится, когда фрагмент пропадет с экрана (метод onPause). У виджета text\_seconds\_elapsed вызывается метод post, все что будет написано в нем, будет исполняться в UI потоке, таким образом мы передаем из background потока в UI поток значение lastNumber. Для увеличения точности вычислений была использована функция System.nanotime: при очередной итерации внешнего цикла while записывается текущее время в наносекундах, после чего вызывается внутренний пустой цикл while с условием выхода, если при очередном вызове функции System.nanotime пройдет 1 или более секунд, как только это условие будет выполнено, будет вызван метод post у text\_seconds\_elapsed.

Листинг 1 – код для секундомера

rivate var timer = Thread**{** while (!stopped) {  
 val time = System.nanoTime()  
 while(System.nanoTime() - time < 1000000000);  
 lastNumber++  
 if(!stopped)  
 text\_seconds\_elapsed.post **{** text\_seconds\_elapsed.*text* = "Seconds elapsed: " + lastNumber  
 **}** }  
**}**

Поток засыпает на 1 секунду, после чего текст в виджете меняется на новый, с увеличенным на единицу значением секунд. В методе onResume поток будет начат методом start. Чтобы гарантировать, что поток не будет тратить ресурсы устройства до этого метода, переменная stopped, которая является условием выхода из цикла потока, будет равна true вплоть до вызова метода onResume.

Листинг 2 – метод onResume

override fun onResume() {  
 super.onResume()  
 if (preferences.contains("lastNumber"))  
 lastNumber = preferences.getInt("lastNumber", 0)  
 stopped = false  
 timer.start()  
}

В ситуации, когда секундомер по какой-то причине пропал с экрана (метод onPause), флаг stopped меняет значение на true, sharedPreferences сохраняет текущее значение секундомера, после чего с помощью метода join мы дожидаемся завершения timer и проверяем, что он действительно был завершен и более не существует. В следующий раз, когда секундомер появится на экране (метод onResume), timer будет снова создан и запущен, данные последнего запуска секундомера будут получены из sharedPreferences.

Листинг 3 – метод onPause

override fun onPause() {  
 super.onPause()  
 preferences.edit().putInt("lastNumber", lastNumber).apply()  
 *println*("Threads count is " + Thread.activeCount())  
 stopped = true  
 timer.join()  
 if(!timer.*isAlive*){  
 *println*("Timer was killed")  
 *println*("Threads count is " + Thread.activeCount())  
 }  
}

Запустим программу и проверим, что счетчик работает (Рис. 1)

Изображение выглядит как текст

Автоматически созданное описание

Рис. 1 Результат запуска секундомера с Java Thread

Приложение работает идентично тому, как оно работало в лабораторной работе № 2. Для того, чтобы проверить, что при повороте экрана не создается дополнительных потоков для timer, повернем экран и посмотрим, что будет выведено на консоль – сначала будет выведено число потоков до завершения timer, потом уведомление о завершении timer, потом число потоков после завершения timer (Рис. 2).

Изображение выглядит как текст

Автоматически созданное описание

Рис. 2 Результат поворота экрана

Как можно видеть, число активных потоков сократилось на 1, что означает, что timer был завершен. Проверим еще раз и убедимся, что число потоков не меняется (Рис. 3).

Изображение выглядит как текст

Автоматически созданное описание

Рис. 3 Результат второго поворота экрана

Посмотрим, что происходит в профайлере при повороте экрана и выходе из activity. При повороте экрана поток прошлого activity останавливается и переходит в состояние dead, а затем с небольшой задержкой начинает работу новый поток, который и будет взаимодействовать с UI (Рис. 4).

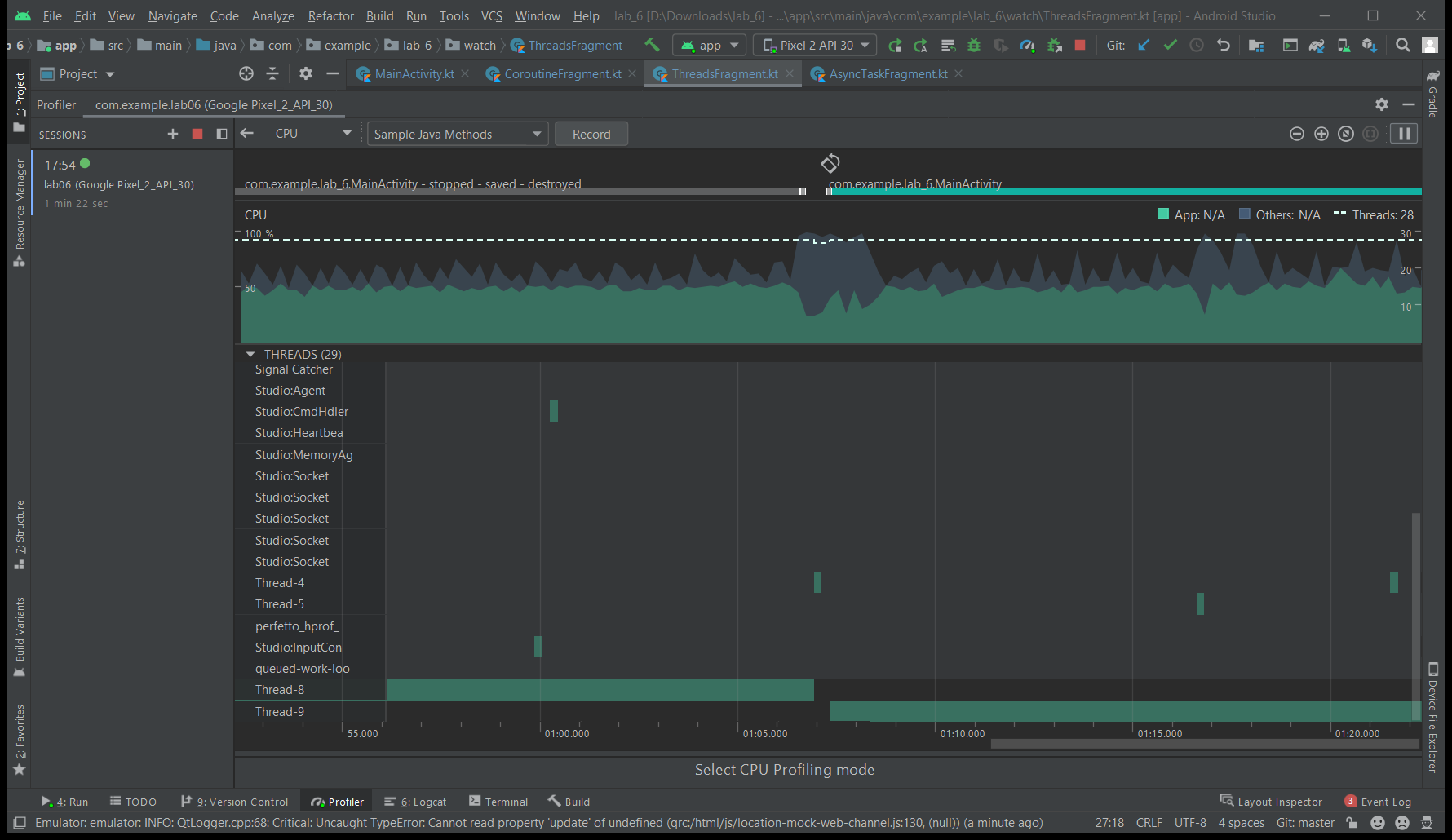


Рис. 4 Момент поворота экрана

При выходе из activity никаких новых потоков не создается, а прошлый поток, в данном случае Thread-9 завершится и перейдет в состояние dead (Рис. 5).

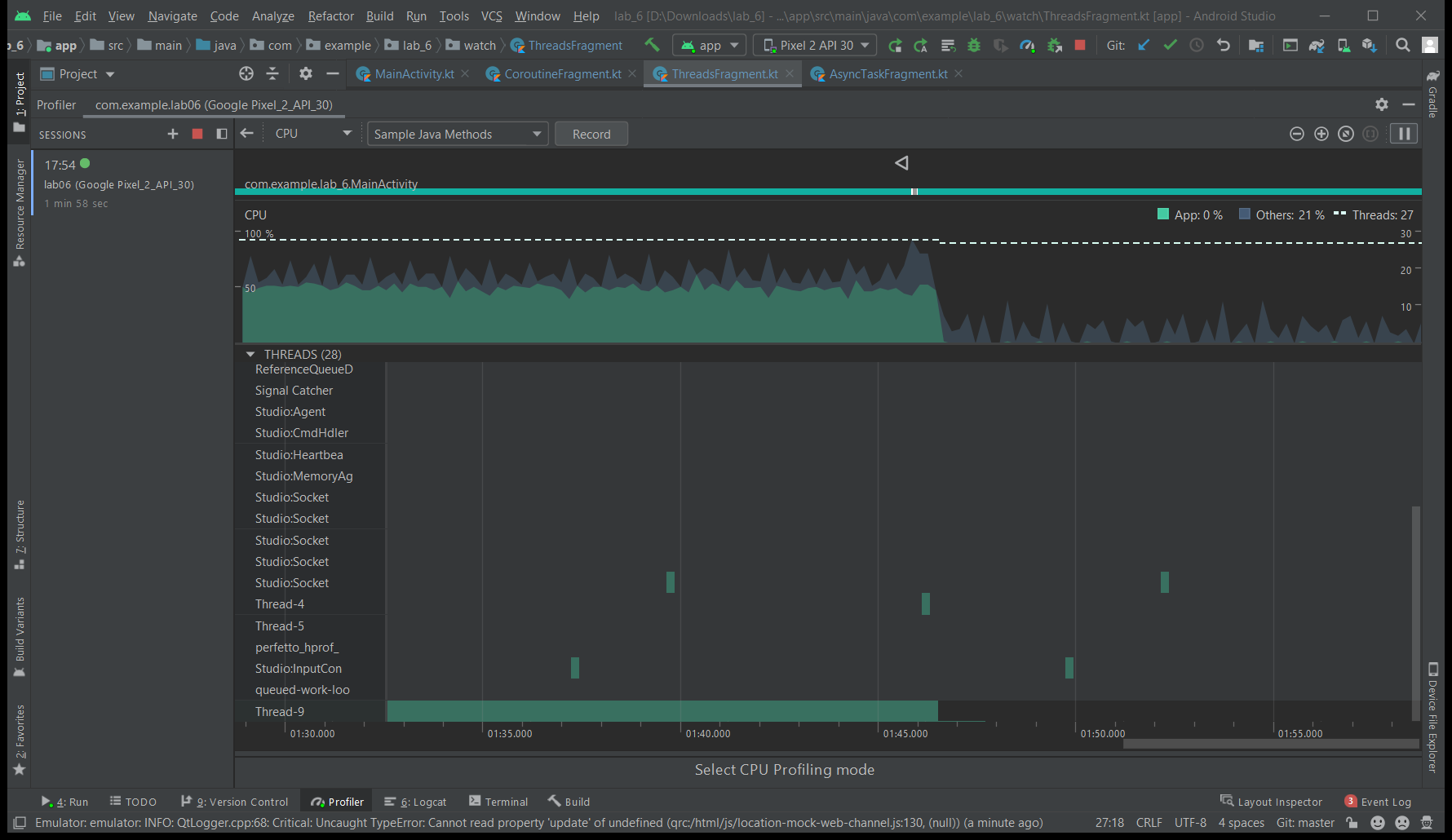


Рис. 5 Момент завершения activity для счетчика

Для решения задачи с помощью AsyncTask был создан класс TimerAsyncTask, который наследуется от AsyncTask и имеет внутри себя 2 метода – doInBackground и onProgressUpdate.

Метод doInBackground запускает цикл while, условием выхода будет вызов метода cancel внутри приложения, в таком случае флаг isCanceled будет равен true и из цикла мы выйдем. Внутри цикла вызывается функция currentTimeMillis, после чего прибавляем единицу к текущему значению lastNumber и вызывается функция publishProgress, из которой мы попадаем в функцию onProgressUpdate. В ней мы уже можем работать с UI, поэтому внутри мы обновляем текст, увеличивая значение секунд на 1.

Листинг 4 – класс TimerAsyncTask

private inner class TimerAsyncTask internal constructor() : AsyncTask<Void, Void, Void>() {  
  
 override fun doInBackground(vararg params: Void?): Void? {  
 while (!*isCancelled*) {  
 val time = System.currentTimeMillis()  
 while(System.currentTimeMillis() - time < 1000);

lastNumber++  
 publishProgress()  
 }  
 *println*("i finished")  
 return null  
 }  
  
 override fun onProgressUpdate(vararg values: Void?) {  
 if(!*isCancelled*) {  
 super.onProgressUpdate(\*values)  
 text\_seconds\_elapsed.*text* = "Seconds elapsed: " + lastNumber++  
 }  
 }  
}

Запускаем AsyncTask в методе onResume с помощью метода execute, в нем же, как и раньше мы берем из SharedPreferences значение lastNumber.

Листинг 5 – метод onResume

override fun onResume() {  
 lastNumber = preferences.getInt("lastNumber", 0)  
 timerAsyncTask = TimerAsyncTask()  
 timerAsyncTask.execute()  
 super.onResume()  
}

Завершаем работу AsyncTask в методе onPause с помощью вызова метода cancel. После этого вызова флаг isCancelled перейдет в состояние true и приложение выйдет из цикла в методе doInBackground, AsyncTask перестанет обновлять UI приложения.

Для проверки наших утверждений запустим приложение и повернем экран, затем завершим activity. Сначала проверим, что будет на консоли (Рис. 6).



Рис. 6 Результат поворота экрана

Как можем видеть, после вызова метода cancel цикл в методе doInBackground действительно был завершен, было напечатано сообщение в консоль и возвращен null. При завершении activity происходит аналогичная ситуация – на консоль выводится сообщение “I finished”, что говорит о том, что AsyncTask завершил свою работу. Теперь обратимся к профайлеру, и посмотрим, что происходит при повороте экрана (Рис. 7).

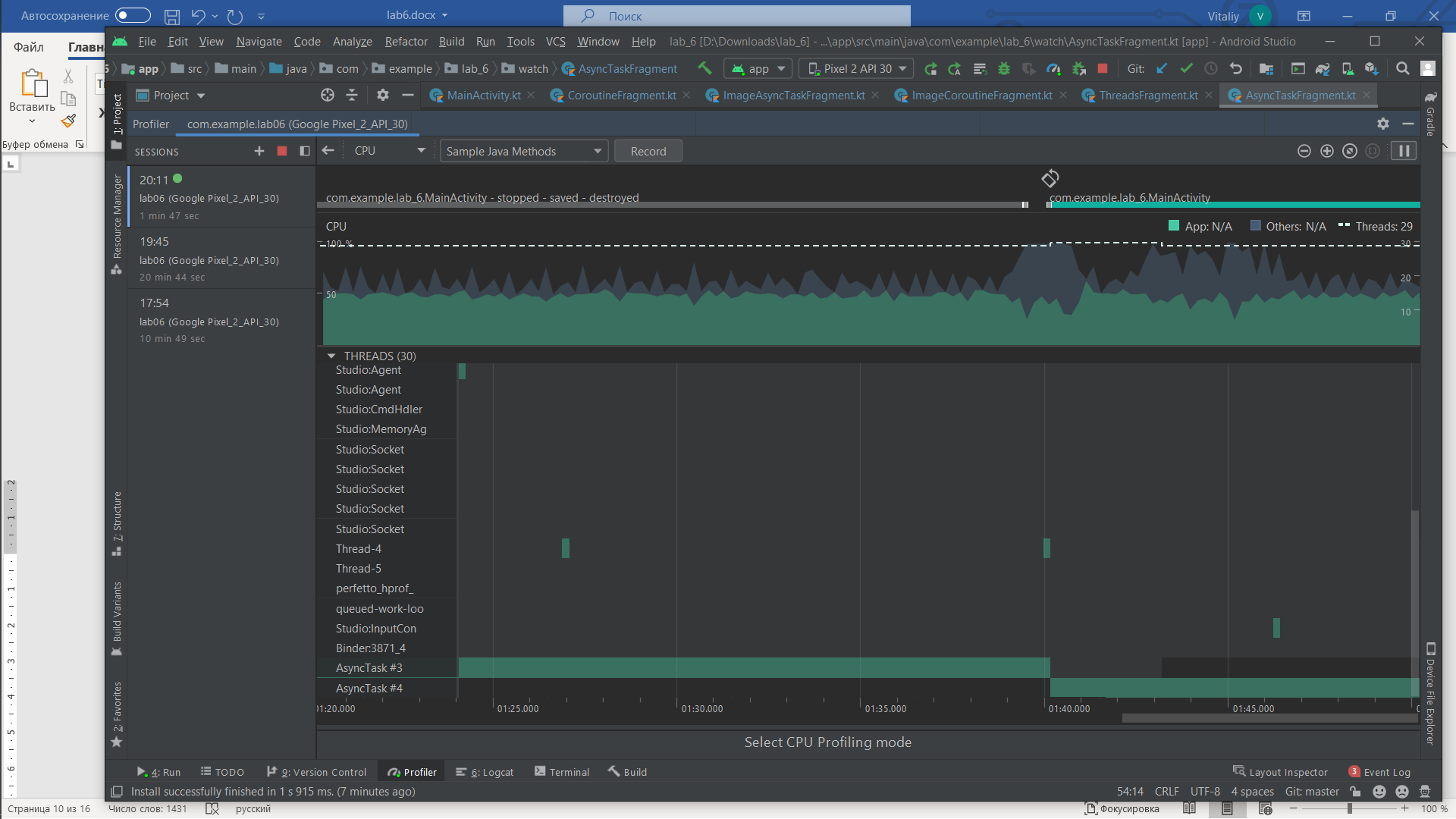


Рис. 7 Результат поворота экрана

Как можно видеть, при повороте экрана старый AsyncTask завершает работу, некоторое время находится в состоянии Sleeping и затем завершается, а для работы с activity создается новый AsyncTask. Однако возможна и другая ситуация (Рис. 8).

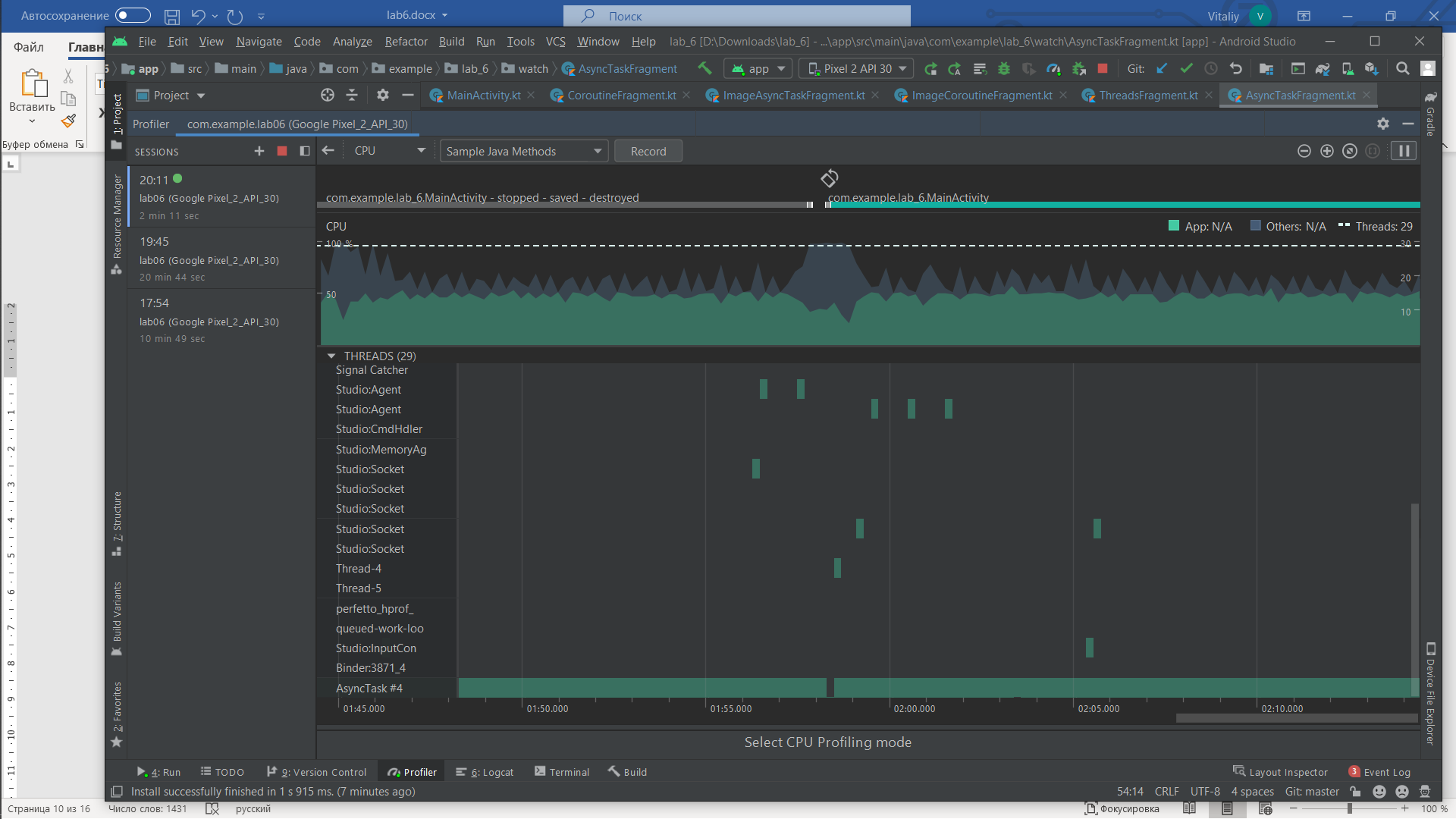


Рис. 8 Быстрый поворот экрана

При повторном повороте возникла следующая ситуация – AsyncTask завершил работу и перешел в состояние Sleeping, однако в новой activity был вызван слишком быстро, из-за чего новый AsyncTask не был создан (может быть создан только 1 экземпляр), а был переиспользован старый, в нем продолжится работа с UI. При завершении activity AsyncTask переходит в состояние Sleeping (Рис. 9).



Рис. 9 Завершение работы activity

Для решения с помощью корутин было сделано следующее: была объявлена переменная типа Job, в методе onResume инициализируем эту переменную с помощью lifecycle.coroutineScope, с помощью метода launchWhenResumed запускаем корутину, в методе onPause с помощью метода cancel завершаем корутину.

Листинг 6 – метод onResume

override fun onResume() {  
 lastNumber = preferences.getInt("lastNumber", 0)  
 job = *lifecycle*.*coroutineScope*.launchWhenResumed **{** while (*isActive*) {  
 withContext(Dispatchers.Default)**{** val time = System.nanoTime()  
 while(System.nanoTime() - time < 1000000000);  
 lastNumber++  
 *println*(lastNumber)  
 **}** text\_seconds\_elapsed.*text* = "Seconds elapsed: $lastNumber"  
 }  
 **}**.*also* **{  
 it**.invokeOnCompletion **{** cause**->** if(cause is CancellationException)  
 *println*("I'm done")  
 **}  
 }** super.onResume()  
}

В самой корутине на данный момент мы находимся в UI потоке, перейдем в background поток вызовом withContext с диспетчером Default, который используется для корутин с большим количеством вычислений. В нем методом System.nanoTime ждем 1 секунду, после чего прибавляем к значению lastNumber единицу и выводим на терминал текущее значение lastNumber для проверки того, что у нас не создалось 2-х и более корутин. В UI потоке мы обновляем текст в TextView. После завершения корутины мы проверяем причину завершения корутины и выводим сообщение на экран. Завершается корутина в одном из двух случаев – либо после вызова метода cancel, либо после вызова метода onDestroy, поскольку launchWhenResumed саспендит корутину при вызове этого метода. После запуска программы посмотрим на профайлер – он показывает, что Default Dispatcher непрерывно работает (Рис.10).



Рис. 10 Работа Default Dispatcher

После закрытия activity работа диспетчера прекращается (Рис. 11).

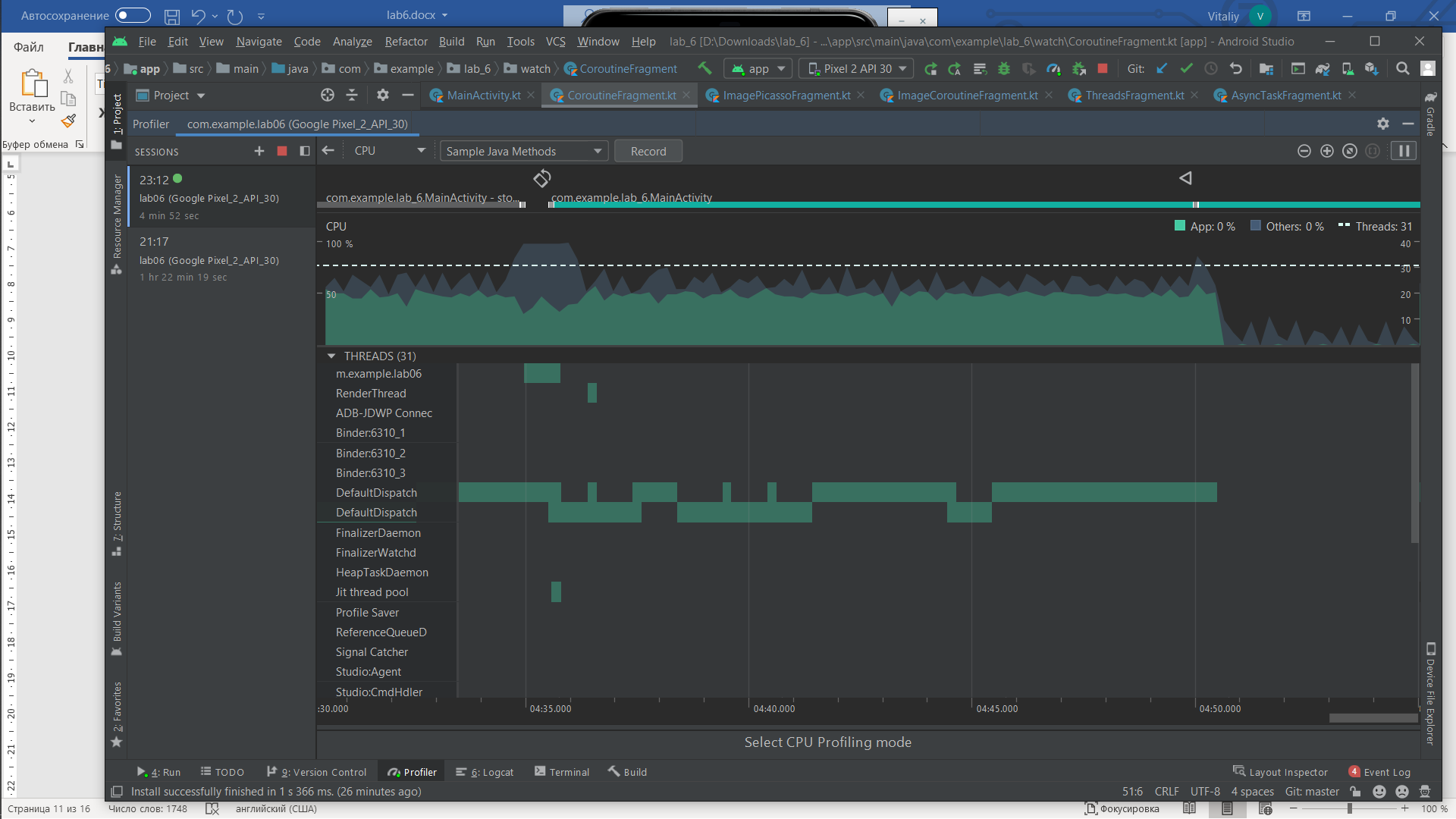


Рис. 11 Завершение работы activity

При повороте экрана изменения в профайлере совсем незаметные, однако в консоли по завершению корутины должно появиться сообщение, его и будем проверять при повороте экрана (Рис. 12).

Изображение выглядит как стол

Автоматически созданное описание

Рис. 12 Вывод консоли при повороте экрана

Как можно видеть, при повороте экрана корутина была завершена, а значение lastNumber сохранено, новая корутина взяла сохраненное значение и продолжила считать с него. Чтобы убедиться, что старая корутина останавливается с концами, закроем activity (Рис. 13).

Изображение выглядит как текст

Автоматически созданное описание

Рис. 13 Результат на консоли после закрытия activity

После закрытия activity было выдано последнее сообщение о завершении корутины и выдача результата на консоль завершена. Это говорит о том, что предыдущая корутина больше не исполняется.

**ЗАДАЧА №2. ЗАГРУЗКА КАРТИНКИ В ФОНОВОМ ПОТОКЕ (ASYNCTASK)**

Для загрузки картинки с помощью AsyncTask был создан класс DownloadImage, наследующийся от AsyncTask. В нем в методе doInBackground по передаваемой URL ссылке скачивается и декодируется картинка. В методе onPostExecute проверяется, скачалась ли картинка. Если она не скачалась, то пользователю выдается сообщение о неудаче, в любом случае картинка устанавливается в передаваемый в качестве параметра imageView (при неудаче будет пустой экран), а используемый при загрузке ProgressBar становится невидимым.

Листинг 7 – класс DownloadImage

private inner class DownloadImage(private var imageView: ImageView):  
 AsyncTask<String, Void, Bitmap>() {  
  
 override fun doInBackground(vararg urls: String): Bitmap? {  
 val currentUrl = urls[0]  
  
 var icon: Bitmap? = null  
  
 try {  
 val inputStream = java.net.URL(currentUrl).openStream()  
 icon = BitmapFactory.decodeStream(inputStream)  
 } catch (e: Exception) {  
 *println*("Error! Can't download file!")  
 }  
 return icon  
 }  
  
 override fun onPostExecute(result: Bitmap?) {  
 if (result == null){  
 Toast.makeText(*context*, "Can't Download", Toast.*LENGTH\_SHORT*).show()  
 }  
 imageView.setImageBitmap(result)  
 load\_bar.*visibility* = View.*INVISIBLE* }  
}

В методе onStart запускается AsyncTask, в методе onStop он завершается методом cancel. В результате с правильной ссылкой на экране отображается нужное изображение (Рис. 14).

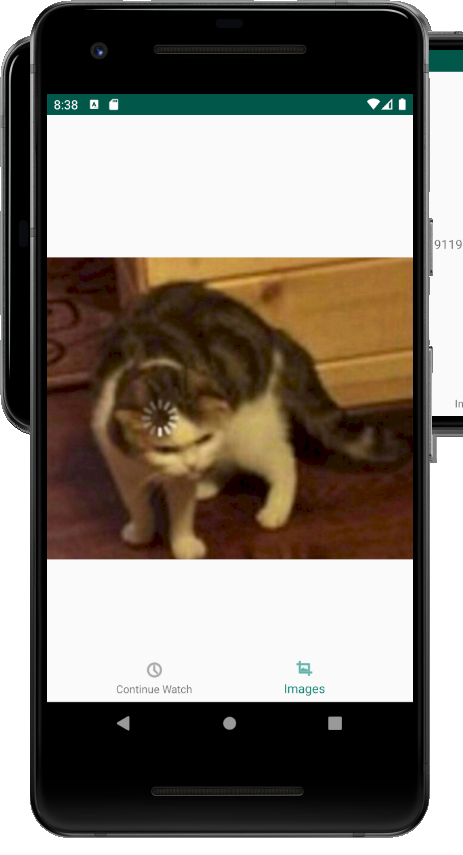


Рис. 14 Удачное скачивание картинки

При неудаче (например, неправильная ссылка) появляется сообщение на экране, пропадает ProgressBar, однако картинка не появляется (Рис. 15).

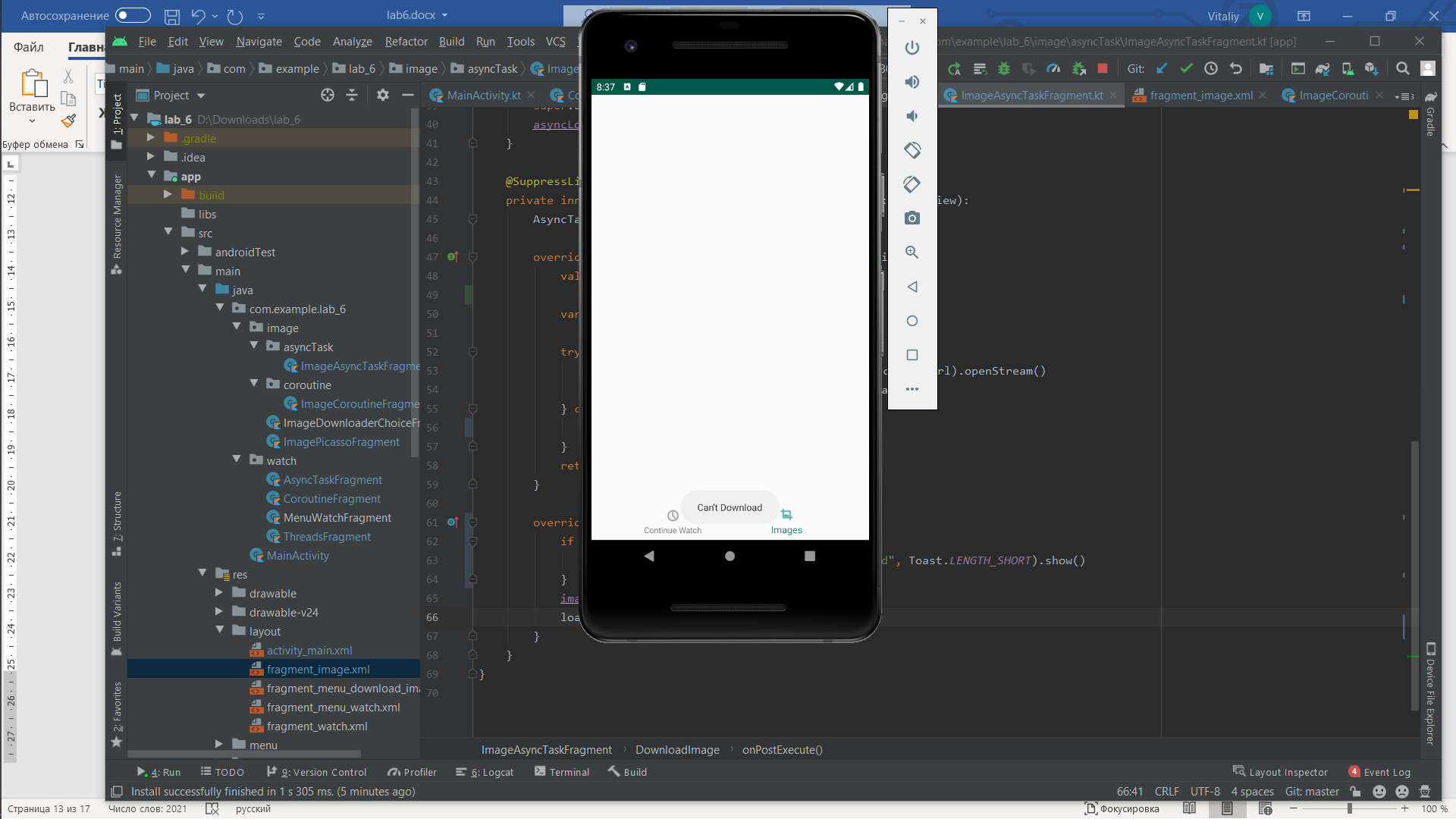


Рис. 15 Неудачное скачивание картинки

**ЗАДАЧА №3. ЗАГРУЗКА КАРТИНКИ В ФОНОВОМ ПОТОКЕ (KOTLIN COROUTINES)**

Для написания задания с помощью корутин была написана функция downloadImage, в ней объявлена корутина job как lifecycle.coroutineScope.launch, в ней мы пытаемся скачать картинку в другой корутине async (диспетчер Dispatchers.IO для передачи данных по сети) с вызываемым методом await, что значит что весь остальной код будет ждать, пока код в async выполнится. После выполнения async картинка ставится в imageView, если что-то пошло не так и картинка равна null, то уведомляем пользователя что картинку скачать невозможно.

Листинг 8 – метод downloadImage

private fun downloadImage() {  
  
 var icon: Bitmap?  
  
 job = *lifecycle*.*coroutineScope*.*launch***{** icon = *async*(Dispatchers.IO) **{** try {  
 URL(imageUrl).openStream().*use* **{** return@async BitmapFactory.decodeStream(**it**)  
 **}** }catch (e: Exception){  
 return@async null  
 }  
 **}**.await()  
 load\_bar.*visibility* = View.*INVISIBLE* if(icon == null){  
 Toast.makeText(*context*, "Can't Download", Toast.*LENGTH\_SHORT*).show()  
 }  
 image.setImageBitmap(icon)  
 **}**}

Завершается корутина либо по завершению попытки скачать картинку, либо после вызова метода cancel в методе onPause. Поведение программы идентично предыдущему. При неправильной ссылке выскакивает уведомление и остается белый экран (Рис. 16).

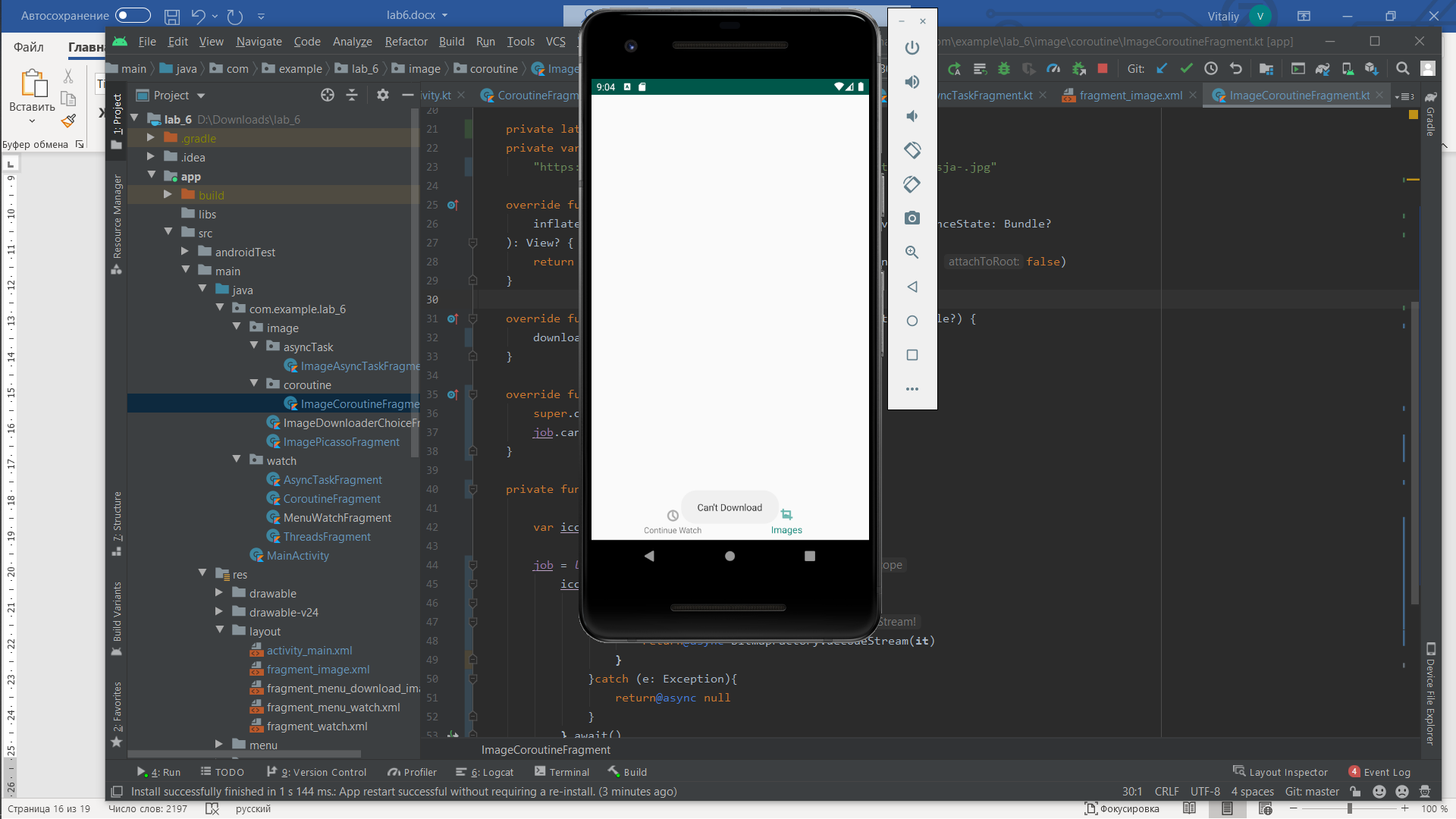


Рис. 16 Неудачная попытка скачивания

При удачной попытке картинка появляется на экране (Рис. 17).

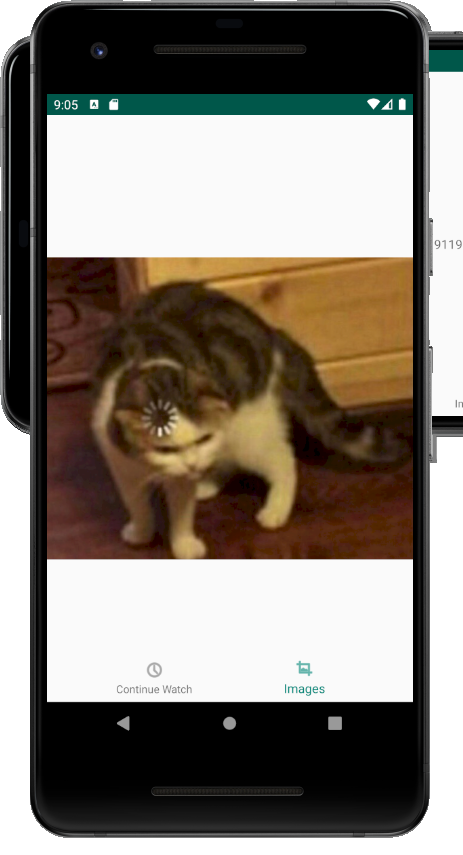


Рис. 17 Удачная попытка скачивания

**ЗАДАЧА № 4. ИСПОЛЬЗОВАНИЕ СТОРОННИХ БИБЛИОТЕК**

В качестве последнего решения используем библиотеку Picasso. С ее использованием достаточно лишь одной строчки Picasso.get().load(url).into, однако нужно добавить обработчики для успешного скачивания и провального, чтобы как и в прошлых случаях уведомить пользователя о том, что что то пошло не так, либо же чтобы убрать ProgressBar.

Листинг 9 – функция для использования библиотеки Picasso

private fun loadImage(url: String) {  
 Picasso.get().load(url).into(image, object : Callback {  
  
 override fun onSuccess() {  
 load\_bar.*visibility* = View.*INVISIBLE* }  
  
 override fun onError(e: Exception) {  
 load\_bar.*visibility* = View.*INVISIBLE* Toast.makeText(*context*, "Can't Download", Toast.*LENGTH\_SHORT*).show()  
 }  
 })  
}

Таким образом мы загружаем в imageView по id “image” картинку, URL которой передаем в качестве аргумента метода. Результат абсолютно идентичен предыдущим двум.

**ЗАКЛЮЧЕНИЕ**

В данной работе были рассмотрены различные способы организации многопоточных решений задач с использованием Java Threads, AsyncTask, Kotlin Coroutines. Для каждого из способов решения было определено: когда запускается поток, как мы передаем данные из background в UI, когда завершается поток. Также были опробованы различные способы скачивания картинок из интернета – поскольку AsyncTask является устаревшим, то оптимальным способом решения является использование Kotlin Coroutines, либо же использование сторонних библиотек, как например библиотека Picasso.

Исходный код для лабораторных работ находится на [github](https://github.com/ADsty/labs-android_labs) (<https://github.com/ADsty/labs-android_labs> )

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

<https://developer.android.com/kotlin/coroutines>

<https://developer.android.com/reference/android/os/AsyncTask.html>

<https://developer.android.com/reference/java/lang/Thread>

<https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md>

<https://developer.android.com/codelabs/kotlin-coroutines>