Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Лабораторная работа № 6

Дисциплина: Проектирование мобильных приложений

Тема: Многопоточные Android приложения

Выполнил студент гр. 3530901/90201 _____С.А. Федоров
(подпись)

Принял старший преподаватель _____А.Н. Кузнецов
(подпись)

“____” _____ 2021 г.

Санкт-Петербург
2021

Оглавление

Цели

- Получить практические навыки разработки многопоточных приложений:
    - Организация обработки длительных операций в background (worked) thread
        - Запуск фоновой операции (Coroutine/ExecutionService/Thread)
        - Остановка фоновой операции (Coroutine/ExecutionService/Thread)
    - Публикация данных из background (worked) thread в main (ui) thread
- Освоить 3 основные группы API для разработки многопоточных приложений:
    - Kotlin Coroutines
    - ExecutionService
    - Java Threads

Задачи

1. Разработайте несколько альтернативных приложений “не секундомер”, отличающихся друг от друга организацией многопоточной работы. Опишите все известные Вам решения.
2. Создайте приложение, которое скачивает картинку из интернета и размещает ее в ImageView в Activity. Используйте ExecutionService для решения этой задачи
3. Перепишите предыдущее приложение с использованием Kotlin Coroutines.
4. Скачать изображение при помощи одной из библиотек: Glide, picasso или fresco.

Задача 1а. Приложение НЕ секундомер при помощи Thread.

В данном задании требуется решить задачу “НЕ секундомер”, представленную в lab2 при помощи Java Threads.

Когда приложение запускается, то начинает выполняться main поток, от которого порождаются дочерние. Завершение главного потока означает завершение выполнения программы. Несмотря на то, что главный поток создается автоматически, им можно управлять через объект класса Thread. Для этого нужно вызвать метод currentThread(), после чего можно управлять потоком.

При создании потока требуется воспользоваться конструктором, чтобы проинициализировать созданный объект.

Потоки можно “контролировать” при помощи следующих методов:

- Start() – для запуска потока
- Interrupt() – для прерывания текущего потока
- isInterruted – булева функция для проверки того, прерван поток или нет

Листинг 1. Реализация при помощи Thread

```
private fun createNewThread() = Thread {
    try {
        while (!Thread.currentThread().isInterrupted) {
            Log.d(TAG, "${Thread.currentThread()} is iterating")
            Thread.sleep(1000)
            textSecondsElapsed.post {
                textSecondsElapsed.text = "${secondsElapsed++}"
            }
        }
    } catch (e: InterruptedException) {
        Thread.currentThread().interrupt()
    }
}
override fun onStop() {
    super.onStop()
    backgroundThread.interrupt()
}
override fun onStart() {
    Log.d(TAG, "onStart()")
    super.onStart()
    backgroundThread = createNewThread()
    backgroundThread.start()
    Log.d(TAG, "${backgroundThread.id}")
}
```

Для передачи данных в UI Thread использовался метод post. Отличие данного кода, от реализации в lab2 заключается в том, что:

- В цикле while теперь условие “Пока текущий поток не прерван”
- Добавлено логирование, чтобы было видно какой поток сейчас запущен
- Добавлена конструкция try catch, чтобы отслеживать ситуации, когда мы будем пытаться выполнять действия в прерванном потоке.
- В блоке catch мы будем завершать поток. Это сделано для того, чтобы не возникло ситуации, когда запущено сразу несколько потоков.

Теперь посмотрим на результаты работы программы

```
2021-11-26 20:59:05.614 24515-24530/com.example.threads D/ContinueWatch: Thread[Thread-2,5,main] is iterating
2021-11-26 20:59:06.615 24515-24530/com.example.threads D/ContinueWatch: Thread[Thread-2,5,main] is iterating
2021-11-26 20:59:07.368 24515-24515/com.example.threads D/ContinueWatch: onPause()
2021-11-26 20:59:07.368 24515-24515/com.example.threads D/ContinueWatch: Saving SEC=2
2021-11-26 20:59:07.376 24515-24515/com.example.threads D/ContinueWatch: onDestroy()
2021-11-26 20:59:07.400 24515-24515/com.example.threads D/ContinueWatch: Restore SEC=2
2021-11-26 20:59:07.400 24515-24547/com.example.threads D/ContinueWatch: Thread[Thread-3,5,main] is iterating
2021-11-26 20:59:07.401 24515-24515/com.example.threads D/ContinueWatch: onResume()
2021-11-26 20:59:08.401 24515-24547/com.example.threads D/ContinueWatch: Thread[Thread-3,5,main] is iterating
2021-11-26 20:59:09.402 24515-24547/com.example.threads D/ContinueWatch: Thread[Thread-3,5,main] is iterating
2021-11-26 20:59:09.797 24515-24515/com.example.threads D/ContinueWatch: onPause()
2021-11-26 20:59:09.798 24515-24515/com.example.threads D/ContinueWatch: Saving SEC=4
2021-11-26 20:59:09.824 24515-24515/com.example.threads D/ContinueWatch: onDestroy()
2021-11-26 20:59:09.910 24515-24515/com.example.threads D/ContinueWatch: Restore SEC=4
2021-11-26 20:59:09.911 24515-24552/com.example.threads D/ContinueWatch: Thread[Thread-4,5,main] is iterating
2021-11-26 20:59:09.912 24515-24515/com.example.threads D/ContinueWatch: onResume()
2021-11-26 20:59:10.913 24515-24552/com.example.threads D/ContinueWatch: Thread[Thread-4,5,main] is iterating
2021-11-26 20:59:11.415 24515-24515/com.example.threads D/ContinueWatch: onPause()
2021-11-26 20:59:11.485 24515-24515/com.example.threads D/ContinueWatch: Saving SEC=5
2021-11-26 20:59:14.078 24515-24515/com.example.threads D/ContinueWatch: onResume()
2021-11-26 20:59:14.079 24515-24554/com.example.threads D/ContinueWatch: Thread[Thread-4,5,main] is iterating
2021-11-26 20:59:15.081 24515-24554/com.example.threads D/ContinueWatch: Thread[Thread-4,5,main] is iterating
```

Рис.1 Результаты логирования при помощи Thread

Исходя из Рис.1 можно сделать вывод, что все предыдущие потоки завершаются при разрушении Activity (5 и 13 строчки) и в любой момент времени работает только один поток.

Задача 1b. НЕ секундомер при помощи ExecutorService.

По заданию требуется решить задачу “НЕ секундомер”, представленную в lab2 при помощи ExecutorService.

Executor—простой интерфейс, содержащий метод execute() для запуска задачи, заданной запускаемым объектом Runnable. ExecutorService представляет собой суб-интерфейс Executor, который добавляет функциональность для управления жизненным циклом потоков. Например, метод shutdown() – для прекращения работы.

ExecutorService – это интерфейс для пула потоков, который позволяет запускать параллельное выполнение задач.

Для реализации задачи при помощи ExecutorService потребуется:

- В методе OnStart создать новый ExecutorService, где будем каждую секунду обновлять значение, пока сервис не будет остановлен
- В методе onStop используем shutdown для прекращения работы. То есть остановим передачу новых задач в ExecutorService, а также останавливаем все потоки

Листинг 2. Реализация при помощи ExecutorService

```
private lateinit var future: Future<*>

override fun onStart() {
    val executor = (applicationContext as MainApplication).executorService
    future = executor.submit {
        while (!executor.isShutdown) {
            Log.d(TAG, "${Thread.currentThread()} is iterating")
            Thread.sleep(1000)
            textSecondsElapsed.post {
                textSecondsElapsed.text = "${secondsElapsed++}"
            }
        }
    }
    super.onStart()
}

override fun onStop() {
    future.cancel(true)
    super.onStop()
}
```

Теперь посмотрим на результаты работы программы

```
2021-11-26 22:18:31.764 27990-28005/com.example.executionservice D/ContinueWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-26 22:18:32.766 27990-28005/com.example.executionservice D/ContinueWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-26 22:18:33.768 27990-28005/com.example.executionservice D/ContinueWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-26 22:18:33.833 27990-27990/com.example.executionservice D/ContinueWatch: onPause()
2021-11-26 22:18:33.833 27990-27990/com.example.executionservice D/ContinueWatch: Saving SEC=3
2021-11-26 22:18:33.856 27990-27990/com.example.executionservice D/ContinueWatch: onDestroy()
2021-11-26 22:18:33.886 27990-27990/com.example.executionservice D/ContinueWatch: Restore SEC=3
2021-11-26 22:18:33.886 27990-28021/com.example.executionservice D/ContinueWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-26 22:18:33.887 27990-27990/com.example.executionservice D/ContinueWatch: onResume()
2021-11-26 22:18:34.887 27990-28021/com.example.executionservice D/ContinueWatch: Thread[pool-2-thread-1,5,main] is iterating
2021-11-26 22:18:35.352 27990-27990/com.example.executionservice D/ContinueWatch: onPause()
2021-11-26 22:18:35.352 27990-27990/com.example.executionservice D/ContinueWatch: Saving SEC=4
2021-11-26 22:18:35.357 27990-27990/com.example.executionservice D/ContinueWatch: onDestroy()
2021-11-26 22:18:35.385 27990-27990/com.example.executionservice D/ContinueWatch: Restore SEC=4
2021-11-26 22:18:35.386 27990-27990/com.example.executionservice D/ContinueWatch: onResume()
2021-11-26 22:18:35.386 27990-28022/com.example.executionservice D/ContinueWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-26 22:18:36.387 27990-28022/com.example.executionservice D/ContinueWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-26 22:18:37.281 27990-27990/com.example.executionservice D/ContinueWatch: onPause()
2021-11-26 22:18:37.350 27990-27990/com.example.executionservice D/ContinueWatch: Saving SEC=5
2021-11-26 22:18:38.976 27990-27990/com.example.executionservice D/ContinueWatch: onResume()
2021-11-26 22:18:38.977 27990-28023/com.example.executionservice D/ContinueWatch: Thread[pool-4-thread-1,5,main] is iterating
2021-11-26 22:18:39.978 27990-28023/com.example.executionservice D/ContinueWatch: Thread[pool-4-thread-1,5,main] is iterating
2021-11-26 22:18:40.979 27990-28023/com.example.executionservice D/ContinueWatch: Thread[pool-4-thread-1,5,main] is iterating
```

Рис.2 Результаты логирования при помощи ExecutorService

Исходя из Рис.2 можно сделать вывод, что при создании нового ExecutorService у нас создается новый Thread Pool. То есть основной поток (thread) никогда не завершается (кроме случаев с завершением работы приложения), а он просто ждет выполнение другой задачи. Получается, что ExecutorService по мере необходимости создает пул потоков.

Также ExecutorService будет повторно использовать ранее созданные Pool, когда они будут доступны. Для этого подождём некоторое время и посмотрим на результаты логирования.

```
2021-11-26 22:30:40.919 28433-28469/com.example.executionservice D/ContinueWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-26 22:30:41.920 28433-28469/com.example.executionservice D/ContinueWatch: Thread[pool-3-thread-1,5,main] is iterating
2021-11-26 22:51:07.630 28433-29023/com.example.executionservice D/ContinueWatch: Thread[pool-1-thread-1,5,main] is iterating
2021-11-26 22:51:08.631 28433-29023/com.example.executionservice D/ContinueWatch: Thread[pool-1-thread-1,5,main] is iterating
```

Рис.3 Появление ранее использовавшегося pool снова

Получается, что основное предназначение пулов – это повышение производительности программ, выполняющих множество краткосрочных асинхронных задач.

Задача 1c. НЕ секундомер при помощи Kotlin Coroutines

По заданию требуется решить задачу “НЕ секундомер”, представленную в lab2 при помощи Kotlin Coroutines.

Корутины — облегченные потоки, то есть те же самые потоки, но более “дешевые” с точки зрения производительности. Корутины, также, как и потоки могут работать параллельно, ждать друг друга и общаться. Основное отличие корутин от потоков – мы можем создать тысячи корутин и не сильно потерять в производительности, когда в случае создания тысячи потоков не все современные машины смогут справиться. Корутины и потоки являются многозадачными, но разница в том, что потоки управляются ОС, а Корутины пользователями.

Для реализации задачи при помощи Kotlin Coroutines потребуется:

- Подключить соответствующую библиотеку
- Использовать lifecycleScope, чтобы связать жизненный цикл Activity/Fragment с ЖЦ coroutine. То есть, новая coroutine будет запущена тогда, когда будет определенная стадия ЖЦ нашего Activity, а также coroutine будет отменена, когда Activity/Fragment будет уничтожен.
- Далее требуется выбрать эту стадию при помощи установки у lifecycleScope метода launchWhenResumed. Я выбрал именно RESUMED, так как при звонке на телефон приложение переходит в состояние RESUMED и не отображается на экране, поэтому не стоит считать время в этот период. Также стоит помнить, что launch возвращает ссылку на job, чтобы была возможность дальнейшего управления этой coroutine.
- Теперь уже устанавливаем условие для цикла while, в котором у нас будет работать coroutine и обновляться значение секунд. Для этого выставим isActive, то есть, когда текущая coroutine все еще активна

Листинг 3. Реализация при помощи Kotlin Coroutines

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    textSecondsElapsed = findViewById(R.id.textSecondsElapsed)

    lifecycleScope.launchWhenResumed {
        Log.d(TAG, "Launch of Coroutine ")
        while (isActive) {
            Log.d(TAG, "Coroutine is working")
            delay(1000)
            textSecondsElapsed.text = "${secondsElapsed++}"

        }
    }
    Log.d(TAG, "Completion of Coroutine")
}
```

Теперь посмотрим на результаты работы программы

```
2021-11-27 00:45:14.455 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:15.457 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:15.526 2025-2025/com.example.coroutines D/ContinueWatch: onPause()
2021-11-27 00:45:15.526 2025-2025/com.example.coroutines D/ContinueWatch: Saving SEC=2
2021-11-27 00:45:15.576 2025-2025/com.example.coroutines D/ContinueWatch: onDestroy()
2021-11-27 00:45:15.602 2025-2025/com.example.coroutines D/ContinueWatch: Completion of Coroutine
2021-11-27 00:45:15.603 2025-2025/com.example.coroutines D/ContinueWatch: Restore SEC=2
2021-11-27 00:45:15.604 2025-2025/com.example.coroutines D/ContinueWatch: onResume()
2021-11-27 00:45:15.606 2025-2025/com.example.coroutines D/ContinueWatch: Launch of Coroutine
2021-11-27 00:45:15.606 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:16.607 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:17.609 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:18.478 2025-2025/com.example.coroutines D/ContinueWatch: onPause()
2021-11-27 00:45:18.478 2025-2025/com.example.coroutines D/ContinueWatch: Saving SEC=4
2021-11-27 00:45:18.499 2025-2025/com.example.coroutines D/ContinueWatch: onDestroy()
2021-11-27 00:45:18.548 2025-2025/com.example.coroutines D/ContinueWatch: Completion of Coroutine
2021-11-27 00:45:18.548 2025-2025/com.example.coroutines D/ContinueWatch: Restore SEC=4
2021-11-27 00:45:18.549 2025-2025/com.example.coroutines D/ContinueWatch: onResume()
2021-11-27 00:45:19.551 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:20.552 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:21.552 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:21.890 2025-2025/com.example.coroutines D/ContinueWatch: onPause()
2021-11-27 00:45:25.668 2025-2025/com.example.coroutines D/ContinueWatch: onResume()
2021-11-27 00:45:25.669 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:26.670 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
2021-11-27 00:45:27.671 2025-2025/com.example.coroutines D/ContinueWatch: Coroutine is working
```

Рис.3 Результаты логирования при помощи Kotlin Coroutines

Исходя из Рис.3 можно сделать вывод, что новая coroutine создается, когда Activity в состоянии RESUMED. Также coroutine отменяется при переходе Activity в DESTROYED.

Задача 2. Загрузка картинки в фоновом потоке

По заданию требуется скачать изображение из интернета и поместить его в ImageView при помощи ExecutorService.

Для того, чтобы загрузить картинку из интернета требуется создать класс ViewModel, в котором мы будем создавать поток для загрузки изображения.

Листинг 4. Файл ViewModelClass, решение при помощи ExecutorService

```
class ViewModelClass(application: Application) :
AndroidViewModel(application) {

    private val executor: ExecutorService =
getApplication<MainApplication>().executorService
    val mutableLiveData = MutableLiveData<Bitmap>()

    fun downloadImage(url: String) {
        executor.execute {
            Log.d(ContentValues.TAG, "Thread sleeping 2 sec")
            Thread.sleep(2000)
            val stream = URL(url).openConnection().getInputStream()
            val bitmap = BitmapFactory.decodeStream(stream)
            mutableLiveData.postValue(bitmap)
        }
    }
}
```

Теперь, когда поток создается можно перейти к MainActivity. Для того, чтобы картинка загружалась по нажатию кнопки и после загрузки находилась в ImageView, следует установить на Bitmap (объект, хранящий изображение) наблюдателя

Листинг 4. Файл MainActivity

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private val viewModel: ViewModelClass by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.btnSubmit.setOnClickListener {
            viewModel.downloadImage(my_image)
            Log.d(ContentValues.TAG, "Download Image")
        }

        viewModel.mutableLiveData.observe(this) {
```

```
            binding.imageView.setImageBitmap(it)
            Log.d(ContentValues.TAG, "Set Image")
        }

    }

    companion object {
        private const val my_image = "https://www.meme-
arsenal.com/memes/dea858c26303c031d221b415d75de83b.jpg"
    }

}
```

Посмотрим на результаты работы программы и логирование

```
2021-11-27 18:24:06.776 11735-11735/com.example.task2 D/ContentValues: Download Image
2021-11-27 18:24:06.779 11735-11764/com.example.task2 D/ContentValues: Thread sleeping 2 sec
2021-11-27 18:24:08.868 11735-11735/com.example.task2 D/ContentValues: Set Image
```

Рис.4 Результаты работы программы

Задача 3. Загрузка картинки в фоновом потоке (Kotlin Coroutines)

По заданию требуется решить предыдущую задачу по загрузке картинки при помощи Kotlin Coroutines.

Для этого потребуется внести изменения в ViewModelClass, в метод downloadImage.

Листинг 5. Файл ViewModelClass, решение при помощи Kotlin Coroutines

```
fun downloadImage(url: String) {
    viewModelScope.launch(Dispatchers.IO) {
        Log.d(TAG, "Sleeping 2 sec")
        delay(2000)
        val stream = URL(url).openConnection().getInputStream()
        val bitmap = BitmapFactory.decodeStream(stream)
        withContext(Dispatchers.Main) {
            mutableLiveData.value = bitmap
        }
    }
}
```

Задача 4. Использование сторонних библиотек

По заданию требуется решить предыдущую задачу по загрузке картинки при помощи одной из сторонних библиотек.

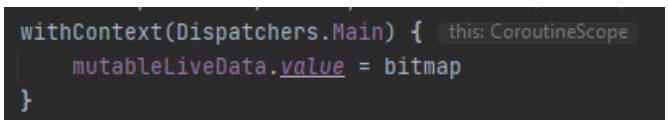
Для решения была выбрана библиотека Glide

Листинг 6. Файл MainActivity, решение при помощи библиотеки Glide

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.btnSubmit.setOnClickListener {
            Glide.with(this).load(my_image).into(binding.imageView)
            Log.d(ContentValues.TAG, "Download Image")
        }
    }

    companion object {
        private const val my_image = "https://www.meme-
arsenal.com/memes/dea858c26303c031d221b415d75de83b.jpg"
    }
}
```

Выводы

- В ходе выполнения данной лабораторной работы было выполнено знакомство с многопоточными Android приложениями. Для отправки значений в UI поток был применен метод post.
- В ходе выполнения Задачи 1 (a, b, c) были изучены основные способы выполнения кода не в UI потоке:
    - Использование Java Threads. Для запуска потока используется метод start(), а для останова interrupt(), и использовался метод post() для выполнения действий в UI потоке. Также в Java Thread требуется следить за потоком (то, что там происходит), так как можно получить InterruptedException
    - Использование ExecutorService, который является интерфейсом пула потоков и позволяет запускать параллельное выполнение задач. При создании нового ExecutorService будет создан новый Thread Pool, то есть, основной поток никогда не завершается (кроме случаев с завершением работы приложения), а он просто ждет выполнение другой задачи. Здесь для запуска задачи в фоне использовался метод submit(), а для прекращения – cancel(), передача данных также выполняется при помощи post().
    - Использование Kotlin Coroutines, которые являются теми же самыми потоками, но облегченными (при их использовании мы не будем терять в производительности). Корутины и потоки являются многозадачными, но разница в том, что потоки управляются ОС, а Корутины пользователями. Для запуска используем launch(), а для прекращения работы метод cancel(). Здесь для передачи можно также использовать метод post(), но также можно использовать переключение контекстов



- При помощи ExecutorService и Kotlin Coroutines была решена задача по загрузке изображения из интернета и установка его в ImageView.
- Также для загрузки изображения использовалась стандартная библиотека Gilde.

Ссылка на github: https://github.com/sergeyfedorov02/Android_labs.git

Список источников

https://developer.android.com/
https://github.com/andrei-kuznetsov/android-lectures

Время выполнения лабораторной работы

- Задача 1 – 260 минут
- Задача 2 – 80 минут
- Задача 3 – 30 минут
- Задача 4 – 30 минут
- Заполнение отчета – 400 минут (отчет заполнялся по мере решения задач, поэтому время решения задачи = отчет + написание кода)