**PRIR lab 13**

**Znajdź i opisz ciekawy problem i jego implementację programowania równoległego/rozproszonego, o którym nie mówiliśmy na zajęciach.**

Coroutine ułatwia pisanie asynchronicznego kodu, poprzez możliwość pisania tego kodu w sposób sekwencyjny. Sekwencyjny – czyli bez żadnych callbacków.

Technicznie Coroutine są przetwarzane na etapie kompilacji i działają dzięki transformacji kodu, więc nie musimy przejmować się wymaganiami związanymi ze wsparciem maszyny wirtualnej lub systemu operacyjnego.

Podstawowe pojęcia:

**Suspend function** – funkcja, która może zostać zawieszona i wykorzystana w ramach Coroutine. Co to oznacza? Można zawiesić wykonanie funkcji do późniejszego wykonania bez blokowania wątku. Blokowanie jest bardzo kosztowne (mamy też ograniczoną ilość wątków), natomiast operacja zawieszenia jest praktycznie darmowa i nie wymaga zmiany kontekstu lub innej akcji systemu operacyjnego.

**Coroutine builder** – służy do tworzenia Coroutine. Mamy kilka podstawowych możliwości:

* **launch()** – tworzy Coroutine i zapomina o niej. Zwraca natomiast obiekt typu **Job**, który pozwala na późniejsze anulowanie wykonywanie danej Coroutine. Wszystkie niezłapane wyjątki spowodują crash aplikacji.
* **async()** — tworzy Coroutine i zwraca obiekt typu **Deferred<out T> : Job**. Obiekt jest „obietnicą”, że w przyszłości zostanie zwrócona wartość T. Aby mieć do niej dostęp musimy wykonać na nim **await().** Ewentualnie będzie zawierać wyjątek, który wystąpił podczas wykonywania operacji.
* **withContext()** – tworzy wewnętrzną Coroutine, która zostanie wykonana na określonym wątku.
* **runBlocking()** – tworzy nową Coroutine i blokuje obecny wątek (przydatny w trakcie pisania testów).

**Coroutine context** – przekazuje wykonanie Coroutine na odpowiedni wątek.

Podstawowe dostępne konteksty (wątki) to:

* **CommonPool** – wywołuje funkcję na wątku „w tle” (aktualnie jest to domyślny kontekst).
* **UI** – wywołuje funkcję na głównym wątku Androida.
* **Unconfined** – wywołuje funkcję na obecnym wątku (przydatny w trakcie pisania testów).

**Przykład wykorzystania:**

Biorąc pod uwagę, że analizujesz dane z 1000 różnych witryn. Musisz wysłać requesta do każdej witryny, poczekać na odpowiedź, a następnie zrobić coś z zagregowanymi danymi.Dzięki wielowątkowości uruchomisz 200 wątków w puli, z których każdy zjada pamięć, wyśle ​​żądanie, a następnie zaśnie. Po zakończeniu niektórych wątków wracają do puli wątków, wykonują kolejne wywołanie i ponownie śpią.Dzięki coroutines możesz wykonać wszystkie 1000 jednoczesnych połączeń w jednym wątku!

Wszystkie połączenia są uruchamiane pojedynczo w jednym wątku. Po wysłaniu żądania wątek przełącza się na inny program. Po wysłaniu wszystkich żądań wątek jest w stanie uśpienia, dopóki nie zaczną pojawiać się wyniki. Po otrzymaniu wszystkich odpowiedzi wątek jest kontynuowany z zebranymi wynikami.

fun parseAll(urls: List<String>) = runBlocking {

val client = HttpClient { }

val results = urls

.map { url ->

async {

client.get<String>(url)

}

}

.awaitAll()

// Do something with results

}

Źródła:

<https://geek.justjoin.it/kotlin-coroutine-pewno-kolejny-krok-dla-programisty-rxjava>

<https://medium.com/@minogin/kotlin-coroutines-real-life-example-d940cc170217>

<http://minogin.com/kotlin-coroutines-real-life-example/>