برنامه نویسی دستگاه های سیار (CE364)

جلسه سیزدهم:
اجرای موازی کارها

سجاد شیرعلی شهرضا

پاییز 1401

شنبه، 26 آذر 1401

اطلاع رسانی

- بخشهای مرتبط با این جلسه:

- Unit 4: Connect to the internet:
  - Pathway 4: Advanced navigation app examples

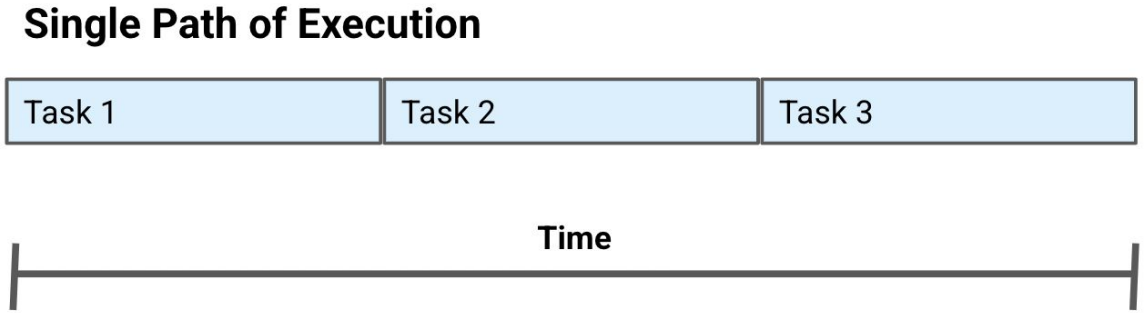
سوال؟

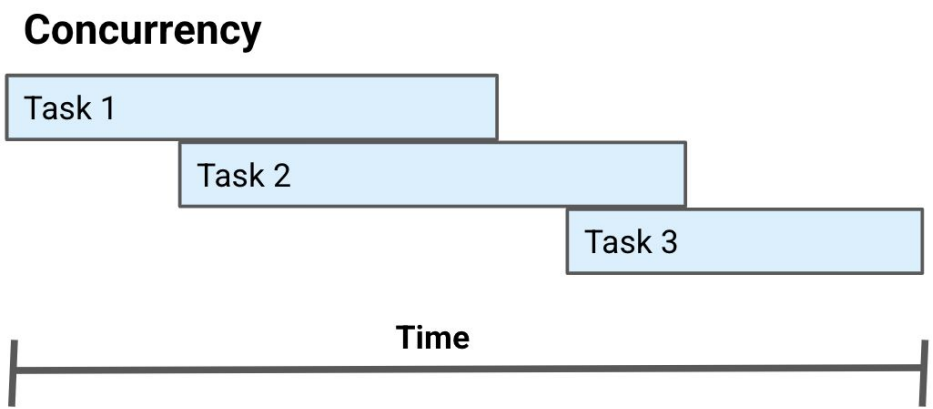
اجرای همزمان

مفهوم ریسمان

- ریسمان (thread): اجرای مجموعه ای از دستورات متوالی
- پیش فرض: یک ریسمان برای کل برنامه
- وظیفه ریسمان اصلی: نمایش رابط کاربری و تعامل با کاربر
- انجام کارهای سنگین پردازشی و یا مرتبط با ورودی خروجی
  - اضافه کردن تاخیر به تعامل کاربر
  - کاهش کیفیت تعامل کاربر
- راه حل: جدا کردن کارهای پشت زمینه از ریسمان اصلی مسئول رابط کاربری

ایده کلی اجرای همزمان





ریسمان

- کوچک ترین واحد برنامه قابل برنامه ریزی برای اجرا

```
fun main() {
    val thread = Thread {
        println("${Thread.currentThread()} has run.")
    }
    thread.start()
}
```

```
Thread[Thread-0,5,main] has run.
```

مثال ساده

```
fun main() {
    val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
    repeat(3) {
        Thread {
            println("${Thread.currentThread()} has started")
            for (i in states) {
                println("${Thread.currentThread()} - $i")
                Thread.sleep(50)
            }
        }.start()
    }
}
```

خروجی مثال ساده

```
Thread[Thread-0,5,main] has started
Thread[Thread-1,5,main] has started
Thread[Thread-2,5,main] has started
Thread[Thread-1,5,main] - Starting
Thread[Thread-0,5,main] - Starting
Thread[Thread-2,5,main] - Starting
Thread[Thread-1,5,main] - Doing Task 1
Thread[Thread-0,5,main] - Doing Task 1
Thread[Thread-2,5,main] - Doing Task 1
Thread[Thread-0,5,main] - Doing Task 2
Thread[Thread-1,5,main] - Doing Task 2
Thread[Thread-2,5,main] - Doing Task 2
Thread[Thread-0,5,main] - Ending
Thread[Thread-2,5,main] - Ending
Thread[Thread-1,5,main] - Ending
```

محدودیت ها ی ریسمان

- سرباره استفاده از ریسمان
  - سرباره ایجاد ریسمان
  - سرباره جابجایی بین ریسمان ها
  - سرباره مدیریت ریسمان ها
- رقابت (race) و غیر قابل پیش بینی بودن
  - عدم مشخص بودن دقیق چگونگی اجرا و جابجایی بین ریسمان ها
  - مشکلات دسترسی به داده مشترک

مثال رقابت میان ریسمان ها

```
fun main() {
    var count = 0
    for (i in 1..50) {
        Thread {
            count += 1
            println("Thread: $i count: $count")
        }.start()
    }
}
```

```
Thread: 50 count: 49 Thread: 43 count: 50 Thread: 1 count: 1
Thread: 2 count: 2
Thread: 3 count: 3
Thread: 4 count: 4
Thread: 5 count: 5
Thread: 6 count: 6
Thread: 7 count: 7
Thread: 8 count: 8
Thread: 9 count: 9
Thread: 10 count: 10
```

```
Thread: 39 count: 41
Thread: 41 count: 41
Thread: 38 count: 41
Thread: 37 count: 41
Thread: 35 count: 41
Thread: 33 count: 41
Thread: 36 count: 41
Thread: 34 count: 41
Thread: 31 count: 41
Thread: 32 count: 41
Thread: 44 count: 42
Thread: 46 count: 43
Thread: 45 count: 44
Thread: 47 count: 45
Thread: 48 count: 46
Thread: 42 count: 47
Thread: 49 count: 48
```

سوال؟

کوروتین

ساختار ساده تر و کاراتر برای برنامه نویسی همزمان

کوروتین (coroutine)

- ساختار راحت تر و ساده تر در کاتلین برای مدیریت همزمانی
  - در کنار امکان استفاده و کار مستقیم با ریسمان ها
- امکان ذخیره حالت
  - فراهم آوردن امکان توقف و شروع مجدد یک کوروتین
  - پیاده سازی شده توسط continuations
  - تعیین زمانی که یک قسمت از برنامه نیاز به در اختیار داشتن کنترل دارد
  - توقف تا انجام کاری توسط قسمت دیگری از برنامه

مفاهیم اصلی

- کار (job)
  - یک واحد از کار که قابل لغو است
    - مثلا کار ساخته شده با launch
- حوزه کوروتین (CoroutineScope)
  - مجموعه توابع برای ایجاد و مدیریت کارها
    - مثلا launch و async
  - حوزه مربوط به مجموعه ای از کارها
- توزیع کننده (dispatcher)
  - مسئول انتخاب ریسمان برای اجرای هر کار

توزیع کننده

- مسئول تعیین یک ریسمان برای اجرای یک کار
- حذف نیاز به مدیریت دستی برای ایجاد و استفاده از ریسمان ها
- بهبود کارایی با جلوگیری از ایجاد ریسمان جدید
- توزیع کننده اصلی (main dispatcher)
  - مسئول اجرای برنامه ریسمان اصلی و رابط کاربری
- توزیع کننده های دیگر برای کارهای خاص:
  - Default, IO, Unconfined

مثال ساده استفاده از کوروتین

```
import kotlinx.coroutines.*

fun main() {
    repeat(3) {
        GlobalScope.launch {
            println("Hi from ${Thread.currentThread()}")
        }
    }
}
```

```
Hi from Thread[DefaultDispatcher-worker-2@coroutine#2,5,main]
Hi from Thread[DefaultDispatcher-worker-1@coroutine#1,5,main]
Hi from Thread[DefaultDispatcher-worker-1@coroutine#3`,5,main]
```

تابع launch

- برچسب suspend:
  - مشخص کردن امکان توقف و شروع دوباره

```
fun CoroutineScope.launch {
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
}
```

مثال از اجرای غیر همزمان توابع

```kotlin
import kotlinx.coroutines.*
import java.time.LocalDateTime
import java.time.format.DateTimeFormatter

val formatter = DateTimeFormatter.ISO_LOCAL_TIME
val time = { formatter.format(LocalDateTime.now()) }

suspend fun getValue(): Double {
    println("entering getValue() at ${time()}")
    delay(3000)
    println("leaving getValue() at ${time()}")
    return Math.random()
}

fun main() {
    runBlocking {
        val num1 = getValue()
        val num2 = getValue()
        println("result of num1 + num2 is ${num1 + num2}")
    }
}
```

```
entering getValue() at 17:44:52.311
leaving getValue() at 17:44:55.319
entering getValue() at 17:44:55.32
leaving getValue() at 17:44:58.32
result of num1 + num2 is 1.4320332550421415
```

اجرای همزمان دو تابع

```
fun main() {
    runBlocking {
        val num1 = async { getValue() }
        val num2 = async { getValue() }
        println("result of num1 + num2 is ${num1.await() + num2.await()}")
    }
}
```

```
entering getValue() at 22:52:25.025
entering getValue() at 22:52:25.03
leaving getValue() at 22:52:28.03
leaving getValue() at 22:52:28.032
result of num1 + num2 is 0.8416379026501276
```

تابع async

```
Fun CoroutineScope.async() {
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
}: Deferred<T>
```

نوع داده Deferred

- در زبان های دیگر: Promise, Future
- یک کار قابل لغو که حاوی اشاره گر به یک مقدار است
- امکان استفاده در ادامه برنامه
- مقدار آن در آینده آماده خواهد بود
- برای صبر کردن و گرفتن مقدار آن:
  - استفاده از تابع await
- ایده:
  - شروع کار از الان
  - عدم صبر برای اتمام آن در حال حاضر

توابع با برچسب suspend

- توابعی که ممکن است برای مدتی متوقف شوند
- یک نمونه: تابع delay
- هر تابعی که یک تابع قابل توقف دیگر را صدا بزند، برچسب قابل توقف خواهد داشت
- توجه کنید runBlocking یک تابع قابل توقف نیست
  - در نتیجه در مثال قبلی، تابع main برچسب توقف ندارد

مثال تبدیل استفاده از ریسمان به کوروتین (استفاده از ریسمان)

```
fun main() {
    val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
    repeat(3) {
        Thread {
            println("${Thread.currentThread()} has started")
            for (i in states) {
                println("${Thread.currentThread()} - $i")
                Thread.sleep(50)
            }
        }.start()
    }
```

مثال تبدیل استفاده از ریسمان به کوروتین (استفاده از کوروتین)

```
import kotlinx.coroutines.*

fun main() {
    val states = arrayOf("Starting", "Doing Task 1", "Doing Task 2", "Ending")
    repeat(3) {
        GlobalScope.launch {
            println("${Thread.currentThread()} has started")
            for (i in states) {
                println("${Thread.currentThread()} - $i")
                delay(5000)
            }
        }
    }
}
```

سوال؟