Parallel Part II

Concurrency

06016415 Functional Programming

IT KMITL
พระจอมเกล้าลาดกระบัง

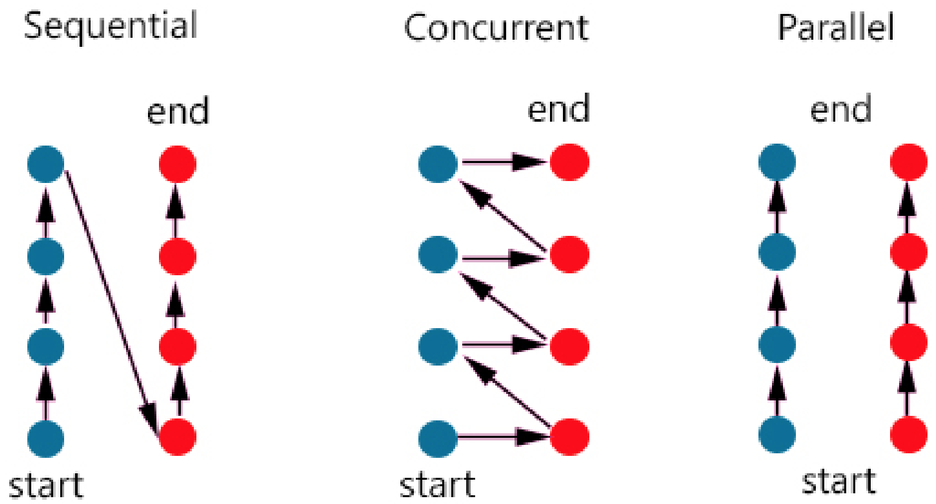
Outline

- Main Concepts
- Futures and Threads
- Synchronization

Concurrency VS Parallel

- **Concurrency programming:** ดำเนินการให้โครงสร้างของโปรแกรมเป็นงานที่เกิดขึ้นไปพร้อมๆกัน (concurrent tasks)

- **Parallel programming:** การประมวลโปรแกรมแบบขนาน โดยมีหลายส่วนดำเนินการไปพร้อมกัน (multiple parts of program execute) ณ เวลาหนึ่งๆ ซึ่งจะเร่งความเร็วในการคำนวณแบบขนาน



ในการทำงานของโปรแกรม จะสามารถทำให้ทำงาน<u>ไม่ต่อเนื่องกันได้</u>หรือไม่?

- **Multiple threads of execution :**
  - แต่ละ threads รันโปรแกรมตามลำดับ แต่จะมีหลาย threads ดำเนินการพร้อมกัน
- **Asynchronous operations:**
  - หยุดการเรียงลำดับการทำงาน แล้วให้บางส่วนดำเนินการไปด้วย แบบไม่เรียงลำดับ

Single-thread

Sequential Execution

```
println("Task 1")
Thread.sleep(1000) // wait 1000 ms
println("Task 2")
```

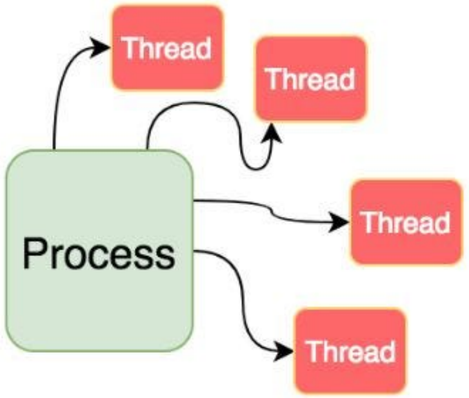
Multi-thread

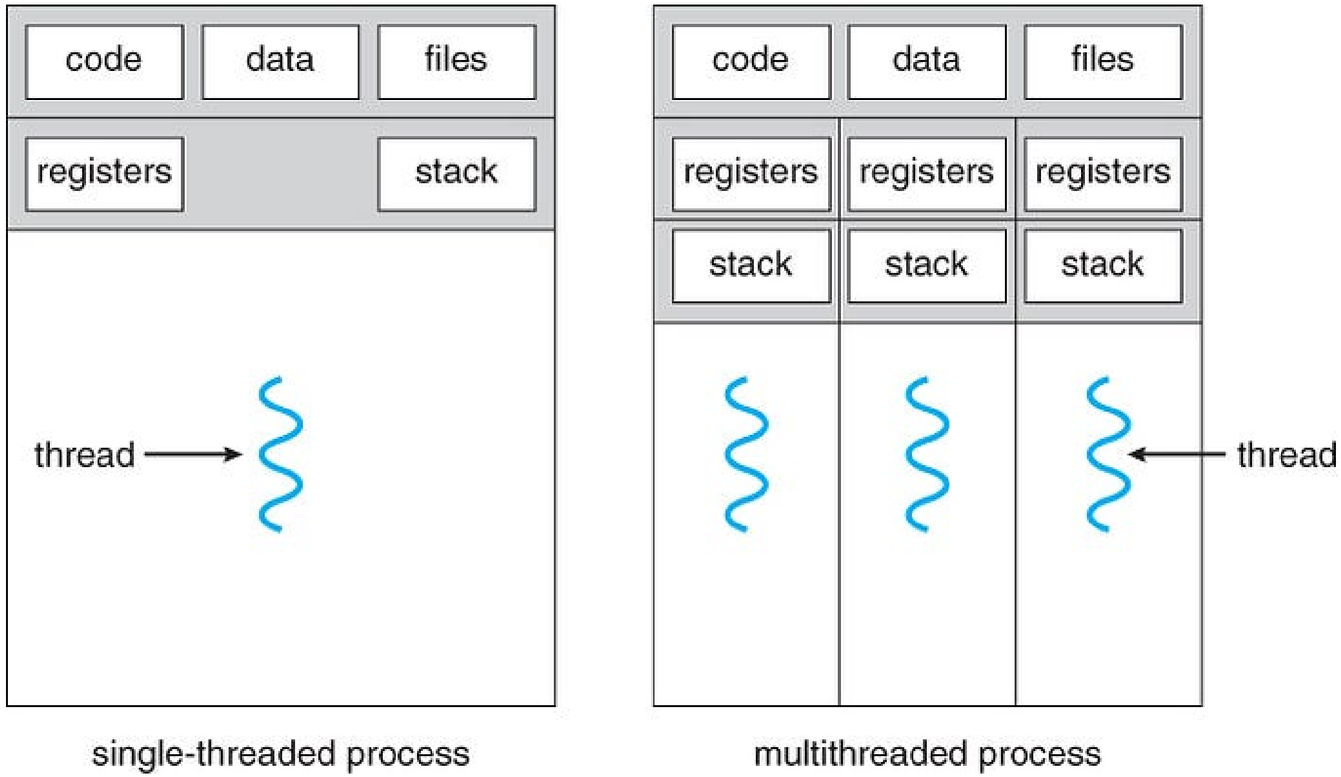
Parallel Execution

```scala
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global

val task1 = Future {
  Thread.sleep(1000)
  println("Task 1 completed")
}

val task2 = Future {
  Thread.sleep(1000)
  println("Task 2 completed")
}
@main def hello(): Unit =
  Thread.sleep(2000) // waif for Future complete
```

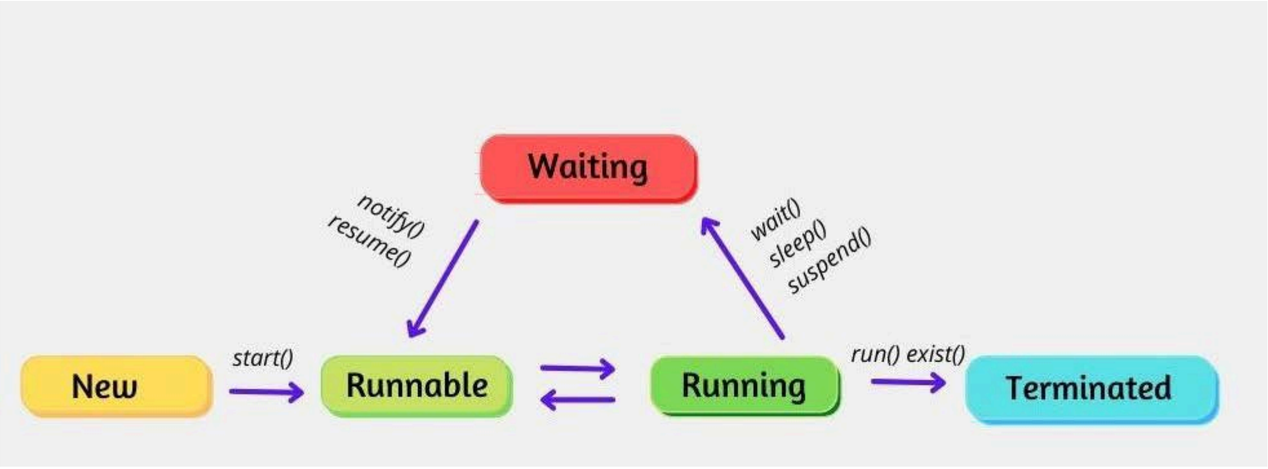


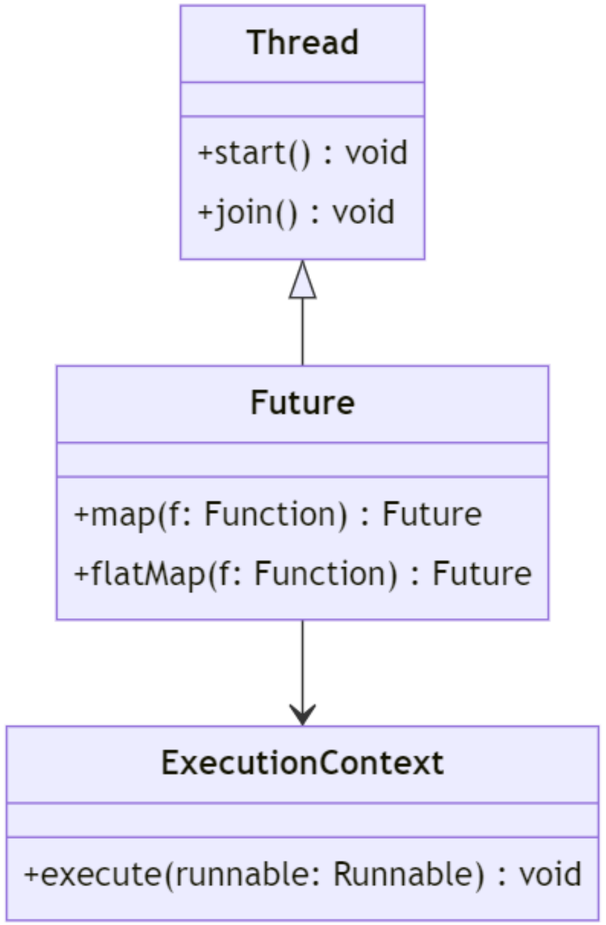


single-threaded process

multithreaded process

https://vivadifferences.com/13-difference-between-process-and-thread-in-os/





https://marketsplash.com/tutorials/scala/scala-multithreading/
https://blog.knoldus.com/things-to-know-about-multithreading-in-scala/

Pseudocode

```
println('A')
println('B')
println('C')
```

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Failure, Success}
```

```
def Print_one(): Unit =
  println('A')
  Future(println('B'))
  println('C')
```

(printing of A) + (printing of B) + (printing of C)

(printing of A) + max((printing of B) + (printing of C))

```
def slowPrint(x: Any) =
  var n = BigInt("1000000000")
  while n > 0 do n -= 1
  println(x)

def Print_two(): Unit =
  slowPrint('A')
  Future(slowPrint('B'))
  slowPrint('C')
```

Daemon Thread : *ทำแบบเบื้องหลัง แบบ background tread*

```scala
val t = new Thread(() => {
  while (true) {
    println("Daemon running...")
    Thread.sleep(500)
  }
})

run | debug
@main def hello(): Unit =
  t.setDaemon(true) // make t be Daemon Thread
  t.start()

  Thread.sleep(2000) // wait 2000 ms
  println("Main thread exits")
```

```
[info] running hello
Daemon running...
Daemon running...
Task 2 completed
Task 1 completed
Daemon running...
Daemon running...
Main thread exits
[success] Total time: 10 s, completed Feb 25, 2025, 9:48:36 AM
Daemon running...
```

- **Asynchronization** (การทำงานไม่พร้อมกัน) : การกระตุ้นให้เกิดนการดำเนินการไม่เป็นไปตามคำสั่ง
  - println('B’) คือ synchronously
  - **Future**(println('B’)) คือ asynchronously

- **Threads of execution** หรือ Threads : เป็นแตกการทำงานออกให้ทำพร้อมกัน

- **Synchronization** (การทำงานพร้อมกัน) : ใช้เมื่อจำเป็นในการประสานงานสิ่งที่เกิดขึ้นพร้อมกัน

- **Nondeterminism** : เป็นผลลัพธ์ของการเกิดพร้อมกัน และจะเป็นอุปสรรคของการเขียนโปรแกรมที่เกิดขึ้นพร้อมกัน
  - เช่น กำหนด multithreaded โดยไม่มี Synchronizationเพิ่มเติม อาจก่อให้เกิดปัญหาได้

- **Future :** เป็นการแสดงค่าที่อาจมีอยู่ หรือไม่มีในปัจจุบัน แต่ในอนาคตจะสามารถใช้ได้ แต่หากกำหนดข้อยกเว้น ก็จะไม่สามารถนำค่านั้นออกมาใช้ได้

- **Promise :** ใช้เพื่อบรรลุค่าผลลัพธ์ของ Future ให้สำเร็จ (โดยการ "ทำตามสัญญา" ให้สำเร็จ) แต่ในทางตรงกันข้าม หากการสัญญานั้นยังไม่สามารถบรรลุได้ในอนาคตเพราะกำหนดข้อยกเว้นไว้ ก็จะแสดงผลลัพธ์เป็น ไม่สำเร็จ

- **Key:**
  - Future มีไว้สำหรับงานที่เกิดขึ้นพร้อมกันเพียงครั้งเดียว ซึ่งอาจใช้เวลานานกว่าจะส่งค่ากลับ
  - ประโยชน์ของ Future คือใช้งานได้กับคำสั่งที่ทำงานพร้อมกับการเรียกย้อนกลับที่หลากหลาย ซึ่งทำให้กระบวนการทำงานกับ Threads ที่เกิดขึ้นพร้อมกันง่ายขึ้น
  - ในการใช้งาน Future ไม่จำเป็นต้องกังวลกับรายละเอียด thread (เพราะทำงานแทนแล้ว) แต่ต้องจัดการกับผลลัพธ์ของอนาคตด้วยวิธีการเช่น onComplete, andThen หรือ ใช้ filter, map เป็นต้น

```scala
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Success, Failure}

val future1 = Future {
  Thread.sleep(1000)
  println("Task 1 completed")
  1
}
val future2 = Future {
  Thread.sleep(1000)
  println("Task 2 completed")
  2
}
val result = for {
  res1 <- future1
  res2 <- future2
} yield res1 + res2

run | debug
@main def hello(): Unit =
  result.onComplete {
  case Success(value) => println(s"Result: $value")
  case Failure(ex)    => println(s"Failed: ${ex.getMessage}")
}

  Thread.sleep(3000)
```

```scala
import scala.concurrent.{Promise, Future}
import scala.concurrent.ExecutionContext.Implicits.global


  def calculate_f(input: Int): Future[Int] = Future {
    // Simulate a long computation
    Thread.sleep(10000)
    input * input
  }

  def calculate_fp(input: Int): Promise[Int] = {
    val promise = Promise[Int]()
    Future {
      // Simulate a long computation
      Thread.sleep(10000)
      promise.success(input * input)
    }
    promise
  }
```

https://docs.scala-lang.org/overviews/core/futures.html#promises

Thread & Future

```scala
def aShortRunningTask(): Int =
  Thread.sleep(500)
  42

val x = aShortRunningTask()


def longRunningAlgorithm(): Int =
  Thread.sleep(10000)
  42

val eventualInt = Future(longRunningAlgorithm())
  //eventualInt
val a = Future(longRunningAlgorithm()).map(_ * 2)
```

```scala
// simulate a slow-running method
def slowlyDouble(x: Int, delay: Long): Future[Int] = Future {
  Thread.sleep(delay)
  x * 2
}

val f = slowlyDouble(2, 5000)
```

```scala
def multipleFutures(): Unit =
  println(s"creating the futures:   ${delta()}")
  // (1) start the computations that return futures
  val f1 = Future { sleep(800); 1 }   // eventually returns 1
  val f2 = Future { sleep(200); 2 }   // eventually returns 2
  val f3 = Future { sleep(400); 3 }   // eventually returns 3
  // (2) join the futures in a `for` expression
  val result =
    for
      r1 <- f1
      r2 <- f2
      r3 <- f3
    yield
      println(s"in the 'yield': ${delta()}")
      (r1 + r2 + r3)
  // (3) process the result
  result.onComplete {
    case Success(x) =>
      println(s"in the Success case: ${delta()}")
      println(s"result = $x")
    case Failure(e) =>
      e.printStackTrace
  }
  println(s"before the 'sleep(3000)': ${delta()}")
  // important for a little parallel demo: keep the jvm alive
  sleep(3000)
```

https://docs.scala-lang.org/scala3/book/concurrency.html

Synchronization

- ในการใช้ Thread ในกรณีที่ต้องการทำงานประสานกัน โดยทั่วไปทำผ่านการ Synchronization
- เราจะใช้ สามารถใช้ synchronizers ประเภทต่างๆ เพื่อปิดกั้น Thread จนกว่าสถานะของแอปพลิเคชันจะอนุญาตให้ดำเนินการได้
- แต่ข้อระวัง คือ การใช้ synchronizers โดยไม่จำเป็น (เพื่อปิดกั้น Thread) อาจทำให้โปรแกรมช้าลง หรือแม้กระทั่งหยุดทำงานในกรณีที่เกิดสะดุด หรือทำงานไม่ได้

- Synchronization ถูกใช้ประโยชน์สำหรับ runtime-system เพื่อเพิ่มประสิทธิภาพการใช้หน่วยความจำ ของงานแบบขนาน (memory usage of parallel tasks) ทำให้หน่วยความจำประสิทธิภาพจากการปรับให้เหมาะสม

Syntax

```
synchronized {
// your logic goes here ..
}
```

- ทำ Synchronization ใช้เมื่อเราแบ่งปันทรัพยากร และเราไม่ต้องการให้มีการแก้ไขหลาย Thread พร้อมกัน
- เพราะมีตรรกะหรือการคำนวณบางอย่างที่ขึ้นอยู่กับข้อมูลอื่น ซึ่งส่งผลให้ ข้อมูลจะไม่สอดคล้องกัน หรืออาจเกิดข้อผิดพลาด และข้อยกเว้นบางอย่างได้

```
synchronized {
a = b + c ; // some logic
z = a  + 10;
}
```

```scala
class SafeBox[A]:
  private var contents = Option.empty[A]
  private val filled = CountDownLatch(1)

  def get: A =
    filled.await()
    synchronized(contents.get)

  def set(value: A): Boolean =
   val setter = synchronized {
      if contents.nonEmpty then false
      else
        contents = Some(value)
        true
  }
    if setter then filled.countDown()
    setter


val exec = Executors.newCachedThreadPool()
val box = SafeBox[Int]()
//exec.execute(() => box.set(0))
//println(box.get)
```

```scala
class Counter {
  private var count = 0

  def increment(): Unit = synchronized {
    val current = count
    // Simulate some work
    Thread.sleep(1000)
    count = current + 1
  }

  def getCount: Int = count
}

def runSynchronizedExample(): Unit = {
  val counter = new Counter()
💡
  val threads = List(
    new Thread(() => counter.increment()),
    new Thread(() => counter.increment())
  )

  threads.foreach(_.start())
  threads.foreach(_.join())

  println(s"The final count is ${counter.getCount}")
}
```

- **Key:**
  - Synchronization ป้องกันการเข้าถึงจาก Multi-Thread บน object เดียวกันพร้อมกัน
  - ช่วยป้องกันปัญหาความไม่สอดคล้องของข้อมูลในอนาคต (future)
  - ทำให้การทำงานของโปรแกรม หรือ process ที่บางครั้งช้ามาก เนื่องจากอนุญาตเพียงครั้งละหนึ่ง Thread เท่านั้น
  - ใน Synchronization ระบบจะล็อกทรัพยากรไว้เพื่อไม่ให้ Thread อื่นสามารถเข้าถึงได้ โดยThread ที่เหลือทั้งหมดจะเข้าสู่สถานะรอ จนกว่า Thread ปัจจุบันจะดำเนินการเสร็จสิ้น
  - ดังนั้น จึงควรพยายามใช้ Synchronization โดยที่ต้องคำนึงถึงค่าผลลัพธ์ที่จะเกิดก่อน (ซึ่งอาจส่งผลต่อค่าปัจจุบัน)