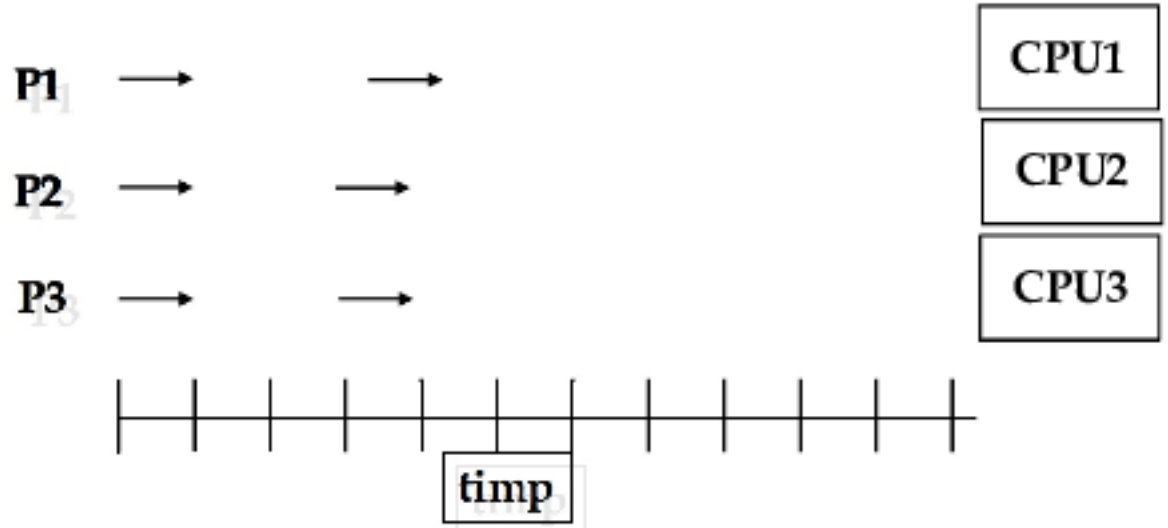
Paradigma Secventiala versus Concurenta

Cursul nr. 10

Mihai Zaharia

Ce este calculul paralel



Numarul de programe/procese active

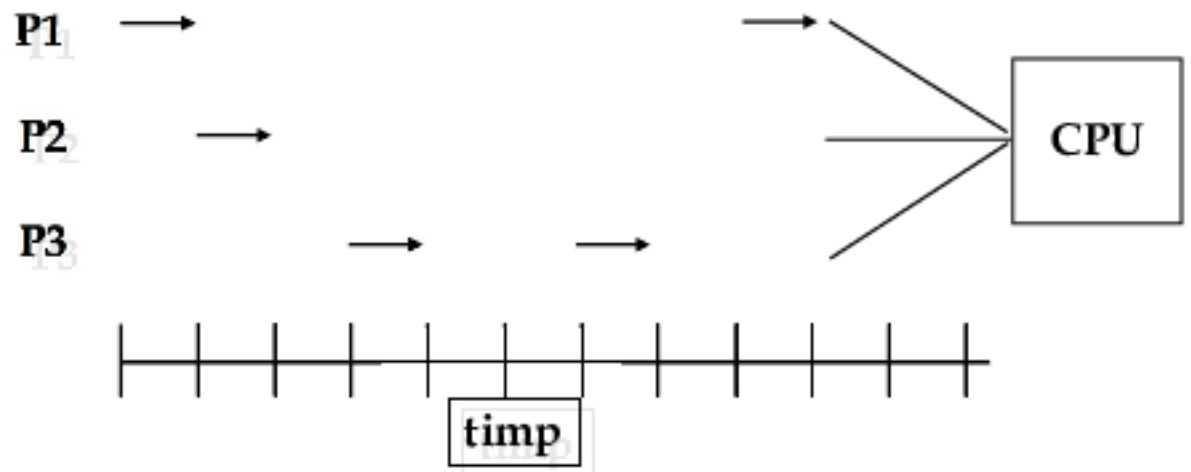
=

numărul de procesoare

Ce este concurența?

- Concureță Vs Paralelism

Concurență

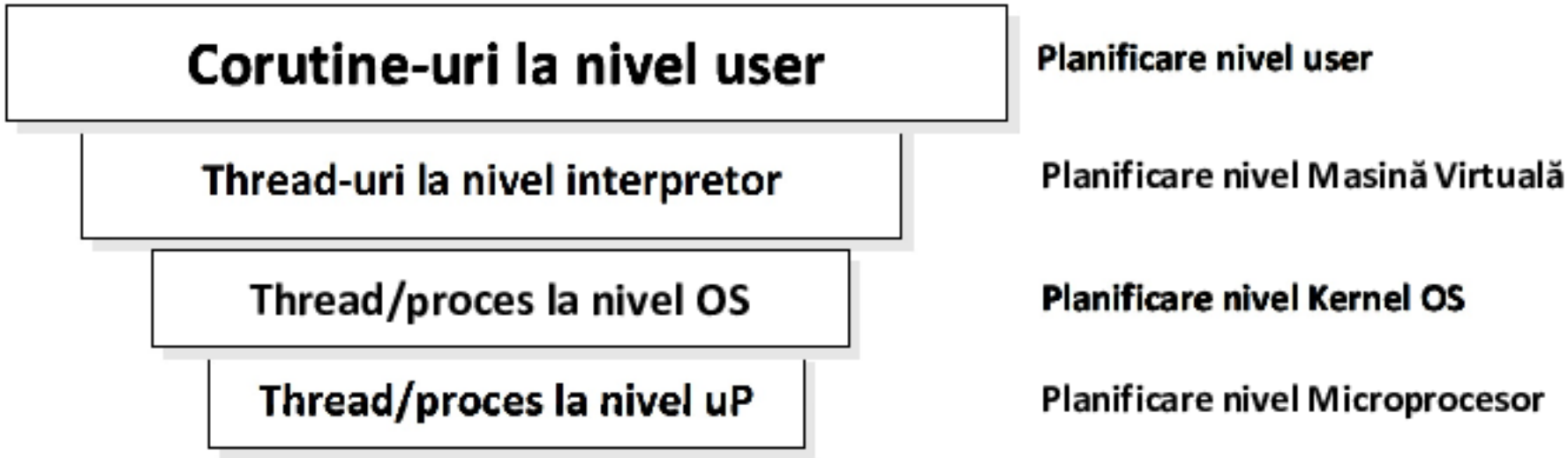


Numărul de entități care efectuează ceva

>

numărul de procesoare

Ierarhia de control la nivel Kotlin

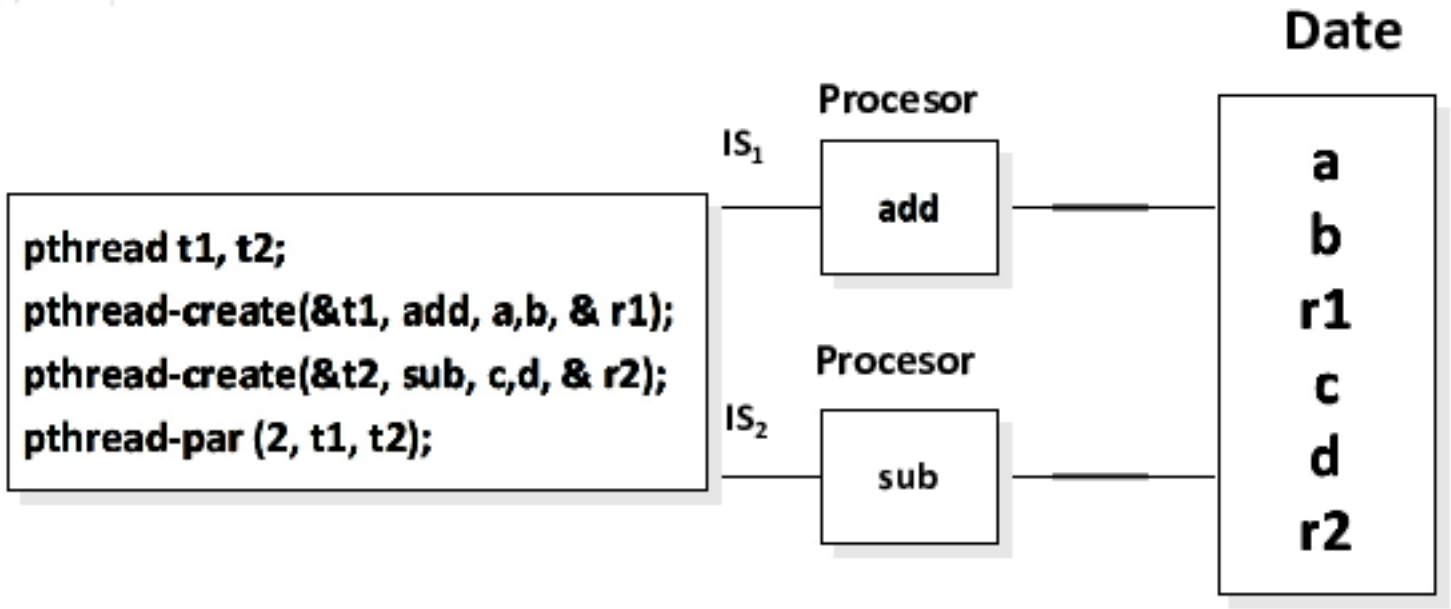


Execuție :

- Concurență de corutine la nivel thread-uri din mașina virtuală
- Concurență de procese/thread-uri la nivel de OS
- Paralelism real: maparea procese / thread-uri : procesor = 1:1

Concurența la nivel de date

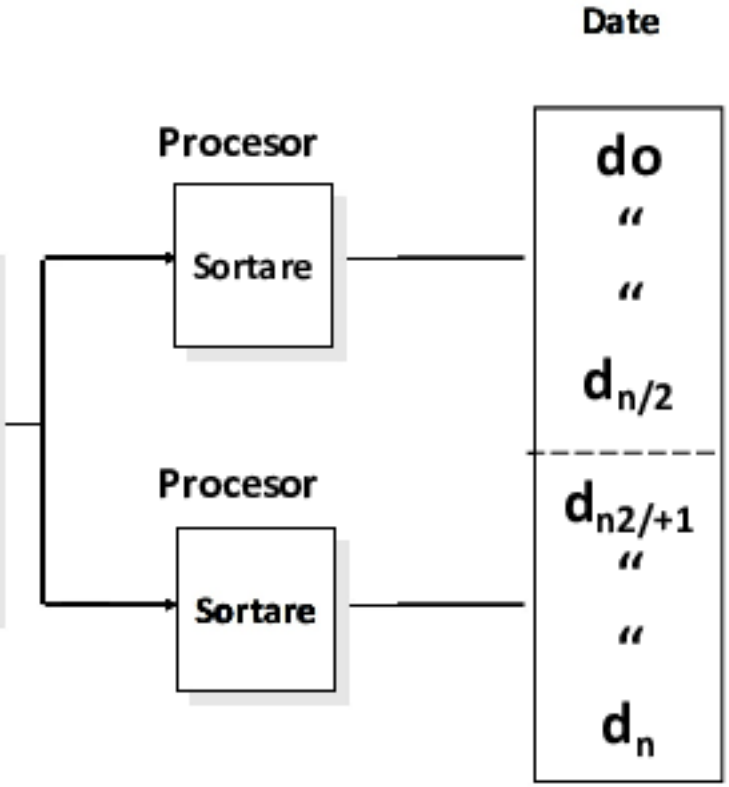
```
int add (int a, int b, int & result)
// corpul
int sub(int a, int b, int & result)
// corpul
```



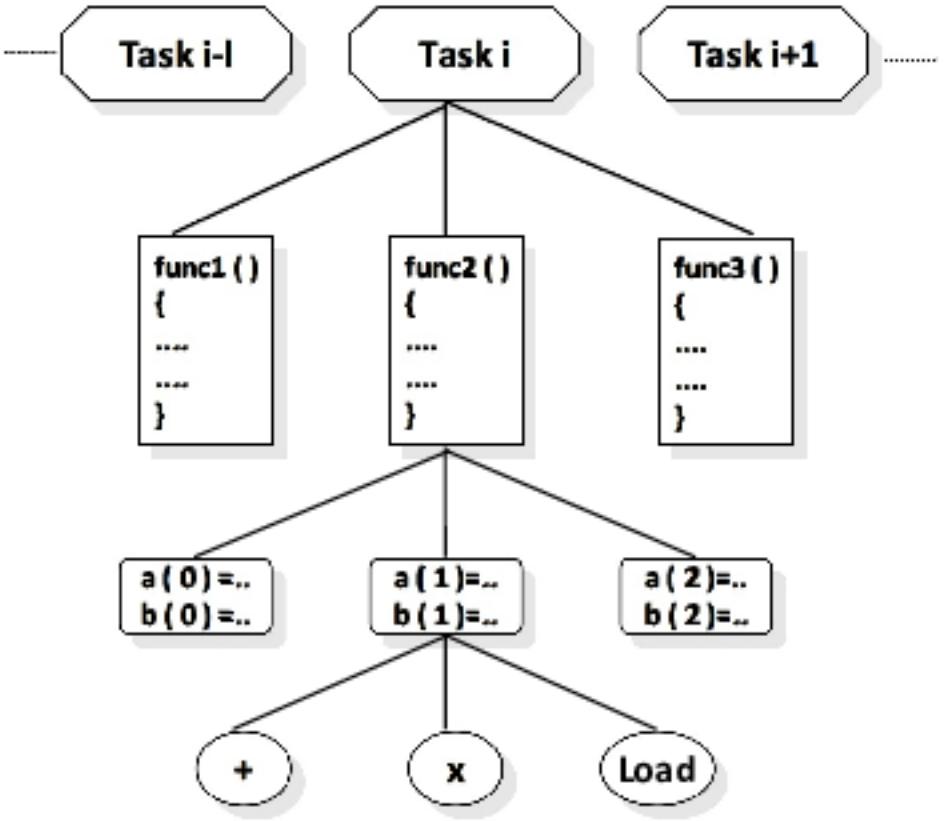
Paralelismul la nivel datelor

```
sort( int *array, int count)
//.....
//.....

pthread-t, thread1, thread2;
“
“
pthread-create(& thread1, sort, array, N/2);
pthread-create(& thread2, sort, array, N/2);
pthread-par(2, thread1, thread2);
```



Granularitatea



Granularitate cod
Entitate Cod
Granularitate mare
(nivel task)
Program

Granularitate medie
(nivel control)
Funcție (corutină/thread)

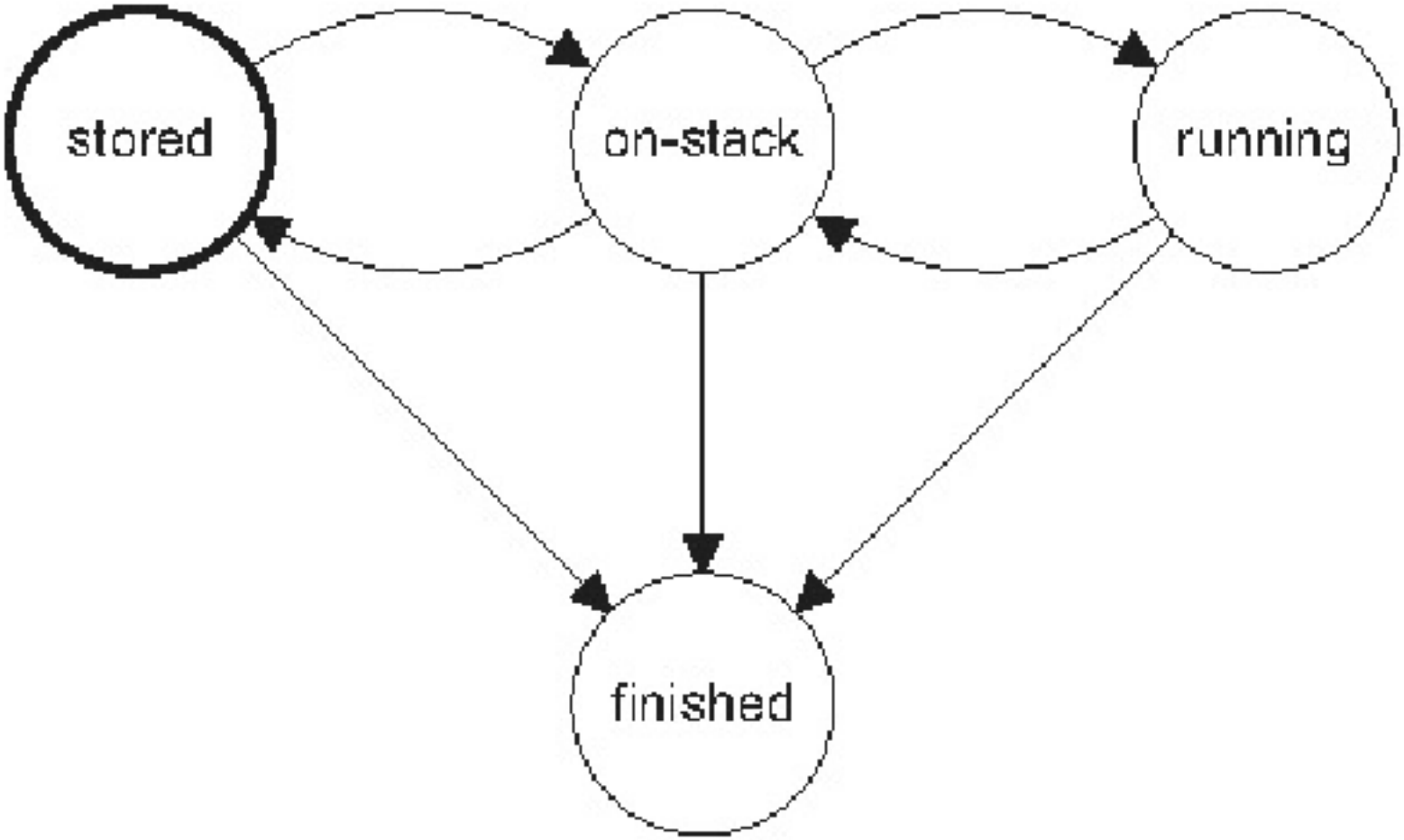
Granularitate fină (nivel date)
Buclă

Granularitate foarte fină
(alegeri multiple)
Cu suport hard

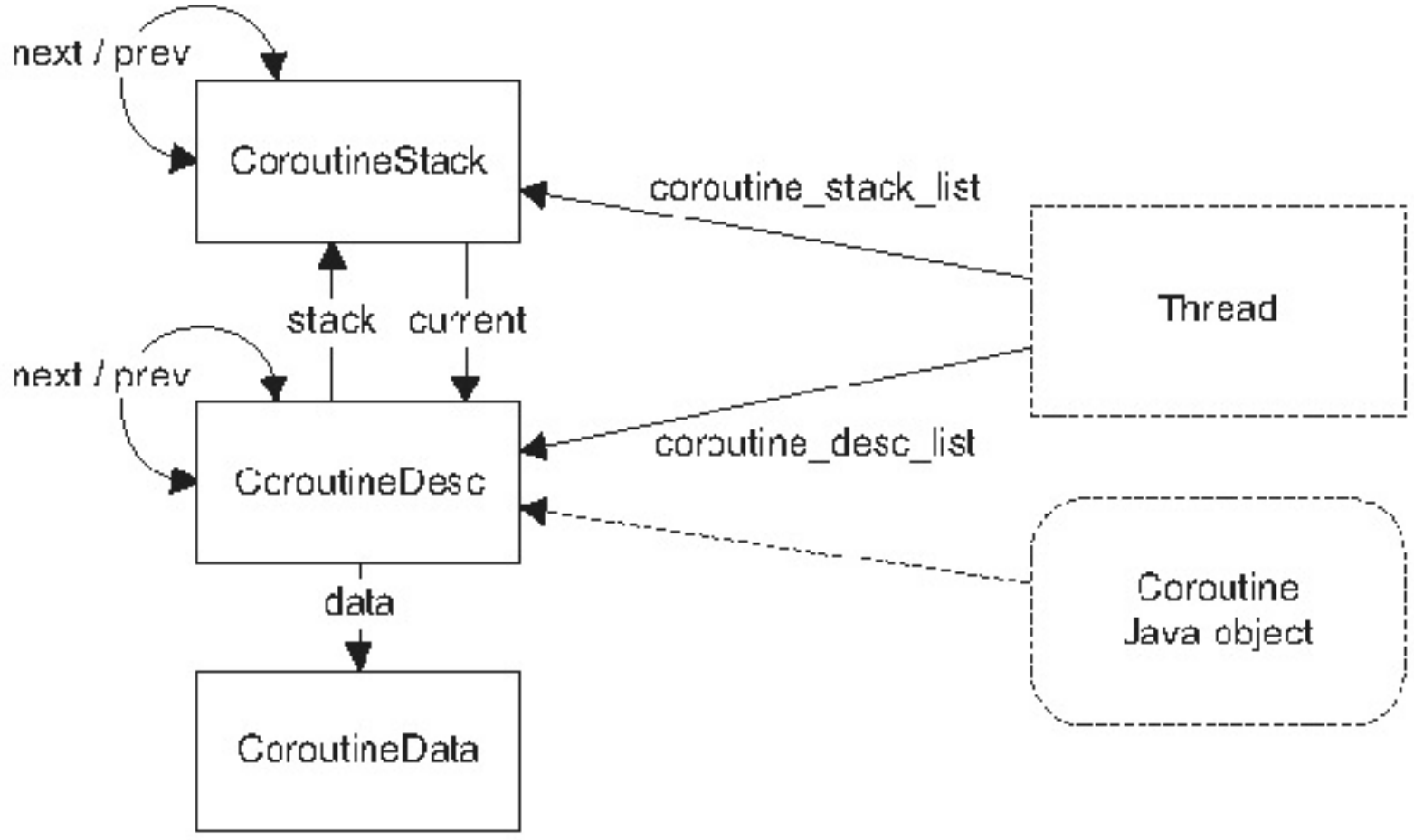
Ce sunt corutinele?

- ceva vechi
- ceva nou

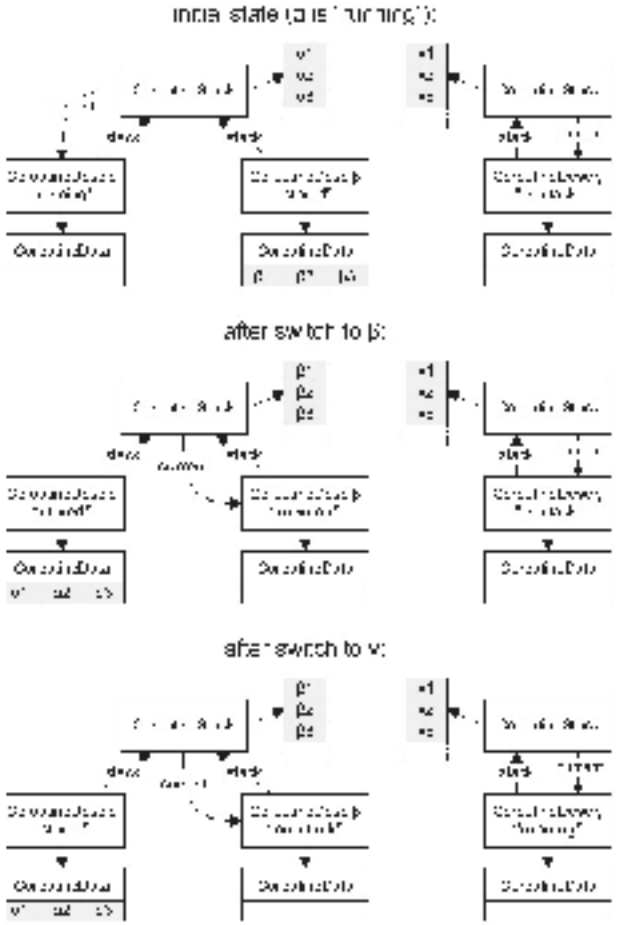
Ciclul de viață al corutinelor



Relația cu JVM



Cum se schimbă stările



Mapare corutine pe thread-uri

```
import kotlin.system.*
import kotlinx.coroutines.*
fun main(args: Array<String>) =
runBlocking {
    println("${Thread.activeCount()} fire
de executie active la pornire")
    val time = measureTimeMillis {
        createCoroutines(10_000)
    }
    println("${Thread.activeCount()} fire
de executie active la sfarsit")
    println("Procesul a durat $time ms")
}
```

```
suspend fun createCoroutines(amount: Int) {
    val jobs = ArrayList<Job>()
    for (i in 1..amount) {
        jobs += GlobalScope.launch {
            println("Am pornit $i in ${Thread.currentThread().name}")
            delay(1000)
            println("S-a terminat $i din
${Thread.currentThread().name}")
        }
    }
    jobs.forEach {
        it.join()
    }
}
//S-a terminat 9998 din DefaultDispatcher-worker-5
//11 fire de executie active la sfarsit
//Procesul a durat 1186 ms - pe sistemul meu
```

Relația corutină-thread

2 fire de executie active la pornire
Pornit 3 in DefaultDispatcher-worker-1
Pornit 5 in DefaultDispatcher-worker-5
Pornit 4 in DefaultDispatcher-worker-3
Pornit 2 in DefaultDispatcher-worker-2
Pornit 6 in DefaultDispatcher-worker-6
Pornit 1 in DefaultDispatcher-worker-4
Pornit 7 in DefaultDispatcher-worker-7
Pornit 8 in DefaultDispatcher-worker-8
Pornit 9 in DefaultDispatcher-worker-4
……..
Terminat 8 din DefaultDispatcher-worker-7
Terminat 6 din DefaultDispatcher-worker-8
Terminat 5 din DefaultDispatcher-worker-5
Terminat 7 din DefaultDispatcher-worker-2
Terminat 2 din DefaultDispatcher-worker-1
Terminat 4 din DefaultDispatcher-worker-4
Terminat 1 din DefaultDispatcher-worker-6
Terminat 3 din DefaultDispatcher-worker-3

Creare diverse tipuri de thread

```
import kotlinx.coroutines.*
fun main() = runBlocking<Unit> {

    launch { // contextul parinte - corutina functiei main cu runBlocking
        println("Corutina principala runBlocking      : Sunt in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Unconfined) { // not confined -- va lucra cu thread-ul principal
        println("Independenta       : Sunt in thread ${Thread.currentThread().name}")
    }
    launch(Dispatchers.Default) { // gestionata de DefaultDispatcher
        println("Implicita             : Sunt in thread ${Thread.currentThread().name}")
    }
    launch(newSingleThreadContext("Threadul Meu")) { // va primi propriul thread
        println("newSingleThreadContext: Sunt in thread ${Thread.currentThread().name}")
    }
}
```

si exemplu de executie

```
Independenta       : Sunt in thread main
Implicita             : Sunt in thread DefaultDispatcher-worker-1
Corutina principala runBlocking      : Sunt in thread main
newSingleThreadContext: Sunt in thread Threadul Meu
```

Exemplu oprire forțată a unei corutine

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        // Emulate some batch processing
        repeat(30) { i ->
            println("Calculam ceva $i ...")
            delay(300L)
        }
    }
    delay(1000L)
    println("main: Utilizatorul a cerut oprirea calculelor")
    job.cancelAndJoin() // da comanda de terminare si astepata efectuarea ei
    println("main: Operatiunea in curs a fost abandonata")
}
```

si rezultatul executiei

```
Calculam ceva 0 ...
Calculam ceva 1 ...
Calculam ceva 2 ...
Calculam ceva 3 ...
main: Utilizatorul a cerut oprirea calculelor
main: Operatiunea in curs a fost abandonata
```

Exemplu de oprire după depasirea limitei de timp

```
import kotlinx.coroutines.*
fun main()
{
puturos()
}
fun puturos()
{
   runBlocking
   {
   val job = launch
     {
      try
       {
        withTimeout(1000L)
            {
            repeat(30) { i ->
            println("Calculez $i ...")
            delay(300L)
            }
          }
        }catch(e: TimeoutCancellationException){println("sunt un lenes si ma opresc")}
      }
   }
}
```

Depășire de timp fără excepții

```
import kotlinx.coroutines.*
import kotlinx.coroutines.withTimeoutOrNull as withTimeoutOrNull1
fun main()
{   if(null == lenes())println("ma opresc din lene")  }
fun lenes(): String? {
var status:String?=""
   runBlocking {
      val status1= withTimeoutOrNull1(1000L) {
         repeat(30) { i ->
            println("Calcul numarul $i ...")
            delay(300L)
         }
         "Gata" //incercati sa-l stergeti
      }
   status=status1
   }
return status;
}
```

Distrugere la ordin

```
import kotlinx.coroutines.*

class Activity : CoroutineScope by CoroutineScope(Dispatchers.Default) {
   fun destroy() {
      cancel() // se realizeaza o extindere a scopului corutinei CoroutineScope
   }
   fun doSomething() {
      repeat(10) { i ->
         launch {
            delay((i + 1) * 200L) // 200ms, 400ms, ... etc
            println("Coroutina $i s-a terminat")
         }
      }
   }
}

fun main() = runBlocking<Unit> {
   val activity = Activity()
   activity.doSomething() // run test function
   println("pornim corutinele")
   delay(500L)
   println("Distrug activitatile!")
   activity.destroy() // le omor pe toate
}
```

si exemplu executie

pornim corutinele
Coroutina 0 s-a terminat
Coroutina 1 s-a terminat
Distrug activitatile!

Thread-local data

```
import kotlinx.coroutines.*

val threadLocal = ThreadLocal<String?>() // se declara referinta catre thread-ul local
fun main() = runBlocking<Unit> {
    threadLocal.set("thread-ul cu prisina:)")
    println("Pre-main, current thread: ${Thread.currentThread()}, numit: '${threadLocal.get()}'")
    val job = launch(Dispatchers.Default + threadLocal.asContextElement(value = "launch")) {
        println("Sunt acum in: ${Thread.currentThread()}, numit: '${threadLocal.get()}'")
        yield()
        println("Dupa yield, sunt in: ${Thread.currentThread()}, numit: '${threadLocal.get()}'")
    }
    job.join()
    println("Dupa ce am oprit thread-urile interne sunt in: ${Thread.currentThread()}, numit:
'${threadLocal.get()}'")
}
```

si executia codului

```
Pre-main, current thread: Thread[main,5,main], numit: 'thread-ul cu prisina:)'
Sunt acum in: Thread[DefaultDispatcher-worker-2,5,main], numit: 'launch'
Dupa yield, sunt in: Thread[DefaultDispatcher-worker-2,5,main], numit: 'launch'
Dupa ce am oprit thread-urile interne sunt in: Thread[main,5,main], numit: 'thread-ul cu prisina:)'
```

Asigurarea coerenței datelor

```
import kotlinx.coroutines.*
import kotlin.system.*
suspend fun CoroutineScope.massiveRun(action: suspend () -> Unit) {
    val n = 100  // numar coroutine care vor fi lansate in executie
    val k = 1000 // numar de repetari a fiecarei corutine
    val time = measureTimeMillis {
        val jobs = List(n)
            {  launch { repeat(k) { action() } }
        }
        jobs.forEach { it.join() }
    }
    println("S-au efectuat ${n * k} operatii in $time ms")
}
val mtContext = newFixedThreadPoolContext(2, "mtPool") // se defineste un context explicit numai cu 2 fire
var counter = 0
fun main() = runBlocking<Unit> {
    CoroutineScope(mtContext).massiveRun {
// se va folosi mt... in loc de Dispatchers.Default pentru a forta aparitia fenomenului
        counter++ //variabila comuna unde vor aparea erori
    }
    println("Numarator = $counter")
}
```

Si un exemplu de executie

S-au efectuat 100000 operatii in 28 ms

Numarator = 90497

Soluții specifice

```
import java.util.concurrent.atomic.*
.....
var counter = AtomicInteger()
.....
GlobalScope.massiveRun {
    counter.incrementAndGet()
}
println("Numarator = ${counter.get()}")
```

si rezultatul executiei

```
Am efectuat 100000 sarcini in 29 ms
Numarator = 100000
```

Izolare cu granularitate mică/fină a firelor

```
GlobalScope.massiveRun {
// desi fiecare corutina este executata cu DefaultDispathcer
   withContext(counterContext) {
// fiecare operatie pe variabila este  limitata la firul unic dedicat
counter++
   }
}
println("Numarator = $counter")
```

si rezultatul executiei

Am terminat 100000 sarcini in 569 ms
Numarator = 100000

Izolarea cu granularitate mare a firelor

```
CoroutineScope(counterContext).massiveRun {
// se executa fiecare corutina intr-un context cu thread unic
   counter++
}
println("Numarator = $counter")
```

si rezultatul executiei

Am terminat 100000 sarcini in 27 ms
Numarator = 100000

Soluția bazată pe excluziunea mutuală

```
GlobalScope.massiveRun {
    mutex.withLock {
        counter++
    }
}
```

withLock echivalenta cu

```
mutex.lock();
try {
...
}
finally { mutex.unlock() }
```

si rezultatul executiei

Am terminat 100000 sarcini in 218 ms
Numarator = 100000

Actori

```
//tipuri de mesaj pentru counterActor
sealed class CounterMsg
object IncCounter : CounterMsg() // mesaj pentru incrementare
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg() // o cerere cu raspuns
acum vom defini o functie care va lansa un actor prin intermediul unui constructir specific
fun CoroutineScope.counterActor() = actor<CounterMsg> {
  var counter = 0 // actor state
  for (msg in channel) { // iterate over incoming messages
    when (msg) {
      is IncCounter -> counter++
      is GetCounter -> msg.response.complete(counter)
    }
  }
}
iar in codul de baza
val counter = counterActor() // creez the actor
GlobalScope.massiveRun { counter.send(IncCounter) }
// send a message to get a counter value from an actor
val response = CompletableDeferred<Int>()
counter.send(GetCounter(response))
println("Counter = ${response.await()}")
counter.close() // termin actorul
```

Si rezultatul executiei

Am terminat 100000 operatii in 248 ms

Numarator = 100000

Async

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

Async - exemplu utilizare

```
import kotlinx.coroutines.*
import java.text.SimpleDateFormat
import java.util.*
fun main() = runBlocking {
   val deferred1 = async { computation1() }
   val deferred2 = async { computation2() }
   printCurrentTime("Astept efectuarea calculelor...")
   val result = deferred1.await() + deferred2.await()
   printCurrentTime("Valoarea calculata este $result")
}
suspend fun computation1(): Int {
   delay(1000L) // simulam durata primei operatii
   printCurrentTime("Am terminat de calculat prima valoare")
   return 131
}
suspend fun computation2(): Int {
   delay(2000L)//simulam durata celui de-al doilea calcul
   printCurrentTime("Am terminat al doilea calcul")
   return 9
}
fun printCurrentTime(message: String) {
   val time = (SimpleDateFormat("hh:mm:ss")).format(Date())
   println("[$time] $message")
}
```

si rezultatul programului

[07:49:59] Astept efectuarea calculelor...
[07:50:00] Am terminat de calculat prima valoare
[07:50:01] Am terminat al doilea calcul
[07:50:01] Valoarea calculata este 140

Produce - este încă în dezvoltare

```
@ExperimentalCoroutinesApi fun <E> CoroutineScope.produce(
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = 0,
    block: suspend ProducerScope<E>.() -> Unit
): ReceiveChannel<E> (source)
```

Deadlock

```
import kotlinx.coroutines.*

lateinit var jobA : Job
lateinit var jobB : Job

fun main(args: Array<String>) = runBlocking {
    jobA = launch {
        println("Sunt in A")
        jobB.join()
        println("S-a terminat B")
    }
    jobB = launch {
        println("Sunt in B")
        jobA.join()
        println("Sa terminat A")
    }
}
```

Si exemplu de executie

```
Sunt in A
Sunt in B

Process finished with exit code 130 (interrupted by signal 2:
SIGINT) (oprit manual)
```