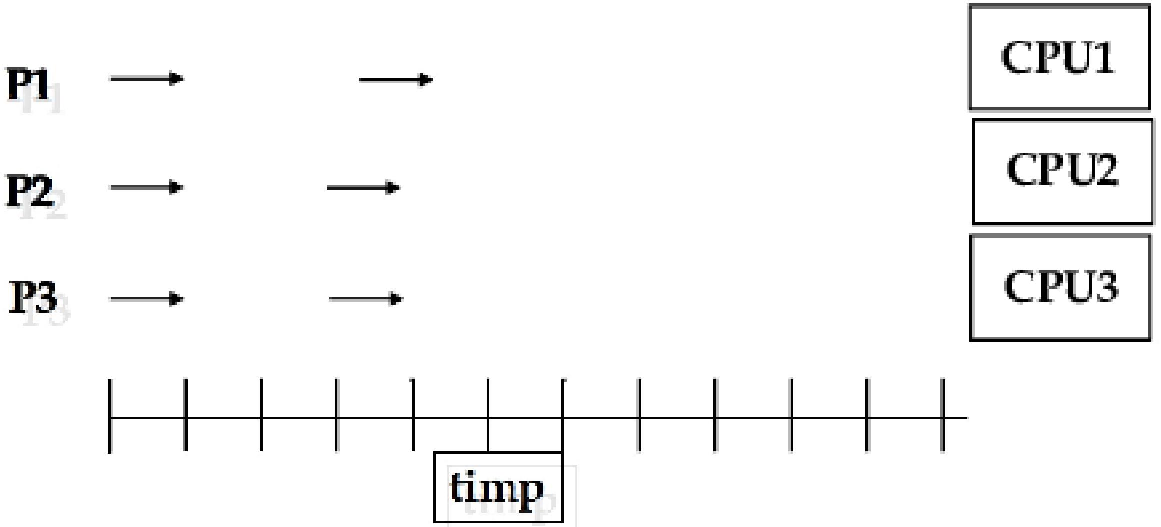
Paradigma Secventiala versus Concurenta

Cursul nr. 10

Mihai Zaharia

Ce este calculul paralel



Numarul de programe/procese active

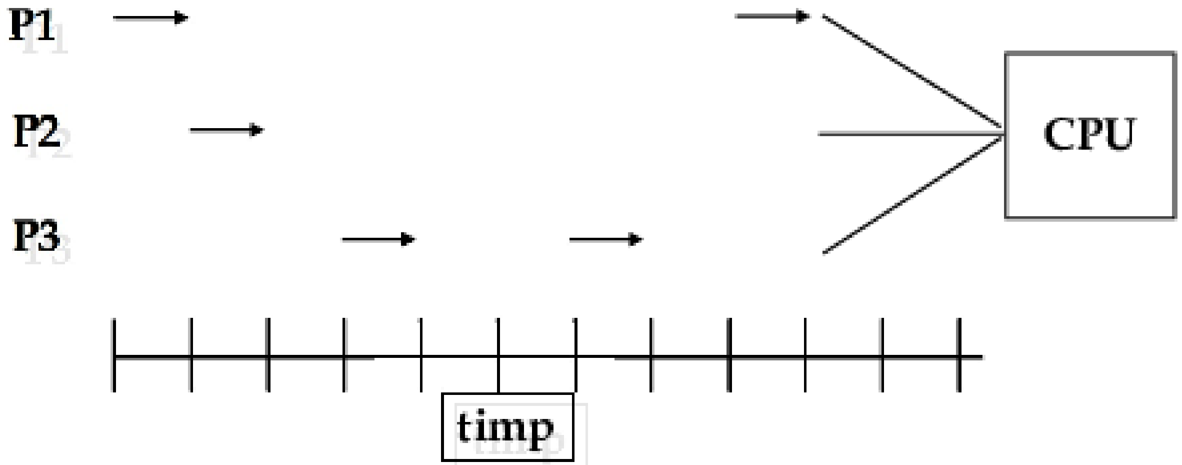
=

numărul de procesoare

Ce este concurența?

- Concureță Vs Paralelism

Concurență

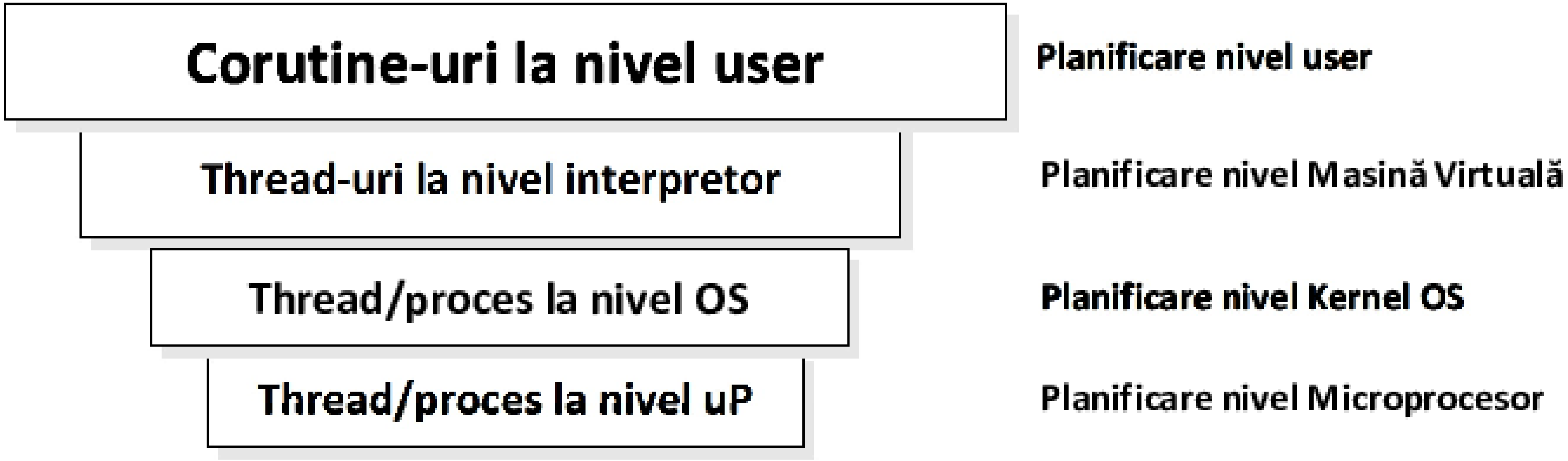


Numărul de entități care efectuează ceva

>

numărul de procesoare

Ierarhia de control la nivel Kotlin



Corutine-uri la nivel user

Planificare nivel user

Thread-uri la nivel interpretor

Planificare nivel Masină Virtuală

Thread/proces la nivel OS

Planificare nivel Kernel OS

Thread/proces la nivel uP

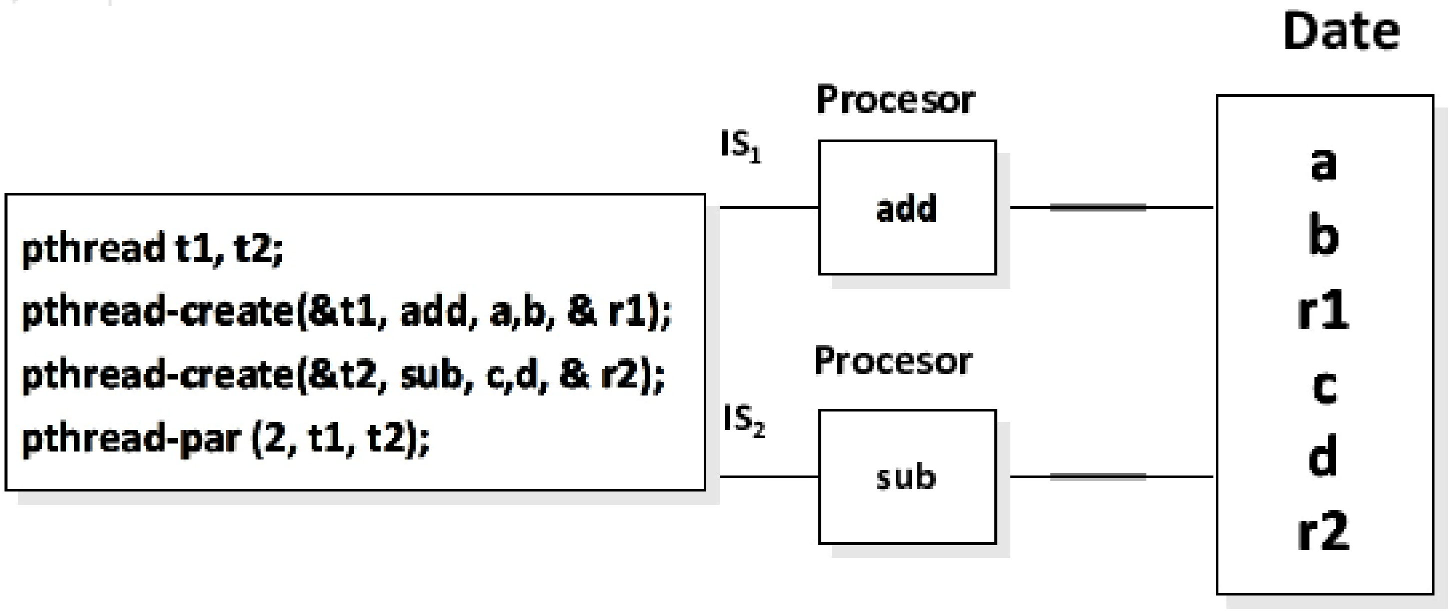
Planificare nivel Microprocesor

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

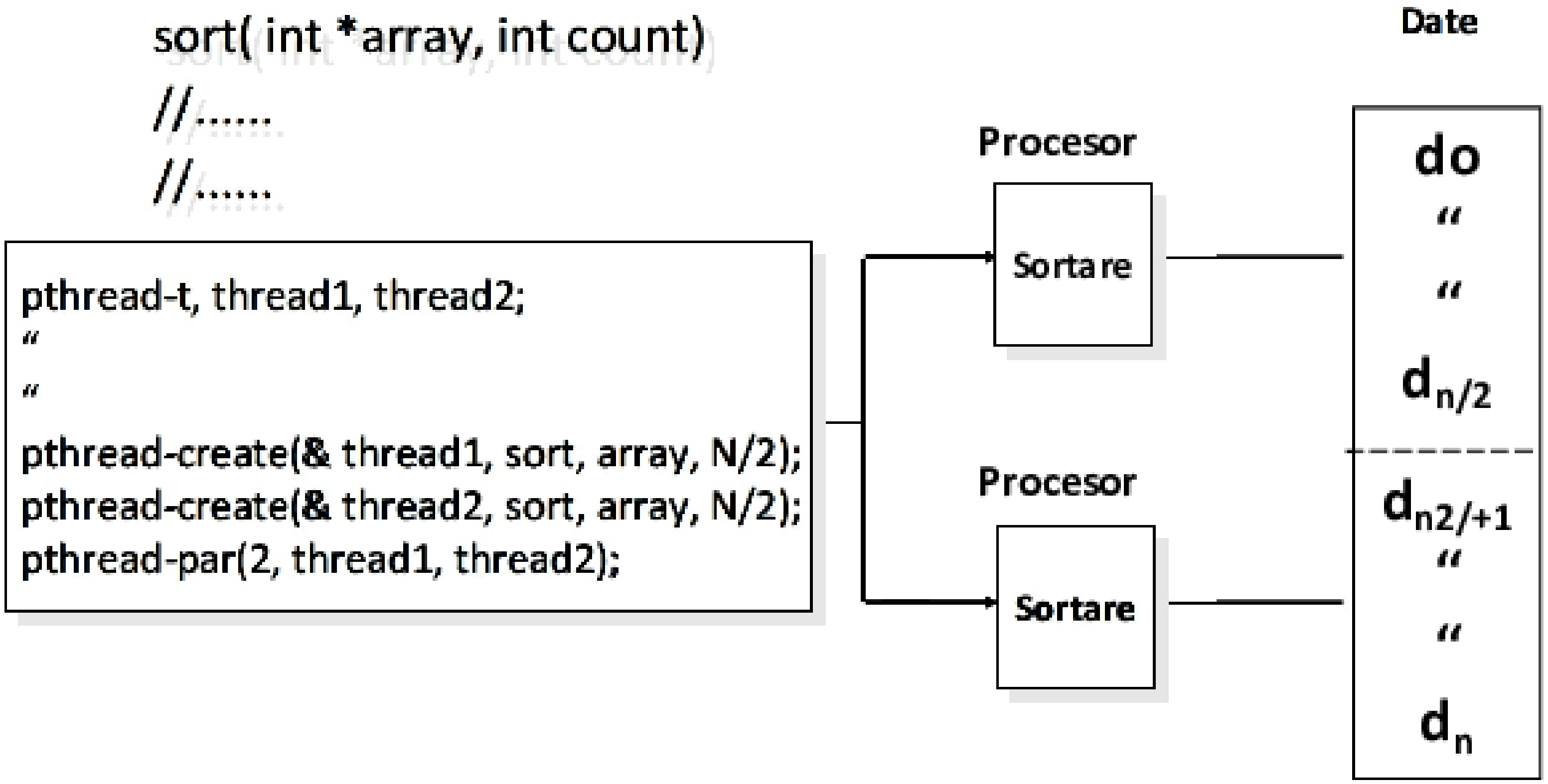
Concurența la nivel de date

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



```
pthread t1, t2;
pthread-create(&t1, add, a,b, & r1);
pthread-create(&t2, sub, c,d, & r2);
pthread-par (2, t1, t2);
```

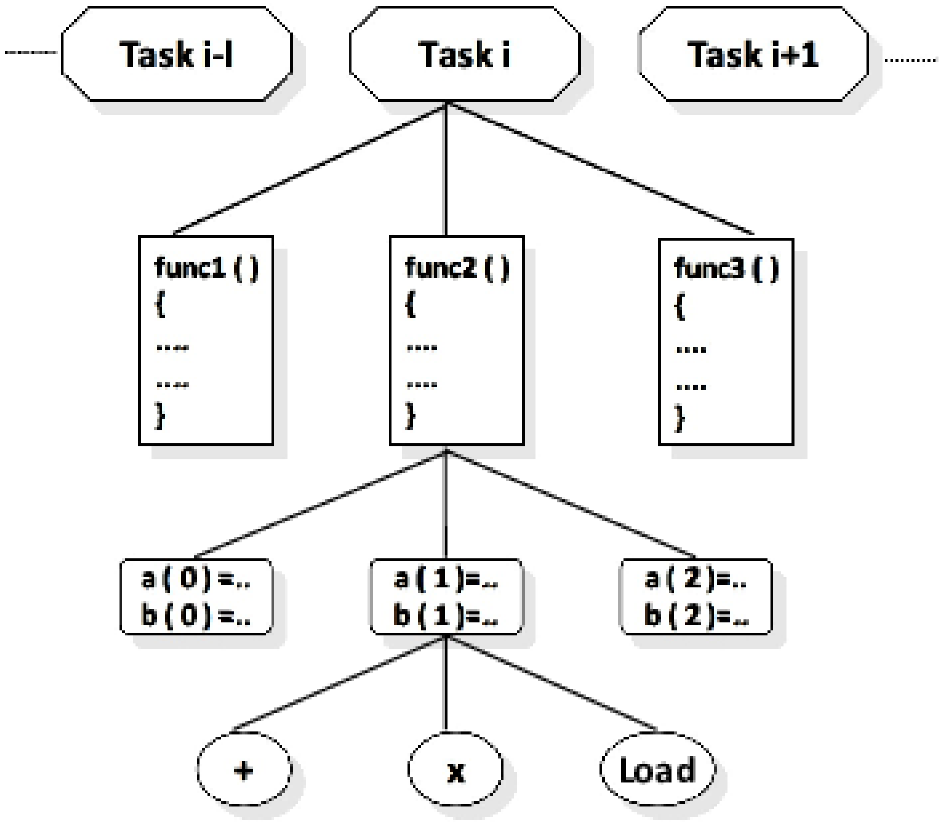
Paralelismul la nivel datelor



```
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)

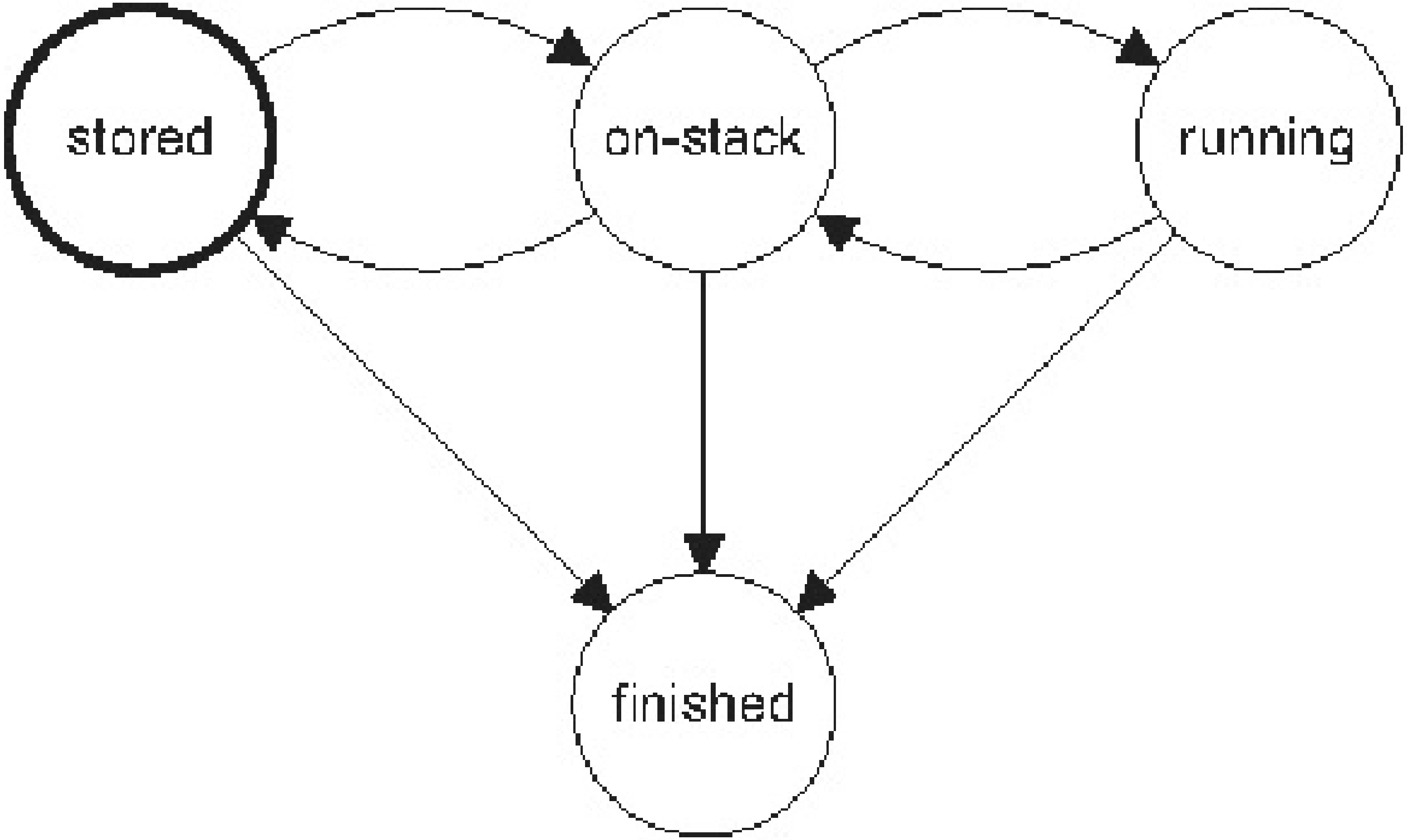
Granularitate fină (nivel date)
Buclă

Granularitate foarte fină
(alegeri multiple)
Cu suport hard

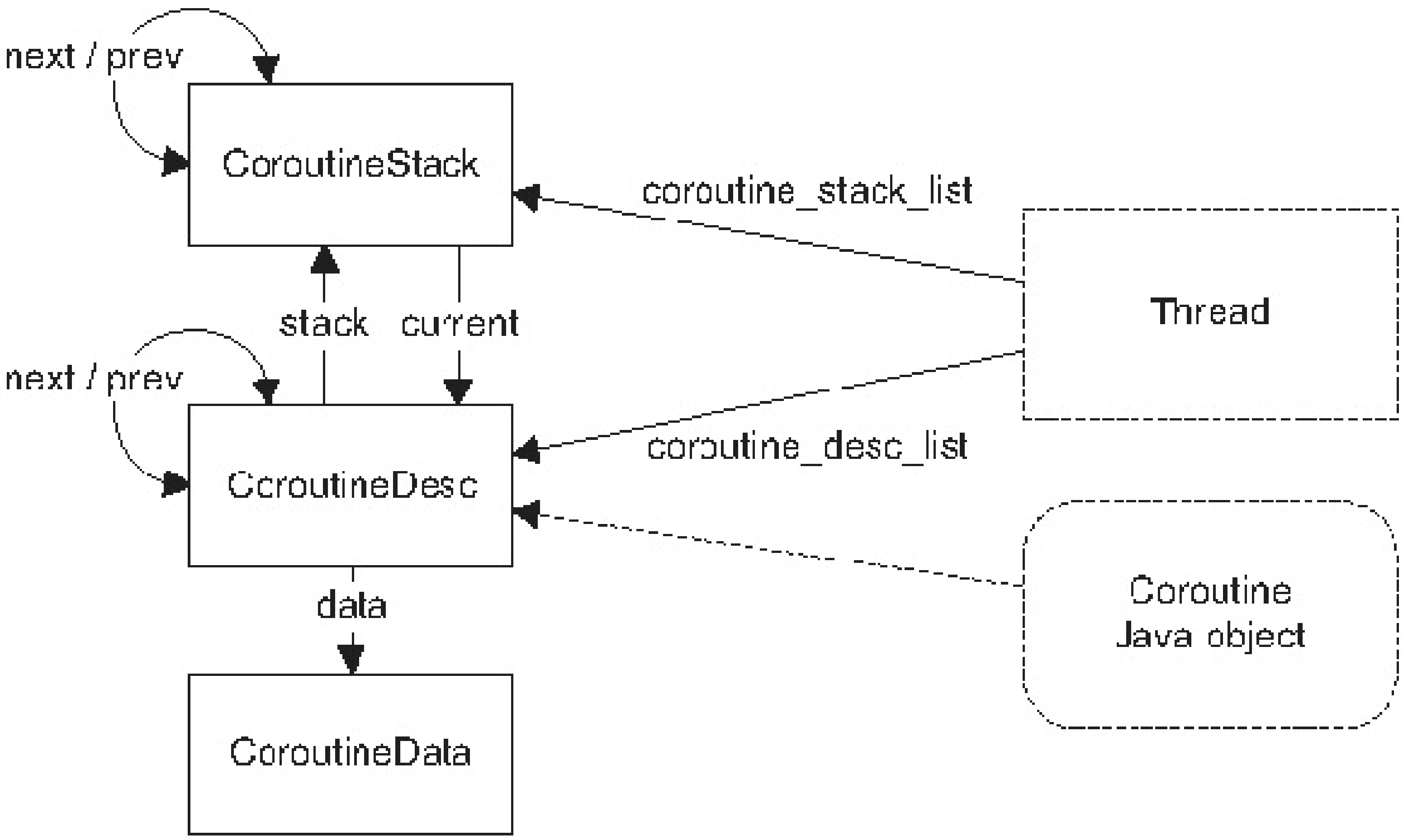
Ce sunt corutinele?

- ceva vechi
- ceva nou

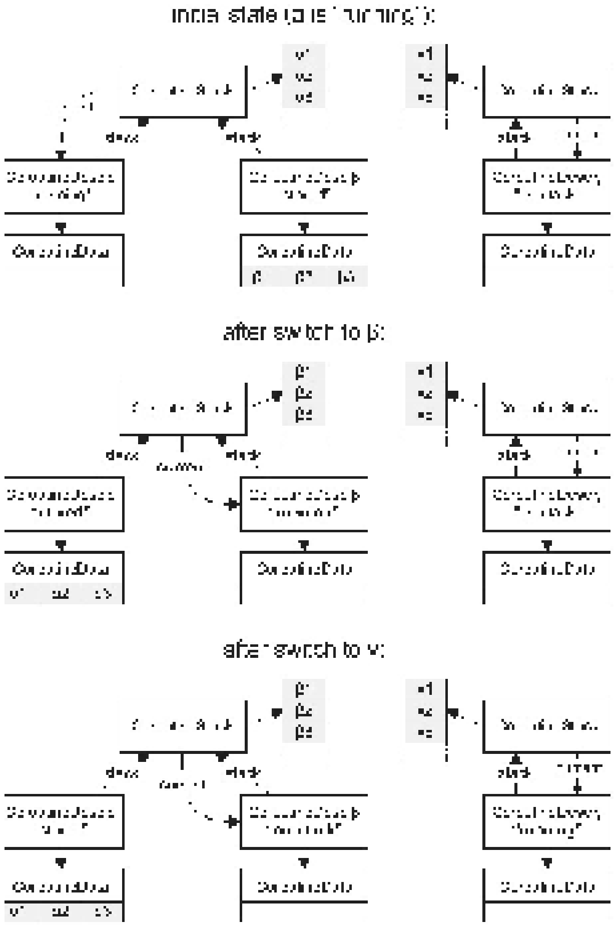
Ciclul de viață al corutinelor



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

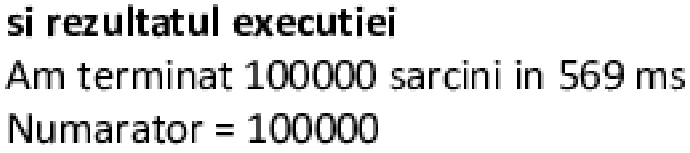
```
import java.util.concurrent.atomic.*
.....
var counter = AtomicInteger()
.....
GlobalScope.massiveRun {
   counter.incrementAndGet()
}
println("Numarator = ${counter.get()}")
```

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)
```

Determinarea stării unui job

Starea	isActive	isCompleted	isCanceled
Created	false	false	false
Active	true	false	false
Canceled	false	true	true
Completed	false	true	false

Optimizare în funcție de numărul core

```
import kotlin.system.*
import kotlinx.coroutines.*
fun main(args: Array<String>) = runBlocking {
   println("${Thread.activeCount()} fire de executie active la pornire")
   val time = measureTimeMillis {
      createCoroutines(10_0)
   }
   println("${Thread.activeCount()} fire de executie active la sfarsit")
   println("Procesul a durat $time ms")
}
suspend fun createCoroutines(amount: Int) {
   val backgroundPool: CoroutineDispatcher by lazy {
      val numProcessors = Runtime.getRuntime().availableProcessors()
      when {
         numProcessors <= 2 -> newFixedThreadPoolContext(2, "background")
         else -> newFixedThreadPoolContext(numProcessors, "background")
          }
   }
```

```
   val jobs = ArrayList<Job>()
   for (i in 1..amount) {
      jobs += GlobalScope.launch(backgroundPool) {
         println("Am pornit $i in
${Thread.currentThread().name}")
         delay(1000)
         println("S-a terminat $i din
${Thread.currentThread().name}")
      }
   }
   jobs.forEach {
      it.join()
   }
 }
```

si rezultat executie

```
2 fire de executie active la pornire
Am pornit 1 in background-1
Am pornit 2 in background-2
....
S-a terminat 69 din background-8
S-a terminat 98 din background-3
S-a terminat 100 din background-2
10 fire de executie active la sfarsit
Procesul a durat 1019 ms
```

Canale

```
val channel = RendezvousChannel<Int>()
```

- fara parametru

```
val rendezvousChannel = Channel<Int>()
```

- similară cu

```
val rendezvousChannel = Channel<Int>(0)
```

- ca efect dar aceasta din urmă poate avea o altă capacitate a tamponului

```
val rendezvousChannel = Channel<Int>(30)
```

Utilizare canale de comunicare

```
import kotlin.system.*
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main(args: Array<String>) = runBlocking {
  val time = measureTimeMillis {
    val channel = Channel<Int>()
    val sender = launch {
      repeat(10) {
        channel.send(it) //am trimis 10 bucati pe canal
        println("Am trimis $it")
      }
    }
    for (i in 1..10) {
      channel.receive() // am primit 10 bucati din canal
    }
  }
  println("Procesul a durat ${time}ms")
}
```

si exemplul de executie

```
Am trimis 0
Am trimis 1
Am trimis 2
Am trimis 3
Am trimis 4
Am trimis 5
Am trimis 6
Am trimis 7
Am trimis 8
Procesul a durat 13ms
Am trimis 9

Process finished with exit code 0
```

Canale neblocante

```
import kotlin.system.*
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val channel = Channel<Int>(Channel.UNLIMITED)
        val sender = launch {
            repeat(10) {
                channel.send(it) //am trimis 10 bucati pe canal
                println("Am trimis $it")
            }
        }
        for (i in 1..8) {
            println(channel.receive() )// am primit 8 bucati din canal
        }
    }
    println("Procesul a durat ${time}ms")
}
```

si rezultatul executiei

```
Am trimis 0
Am trimis 1
Am trimis 2
Am trimis 3
Am trimis 4
Am trimis 5
Am trimis 6
Am trimis 7
Am trimis 8
Am trimis 9 //nepreluate
0
1
2
3
4
5
6
7
Procesul a durat 13ms

Process finished with exit code 0
```

ConflatedChannel

```
import kotlin.system.*
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main(args: Array<String>) = runBlocking {
   val time = measureTimeMillis {
      val channel = Channel<Int>(Channel.CONFLATED)
      launch {
         repeat(5) {
            channel.send(it)
            println("Am trimis $it")
         }
      }
      for (i in 1..5) {
         val element = channel.receive()
         println("Am primit $element")
      }   //comentati for-ul si nu se va bloca
   }
   println("Procesul a durat ${time}ms")
}
```

rezultat cu for activ

```
Am trimis 0
Am trimis 1
Am trimis 2
Am trimis 3
Am trimis 4
Am primit 0
Am primit 4

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

cu for comentat

```
Am trimis 0
Am trimis 1
Am trimis 2
Am trimis 3
Am trimis 4
Am primit 0
Procesul a durat 12ms

Process finished with exit
code 0
```

Thread-uri Kotlin

```
fun thread(
  start: Boolean = true,
  isDaemon: Boolean = false,
  contextClassLoader: ClassLoader? = null,
  name: String? = null,
  priority: Int = -1,
  block: () -> Unit
): Thread //definiția din biliotecă
```

- și un program de test:

```
import kotlin.concurrent.*
fun main(args: Array<String>){
    thread(start = true) {
        println("Thread Kotlin ${Thread.currentThread()} s-a executat.")
    }
}
```

Controlul thread-urilor din Java

```
fun main(args: Array<String>){
   object : Thread() {
      override fun run() {
         println("Sunt in thread-ul singleton ${Thread.currentThread()}")
      }
   }.start()
   val t1=SimpleThread()
   t1.run()
   val t2=SimpleRunnable()
   t2.run()
   val thread = Thread {
      println("Thread lambda ${Thread.currentThread()} s-a executat.")
   }
   thread.start()
}
class SimpleThread: Thread() {
   public override fun run() {
      println("Instanta clasei derivate din Thread ${Thread.currentThread()} s-a executat.")
   }
}
class SimpleRunnable: Runnable {
   public override fun run() {
      println("Instanta clasei care implementeaza Runnable ${Thread.currentThread()} s-a executat.")
   }
}
```

```
Sunt in thread-ul singleton Thread[Thread-0,5,main]
Instanta clasei derivate din Thread Thread[main,5,main] s-a
executat.
Instanta clasei care implementeaza Runnable
Thread[main,5,main] s-a executat.
Thread lambda Thread[Thread-2,5,main] s-a executat.

Process finished with exit code 0
```

Metodă elegantă de apel thread

```
import java.lang.Thread.*

fun main(args: Array<String>){
  val thread1 = thread(start = true, name = "Speedy", priority = MAX_PRIORITY) {
    println("Threadul ${Thread.currentThread()} s-a executat.")
  }
  val thread2 = thread(start = true, name = "Turtle", priority = MIN_PRIORITY) {
    println("Threadul ${Thread.currentThread()} s-a executat.")
  }
}
public fun thread(start: Boolean = true, isDaemon: Boolean = false, contextClassLoader: ClassLoader? = null, name: String? = null, priority: Int = -1, block: () ->
Unit): Thread {
  val thread = object : Thread() {
    public override fun run() {
      block()
    }
  }
  if (isDaemon)
    thread.isDaemon = true
  if (priority > 0)
    thread.priority = priority
  if (name != null)
    thread.name = name
  if (contextClassLoader != null)
    thread.contextClassLoader = contextClassLoader
  if (start)
    thread.start()
  return thread
}
```

si rezultatul executiei

```
Threadul Thread[Speedy,10,main] s-a executat.
Threadul Thread[Turtle,1,main] s-a executat.

Process finished with exit code 0
```

Utlizarea funcțiilor/blocurilor cu excluziune mutuală

```
import java.util.concurrent.*
fun main(args: Array<String>){
   val g=gigel()
   val executor = Executors.newFixedThreadPool(5)
   for (i in 0..9) {
      val worker = Runnable {
         println("Sunt in firul " + i)
         g.synchronizedMethod()
         g.methodWithSynchronizedBlock()
      }
      executor.execute(worker)
   }
   executor.shutdown()
   while (!executor.isTerminated) {
   }
   println("S-au terminat toate firele din piscina")
}
class gigel {
   @Synchronized
   fun synchronizedMethod() {
      println("Sunt in metoda sincronizata ${Thread.currentThread()}")
   }
   fun methodWithSynchronizedBlock() {
      println("Zona fara sincronizare: ${Thread.currentThread()}")
      synchronized(this) {
         println("Sectiune cu sincronizare: ${Thread.currentThread()}")
      }
   }
}
```

si exemplu partial de iesire

```
Sunt in firul 0
Sunt in firul 1
Sunt in metoda sincronizata Thread[pool-1-thread-1,5,main]
Zona fara sincronizare: Thread[pool-1-thread-1,5,main]
Sunt in metoda sincronizata Thread[pool-1-thread-2,5,main]
Zona fara sincronizare: Thread[pool-1-thread-2,5,main]
Sectiune cu sincronizare: Thread[pool-1-thread-1,5,main]
Sunt in firul 2
Sunt in firul 3
Sunt in firul 4
Sunt in metoda sincronizata Thread[pool-1-thread-5,5,main]
Sunt in firul 5
Zona fara sincronizare: Thread[pool-1-thread-5,5,main]
Sunt in metoda sincronizata Thread[pool-1-thread-4,5,main]
Zona fara sincronizare: Thread[pool-1-thread-4,5,main]
Sunt in metoda sincronizata Thread[pool-1-thread-3,5,main]
Zona fara sincronizare: Thread[pool-1-thread-3,5,main]
Sectiune cu sincronizare: Thread[pool-1-thread-2,5,main]
Sectiune cu sincronizare: Thread[pool-1-thread-3,5,main]
....
S-au terminat toate firele din piscina
```

Variabile comune inter-thread - @Volatile

```
import java.util.concurrent.*
fun main(args: Array<String>){
   val executor = Executors.newFixedThreadPool(5)
   for (i in 0..4) {
      val worker = Runnable {
         println("Sunt in firul " + i)
         println(faceceva.inc())
      }
      executor.execute(worker)
   }
   executor.shutdown()
   while (!executor.isTerminated) {
   }
   println("S-au terminat toate firele din piscina")
}
object faceceva {
   @Volatile
   private var i = 0
   fun inc(): Int {
      i=i+1
      return i
   }
}
```

si exemplul de executie

```
Sunt in firul 0
Sunt in firul 1
Sunt in firul 2
Sunt in firul 3
1
4
2
3
Sunt in firul 4
5
S-au terminat toate firele din piscina

Process finished with exit code 0
```

Metodele Wait & Notify

- **wait()**
  - Apelată de un obiect
- **notifyAll()**
  - Apelată de un obiect
  - Trebuie să aibă deja controlul asupra lock-ului respectivului obiect.

wait(), notify() and notifyAll() la piață

```
import java.util.*
import kotlin.concurrent.thread
class TaranOrasean(private val maxItems: Int) {
   @Volatile private var items = 0
   private val rand = Random()
   private val lock= java.lang.Object()
   fun produce() = synchronized(lock) {
      while (items >= maxItems) {
         lock.wait()
      }
      items++
      println("Am produs $items: alimente in ${Thread.currentThread()}")
      lock.notifyAll()
   }
   fun consume() = synchronized(lock) {
      while (items <= 0) {
         lock.wait()
      }
      items--
      println("Am utilizat $items : alimente in ${Thread.currentThread()}")
      lock.notifyAll()
   }
}
fun main(args: Array<String>) {
   println("starting: ${Thread.currentThread()}")

   val example = TaranOrasean(5)

   for (i in 0..14) {
      thread(start = true) {
         if (i< 5) {
            example.consume()
         } else {
            example.produce()
         }
      }
   }
   println("S-a inchis piata: ${Thread.currentThread()}")
}
```

Exemplu simplu reflecție Java

```
fun main(args: Array<String>){
    val s = "Hello world"
    val length = s.javaClass.getMethod("length")
    val x = length.invoke(s) as Int
    println(x)
}
```

Exemplu de reflexie generică Kotlin - proprietăți

```
fun main(args: Array<String>){
    val prop = Person::name
    print(prop)
}
class Person(val name: String, var age: Int) {
    fun present() = "Sunt $name, si am $age ani"
    fun greet(other: String) = "Salut, $other, sunt $name"
}
```

si iesirea

```
val Person.name: kotlin.String
Process finished with exit code 0
```

Immutable KProperty1<R, V>,

mutable KMutableProperty1<R, V>.

Reflexie la nivel de instanță

```
fun main(args: Array<String>){//inspectie instanta Kotlin
   val person = Person("Lisa", 23)
   println(person.present())
   printProperty(person, Person::name)
   incrementProperty(person, Person::age)
   println(person.present())
}

class Person(val name: String, var age: Int) {
   fun present() = "Sunt $name, si am $age ani"
   fun greet(other: String) = "Salut, $other, sunt $name"
}
fun <T> printProperty(instance: T, prop: KProperty1<T, *>) {
      println("${prop.name} = ${prop.get(instance)}")
   }
fun <T> incrementProperty( instance: T, prop: KMutableProperty1<T, Int>) {
   val value = prop.get(instance)
   prop.set(instance, value + 1)
}
```

Si iesirea programului

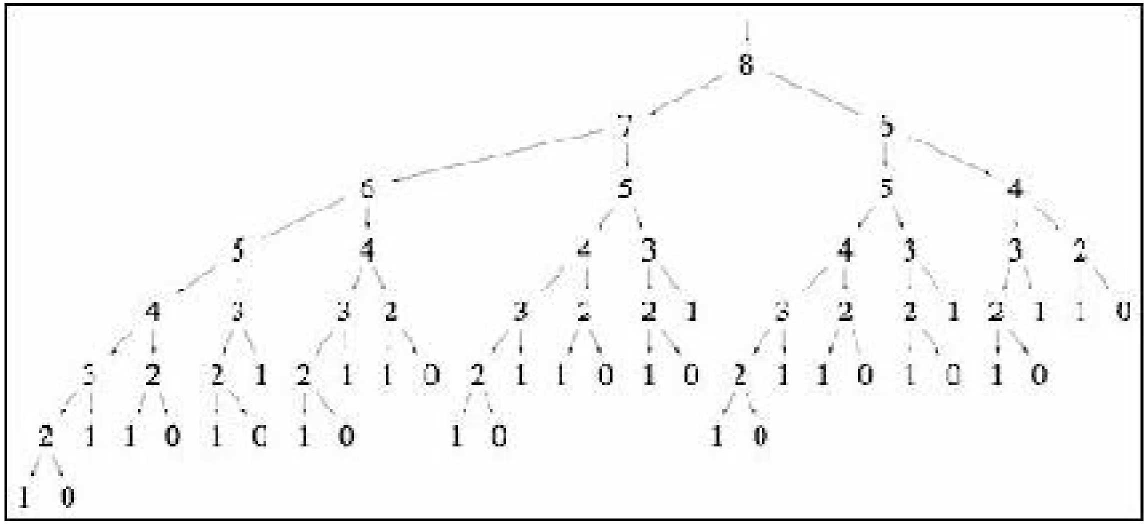
```
Sunt Lisa, si am 23 ani
name = Lisa
Sunt Lisa, si am 24 ani

Process finished with exit code 0
```

Memoization

```
import kotlin.system.*

fun main(args: Array<String>) {
val time = measureTimeMillis {
   println(fib(30))
}
println("Procesul a durat ${time}ms")
}
fun fib(k: Int): Long = when (k) {
   0 -> 1
   1 -> 1
   else -> fib(k - 1) + fib(k - 2)
}
```



Memoizare Fibbonaci cu map

```
import kotlin.system.*
fun main(args: Array<String>) {
   val k=8
   val time = measureTimeMillis {
      println("Fibbonaci($k)="+ memfib(k).toString())
   }
   println("Procesul a durat ${time}ms")
   println(map)
}
val map = mutableMapOf<Int, Long>()
fun memfib(k: Int): Long {
   return map.getOrPut(k) {
      when (k) {
         0 -> 1
         1 -> 1
         else -> memfib(k - 1) + memfib(k - 2)
      }
   }
}
```

si rezultatul executiei

```
Fibbonaci(8)=34
Procesul a durat 1ms
{1=1, 0=1, 2=2, 3=3, 4=5, 5=8, 6=13, 7=21, 8=34}

Process finished with exit code 0
```

Memoizare generalizată

```
fun <A, R> memoize(fn: (A) -> R): (A) -> R {
      val map = ConcurrentHashMap<A, R>()
      return { a ->
        map.getOrPut(a) {  fn(a)       }
      }
   }
```

și o versiune înbunătățită:

```
fun <A, R> Function1<A, R>.memoized(): (A) -> R {
      val map = ConcurrentHashMap<A, R>()
      return {
        a -> map.getOrPut(a) {  this.invoke(a)    }
      }
   }
 val memquery = ::query.memoized()
```

Alias de tip

```
typealias Cache = HashMap<String, Boolean>
```

- sau

```
fun process(exchange: Exchange<HttpRequest, HttpResponse>):
Exchange<HttpRequest, HttpResponse>
```

- se poate înlocui cu:

```
typealias HttpExchange = Exchange<HttpRequest, HttpResponse>
fun process2(exchange: HttpExchange): HttpExchange
```

- sau

```
typealias Width = Int
   typealias Length = Int
   typealias Height = Int
   fun volume(width: Width, length: Length, height: Height): Int
```

Either

```
sealed class Either<out L, out R>
     class Left<out L>(value: L) : Either<L, Nothing>()
    class Right<out R>(value: R) : Either<Nothing, R>()
```

Fold

```
sealed class Either<out L, out R> {
      fun <T> fold(lfn: (L) -> T, rfn: (R) -> T): T = when (this) {
        is Left -> lfn(this.value)
        is Right -> rfn(this.value)
      }
    }
```