ПРИЛОЖЕНИЕ Д

```
package program

import kotlinx.coroutines.*
import program.Log.logIn
import program.Log.logOut
import java.util.*


fun main() {
   fun Array<FloatArray>.toMatrixString() = joinToString("\n") {
      it.joinToString(" ") { String.format("% f", it) }
   }

   val a = arrayOf(
         floatArrayOf(1F, 2F, 3F),
         floatArrayOf(2F, 4F, 8F),
         floatArrayOf(1F, 4F, 9F)
   )

   val sequentialResult = SequentialInvMatrixProgram(a).execute()
   val sequentialLog = Log.content

   val parallelResult = ParallelInvMatrixProgram(a).execute()
   val parallelLog = Log.content

   val parallelResultWithDelay = ParallelInvMatrixProgram(a).execute(true)
   val parallelWithDelayLog = Log.content


   assert(Arrays.deepEquals(sequentialResult, parallelResult))
   assert(Arrays.deepEquals(parallelResultWithDelay, parallelResult))

   println("Последовательное выполнение:")
   println(sequentialLog)
   println()
   println("Параллельное выполнение:")
   println(parallelLog)
   println()
   println("Параллельное выполнение(с задержкой):")
   println(parallelWithDelayLog)
   println()

   println("матрица A")
   println(a.toMatrixString())
   println()
   println("матрица A^(-1)")
   println(parallelResult.toMatrixString())
}

class ParallelInvMatrixProgram(a: Array<FloatArray>) : InvMatrixProgram(a) {
   override val executeOrder: Operator by lazy {
      (
            (s1 and s2 before s3 before s4) and
                  (s5 and s6 before s7 before s8) and
                  (s9 and s10 before s11 before s12) before s13
            ) and
            (s14 and s15 before s16) and
            (s17 and s18 before s19) and
            (s20 and s21 before s22) and
            (s23 and s24 before s25) and
            (s26 and s27 before s28) and
            (s29 and s30 before s31) before s32 before s33
   }

   private inline val s1 get() = operators[0]
   private inline val s2 get() = operators[1]
   private inline val s3 get() = operators[2]
   private inline val s4 get() = operators[3]
   private inline val s5 get() = operators[4]
```

```
    private inline val s6 get() = operators[5]
    private inline val s7 get() = operators[6]
    private inline val s8 get() = operators[7]
    private inline val s9 get() = operators[8]
    private inline val s10 get() = operators[9]
    private inline val s11 get() = operators[10]
    private inline val s12 get() = operators[11]
    private inline val s13 get() = operators[12]
    private inline val s14 get() = operators[13]
    private inline val s15 get() = operators[14]
    private inline val s16 get() = operators[15]
    private inline val s17 get() = operators[16]
    private inline val s18 get() = operators[17]
    private inline val s19 get() = operators[18]
    private inline val s20 get() = operators[19]
    private inline val s21 get() = operators[20]
    private inline val s22 get() = operators[21]
    private inline val s23 get() = operators[22]
    private inline val s24 get() = operators[23]
    private inline val s25 get() = operators[24]
    private inline val s26 get() = operators[25]
    private inline val s27 get() = operators[26]
    private inline val s28 get() = operators[27]
    private inline val s29 get() = operators[28]
    private inline val s30 get() = operators[29]
    private inline val s31 get() = operators[30]
    private inline val s32 get() = operators[31]
    private inline val s33 get() = operators[32]
}

class SequentialInvMatrixProgram(a: Array<FloatArray>) : InvMatrixProgram(a) {
    override val executeOrder: Operator by lazy { operators.reduce { acc, o > acc before o } }
}

abstract class InvMatrixProgram(val a: Array<FloatArray>) {
    private val A = Array(3) { FloatArray(3) { 0F } }
    private val AT = Array(3) { FloatArray(3) { 0F } }
    private val invA = Array(3) { FloatArray(3) { 0F } }

    private var det = 0F
    private val t = FloatArray(18) { 0F }
    private val temp = FloatArray(3) { 0F }

    protected val operators: List<Operator> by lazy {
        listOf<suspend () > Unit>(
                { t[0] = a[1][1] * a[2][2] },
                { t[1] = a[1][2] * a[2][1] },
                { A[0][0] = t[0]  t[1] },
                { temp[0] = a[0][0] * A[0][0] },

                { t[2] = a[1][0] * a[2][2] },
                { t[3] = a[1][2] * a[2][0] },
                { A[0][1] = (t[2]  t[3]) },
                { temp[1] = a[0][1] * A[0][1] },

                { t[4] = a[1][0] * a[2][1] },
                { t[5] = a[1][1] * a[2][0] },
                { A[0][2] = t[4]  t[5] },
                { temp[2] = a[0][2] * A[0][2] },

                { det = temp[0] + temp[1] + temp[2] },

                { t[6] = a[0][1] * a[2][2] },
                { t[7] = a[0][2] * a[2][1] },
                { A[1][0] = (t[6]  t[7]) },

                { t[8] = a[0][0] * a[2][2] },
                { t[9] = a[0][2] * a[2][0] },
                { A[1][1] = t[8]  t[9] },

                { t[10] = a[0][0] * a[2][1] },
                { t[11] = a[0][1] * a[2][0] },
                { A[1][2] = (t[10]  t[11]) },
```

```
            { t[12] = a[0][1] * a[1][2] },
            { t[13] = a[0][2] * a[1][1] },
            { A[2][0] = t[12]  t[13] },

            { t[14] = a[0][0] * a[1][2] },
            { t[15] = a[0][2] * a[1][0] },
            { A[2][1] = (t[14]  t[15]) },

            { t[16] = a[0][0] * a[1][1] },
            { t[17] = a[0][1] * a[1][0] },
            { A[2][2] = t[16]  t[17] },

            {
               for (i in 0 until 3)
                  for (j in 0 until 3)
                     AT[i][j] = A[j][i]
            },

            {
               for (i in 0 until 3)
                  for (j in 0 until 3)
                     invA[i][j] = AT[i][j] / det
            }
      ).map { SimpleOperator(it) }
   }

   protected abstract val executeOrder: Operator

   fun execute(withDelay: Boolean = false): Array<FloatArray> {
      preExecute(withDelay)
      runBlocking { executeOrder.execute(withDelay) }

      return invA
   }

   protected open fun preExecute(withDelay: Boolean) {
      Log.reset()
      executeOrder.setDepth()
   }
}


sealed class Operator

class SequentialGroupOperator(val operators: List<Operator>) : Operator()
class ParallelGroupOperator(val operators: List<Operator>) : Operator()

class SimpleOperator(val action: suspend () > Unit) : Operator() {
   val operatorName = Log.run { "S$simpleOperatorNumber".also { simpleOperatorNumber++ } }
   var depth: Int = 1
}


suspend fun Operator.execute(withDelay: Boolean = false) {
   when (this) {
      is SimpleOperator > {
         logIn()
         if (withDelay) {
            delay(333)
         }
         action()
         logOut()
      }

      is SequentialGroupOperator > {
         operators.forEach {
            GlobalScope.async { it.execute(withDelay) }.await()
            if (withDelay) {
               delay(333)
            }
         }
      }
```

```
      is ParallelGroupOperator > {
         operators.map { GlobalScope.async { it.execute(withDelay) } }.awaitAll()
         if (withDelay) {
            delay(333)
         }
      }
   }
}

fun Operator.setDepth(depth: Int = 0) {
   when (this) {
      is SimpleOperator > this.depth = depth
      is ParallelGroupOperator > operators.forEach { it.setDepth(depth + 1) }
      is SequentialGroupOperator > operators.forEach { it.setDepth(depth + 1) }
   }
}

infix fun Operator.before(second: Operator) =
      when {
         this is SequentialGroupOperator && second is SequentialGroupOperator > SequentialGroupOperator
             (operators + second.operators)
         this is SequentialGroupOperator > SequentialGroupOperator(operators + second)
         second is SequentialGroupOperator > SequentialGroupOperator(listOf(this) + second.operators)

         else > SequentialGroupOperator(listOf(this, second))
      }

infix fun Operator.and(second: Operator) =
      when {
         this is ParallelGroupOperator && second is ParallelGroupOperator > ParallelGroupOperator(
             operators + second.operators)
         this is ParallelGroupOperator > ParallelGroupOperator(operators + second)
         second is ParallelGroupOperator > ParallelGroupOperator(listOf(this) + second.operators)

         else > ParallelGroupOperator(listOf(this, second))
      }


object Log {
   private val log = mutableListOf<String>()
   val content get() = log.joinToString("\n")

   var simpleOperatorNumber = 1


   fun SimpleOperator.logIn() {
      synchronized(log) {
         log += " ${"".repeat(2 * depth)}> $operatorName"
      }
   }

   fun SimpleOperator.logOut() {
      synchronized(log) {
         log += "<${"".repeat(2 * depth)} $operatorName"
      }
   }

   fun reset() {
      simpleOperatorNumber = 1
      log.clear()
   }
}
```