Problems to solve

1. Nonblocking rendezvous channel implementation **(done)**
   - Should be lock-free **(have idea)**
2. Closing and cancellation support **(have idea)**
3. Extending to buffered (both limited and unlimited) channel
4. Select expression
5. Broadcast/conflated
   - This is more about implememtation, out of the research scope

Rendezvous channel

Channel is a simple abstraction to transfer a stream of values between coroutines. Two basic operations form a channel: `send(val: T)` and `receive(): T`. The standard channel is conceptually very similar to `BlockingQueue` with one element. One key difference is that instead of a blocking put operation it has a suspending send, and instead of a blocking take operation it has a suspending receive.

Here is a basic channel interface:

```
interface ChannelKoval<E> {
    suspend fun send(element: E)
    fun offer(element: E): Boolean

    suspend fun receive(): E
    fun poll(): E?

    // Cannot invoke `send` on a closed channel,
    // `receive` can retrieve already waiting senders
    fun close(cause: Throwable? = null): Boolean
}
```

When a coroutine becomes blocked it is added to the waiting queue for this channel. Note that only one type of channel utilizers (sender or receiver) could be in the queue, and this condition should be checked during a `push` operation. Here is an example of such queue contract.

```
class WaitingQueue {

  /*
   * Atomically checks that the queue does not
   * have any receiver and adds sender to it,
   * otherwise polls the first receiver
   */
  fun addSenderOrPollReceiver(Coroutine c): Coroutine? {
    ...
```

```
  }

  /**
   * Atomically checks that the queue does not
   * have any sender and adds receiver to it,
   * otherwise polls the first sender
   */
  fun addReceiverOrPollSender(Coroutine c): Coroutine? {
    ...
  }
}
```

The main idea of such queue implementation is based on using segments in MS queue, so fewer allocations (and GC work) are required and a node could be cached for several send/receive operations. For this purpose, each node has a `_data` array (usually with size 16 or 32), in which coroutines with the elements to be sent are stored. In order to determine is this coroutine receiver or sender, a special `RECEIVER_ELEMENT` is stored for receivers. In order to manage this array, special `_deqIdx` and `_enqIdx` are used which specify indexes for next deque and enque operations respectively. Alike standard MS queue nodes, this implementation has `_next` field to the next node and MS queue algorithm is used for adding/removing these segment nodes.

Code for `Node` class:

```
private class Node {
  var _deqIdx = 0
  var _enqIdx = 0
  _data = AtomicReferenceArray<Any?>(SEGMENT_SIZE * 2)

  _next: Node? = null
}
```