Delivery 2

Task 1: Preliminaries (28p)

1.1: Implementing the TransactionQueue (7p)

- **Define a datastructure to hold transactions**
- **Implement functions of `TransactionQueue` in a thread-safe manner.** Solution:

```
class TransactionQueue {
 private var queue: Queue[Transaction] = Queue()

 private def mutateQueue[ReturnType](
     function: Queue[Transaction] => (pri
         ReturnType,
         Queue[Transaction]
     )
 ): ReturnType = {
   synchronized {
     val result = function(queue)
     this.queue = result._2
     result._1
   }
 }

 def pop: Transaction =
   mutateQueue[Transaction]((queue: Queue[Transaction]) => queue.dequeue)

 def isEmpty: Boolean = queue.isEmpty

 def push(t: Transaction): Unit =
   mutateQueue[Unit](queue => (Unit, queue.enqueue(t)))

 def peek: Transaction = queue.head

 def iterator: Iterator[Transaction] = queue.iterator
}
```

1.2 Account functions (14p)

- **`withdraw` removes an amount of money from the account.**

```
def withdraw(amount: Double): Either[Unit, String] =
    synchronized {
      if (amount < 0) {
        Right("Invalid amount")
      } else if (amount > balance.amount) {
        Right("Insufficient funds: Tried to withdraw " + amount + " from " + balance.amount)
      } else {
        balance.amount -= amount
        Left(Unit)
      }
    }
```

- **`deposit` inserts an amount of money to the account.**

```
def deposit(amount: Double): Either[Unit, String] =
  if (amount < 0) {
    Right("Invalid amount")
  } else {
    synchronized {
      balance.amount += amount
    }
    Left(Unit)
  }
```

- **`getBalanceAmount` returns the amount of funds in the account.**

```
def getBalanceAmount: Double =
synchronized {
  balance.amount
}
```

1.3 Eliminating Exceptions (7p)

- **`withdraw` should fail if we withdraw a negative amount or if we request a withdrawal that is larger than the available funds.**
- **`deposit` should fail if we deposit a negative amount.**
- **Both should be thread safe.**
- **Both should return an Either datatype and should not throw exceptions.**

Answer: See code snippets in 1.2.

Task 2: Creating the Bank (21p)

- **`addTransactionToQueue` creates a new transaction object and places it in the `transactionQueue`. This function should also make the system start processing transactions concurrently.**

```scala
def addTransactionToQueue(
      from: Account,
      to: Account,
      amount: Double
  ): Unit = {
      transactionsQueue.push(new Transaction(
          transactionsQueue,
          processedTransactions,
          from,
          to,
          amount,
          allowedAttempts
      ))

      val thread = new Thread {
          override def run():Unit = {
              processTransactions
          }
      }
      thread.start()
  }
```

- **`processTransactions` runs through the `transactionQueue` and starts each transaction one at a time. If a transactions' status is pending, push it back to the queue and recursively call `processTransactions`. Otherwise, the transaction has either failed, or succeeded, and should be put in the processed transac tions queue.**

```scala
private def processTransactions: Unit =
    while (!transactionsQueue.isEmpty) {
            synchronized {
                if (transactionsQueue.isEmpty) {
                    return
                }
                val transaction = transactionsQueue.pop
                val thread = new Thread {
                    override def run():Unit = {
                        transaction run
                        if (transaction.status == TransactionStatus.PENDING) {
                            transactionsQueue.push(transaction)
                            processTransactions
                        } else {
                        processedTransactions.push(transaction)
                        }
                    }
                }
                thread.start()
            }
    }
```

Task 3: Actually solving the bank problem (51p)

The goal of `doTransaction` is to transfer money safely, which means withdrawing money from one account and depositing it to the other account.

Each transaction is allowed to try to complete several times, indicated by the `allowedAttempts` variable. A transactions status is `PENDING` till it has either succeeded or used up all its attempts.

For the solution of this, we have the `Transaction` class:.

```scala
class Transaction(
    val transactionsQueue: TransactionQueue,
    val processedTransactions: TransactionQueue,
    val from: Account,
    val to: Account,
    val amount: Double,
    val allowedAttemps: Int
) extends Runnable {

  var status: TransactionStatus.Value = TransactionStatus.PENDING
  var attempt = 0

  override def run: Unit = {

    def doTransaction() = {
      attempt += 1
      val withdrawResult = from withdraw(amount)
      withdrawResult match {
        case Left(_) => {
          val depositResult = to deposit(amount)
          depositResult match {
            case Left(_) => {
              status = TransactionStatus.SUCCESS
            }
            case Right(string) => {
              println(string)
              from deposit(amount)
              if (attempt < allowedAttemps) {
                status = TransactionStatus.PENDING
              } else {
                status = TransactionStatus.FAILED
              }
            }
          }
        }
        case Right(string) => {
          if (attempt < allowedAttemps) {
            status = TransactionStatus.PENDING
          } else {
            status = TransactionStatus.FAILED
          }

        }
      }
    }

    synchronized {
      if (status == TransactionStatus.PENDING) {
        if (attempt < allowedAttemps) {
          doTransaction()
          Thread.sleep(50)
        } else {
          status = TransactionStatus.FAILED
          print("Too many attempts")
        }

      }
    }
  }
}
```