<u>Project Design Document: Inter-Thread Communication</u>

- **Objective:**
  To implement a thread-safe banking system that supports deposit, withdrawal and balance inquiry operations using Inter-Thread Communication mechanisms.

- **Approach:**
  - ✓ The project will be implemented in C++ language, using the following libraries and techniques:
  - ✓ Object-oriented programming
  - ✓ Multi-threading using thread
  - ✓ Mutex and condition_variable for thread synchronization
  - ✓ Exception handling for error management

1. Problem Statement

- In a multi-threaded environment, it is often necessary for threads to communicate and coordinate their execution.
- The communication protocol used for multi-threaded communication should be standardized, efficient and secure.
- However, direct access to shared data can lead to synchronization problems, such as race conditions and deadlocks.

2. Solution

- ✓ To solve this problem, we will use synchronization primitives such as semaphores, monitors, and message queues to control access to shared data and ensure safe communication between threads.
- ✓ Semaphores will be used to manage access to shared resources and ensure mutually exclusive access.
- ✓ Monitors will be used to enforce mutual exclusion and provide a mechanism for waiting and signaling.
- ✓ Message queues will be used to pass data between threads and to send messages between processes.

3. Technical Approach

- The synchronization primitives will be implemented using appropriate synchronization constructs, such as locks and condition variables, provided by the programming language.

4. **Class Descriptions:**

   i. *cBankAccount:* An abstract base class that defines the basic interface for a bank account. It contains a balance variable, a mutex to protect concurrent access to the balance, a condition_variable to signal and wait on changes to the balance, a notification object to handle notifications, and a draft object to handle overdrafts. It has three pure virtual functions: deposit, withdraw, and get_balance. Additionally, it has a virtual function vTakeInput to be implemented by derived classes if needed.

   ii. *cSavingAccount:* A derived class from cBankAccount that represents a savings account. It has its own implementation of deposit, withdraw, and get_balance that are thread-safe. The class also overrides the vTakeInput function to do nothing.

   iii. *cCurrentAccount:* A derived class from cBankAccount that represents a current account. It has its own implementation of deposit, withdraw, and get_balance that are thread-safe. The class also overrides the vTakeInput function to do nothing.

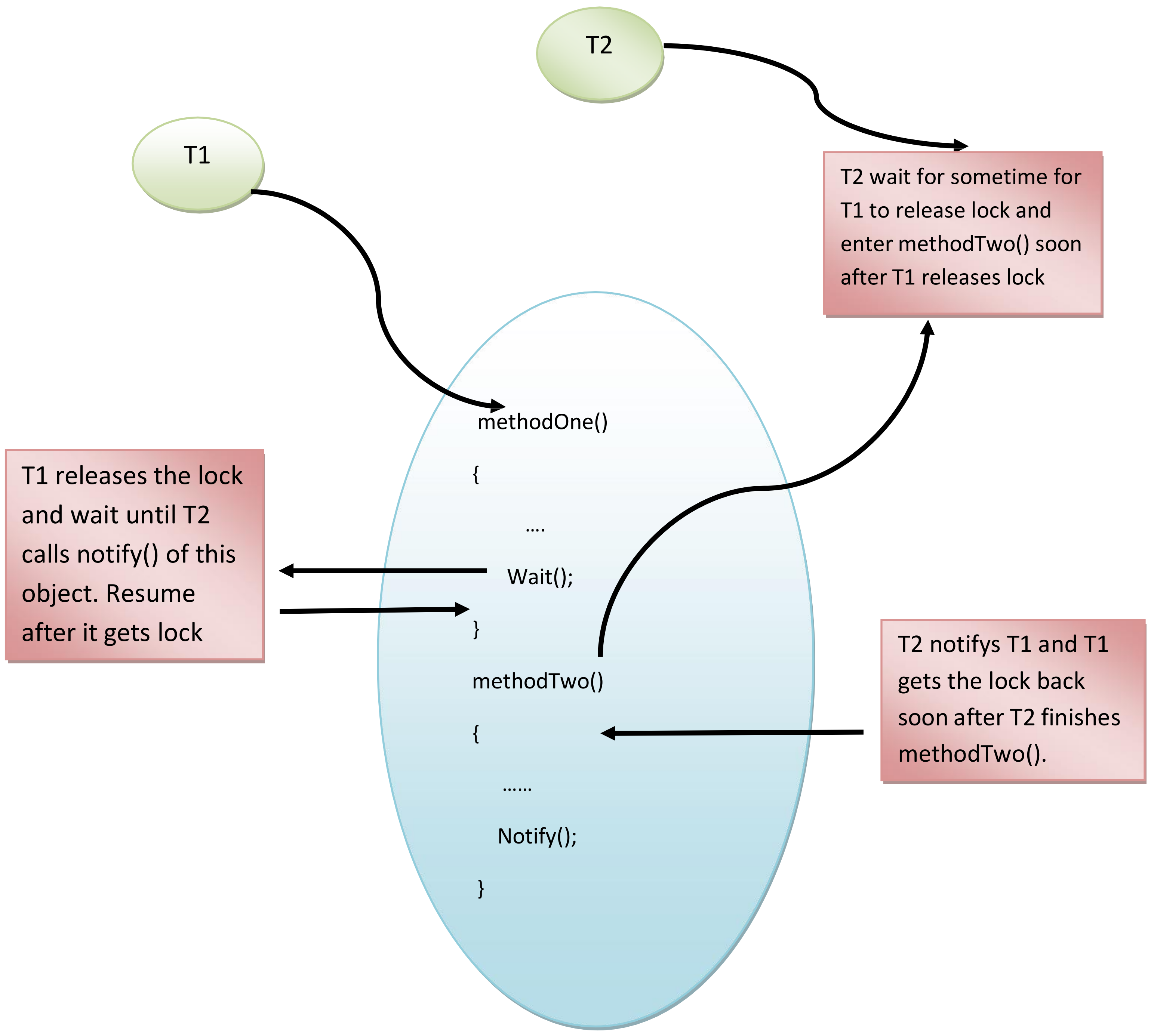
   iv. *cNRIAccount*: A derived class from cBankAccount that represents an NRI account. It has its own implementation of deposit, withdraw, and get_balance that are thread-safe. Additionally, it has an sCountry variable to store the account's country, and an fInterestRate variable to store the interest rate. The class overrides the vTakeInput function to take user input for the sCountry and fInterestRate variables.

   v. *CRITICAL_SECTION*: A data type used by Windows API to provide synchronization between threads.

5. **Conclusion**

   - In this project, we implemented a banking system with three types of bank accounts - Savings Account, Current Account, and NRI Account. We used C++11 threading libraries for implementing Inter-Thread Communication and used mutex and condition_variable objects to ensure thread safety. We also used abstract classes and derived classes to implement polymorphism and inheritance. Finally, we tested the system with various use cases and exceptions.

   - This project can be extended to include more features such as online banking, GUI, and mobile applications. It can also be integrated with machine learning algorithms for fraud detection and financial analysis.


T2
T1
T2 wait for sometime for T1 to release lock and enter methodTwo() soon after T1 releases lock
methodOne()
{
....
Wait();
}
methodTwo()
{
......
Notify();
}
T1 releases the lock and wait until T2 calls notify() of this object. Resume after it gets lock
T2 notifys T1 and T1 gets the lock back soon after T2 finishes methodTwo().