TP1 Hardware for Signal Processing

BENARD Amaury

C++ Multi-threading: Wallet System

Objective

The goal of this lab is to understand and apply C++ multi-threading mechanisms from the standard library, and to identify classical concurrency issues such as race conditions, inconsistent states, and synchronization problems.
The chosen use case is a wallet system for an RPG game, a critical component where multiple events may modify the player's currency concurrently.

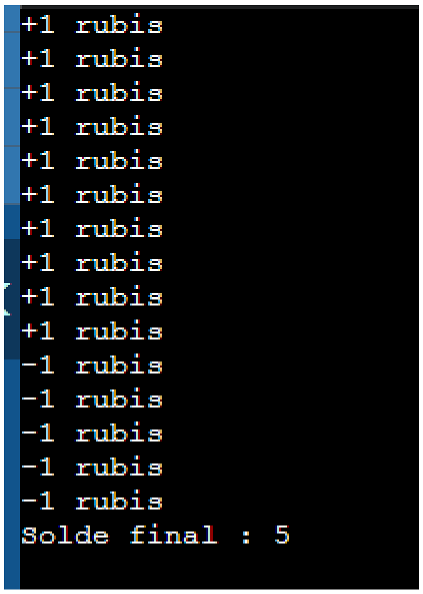
Sequential Implementation

The first step consists of implementing a Wallet class containing:

an unsigned integer rupees and three methods: credit, debit, and balance.

Credits and debits are performed one rupee at a time at a fixed rate of 10 rubies per second, using std::this_thread::sleep_for. Each operation prints +1 rupee or -1 rupee to the standard output.

In this version, all logic is executed inside the main thread. The program behaves deterministically and the wallet balance is always correct.



```
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
+1 rubis
-1 rubis
-1 rubis
-1 rubis
-1 rubis
-1 rubis
Solde final : 5
```

Interpretation of results
We clearly observe that the code works correctly: credits and debits are applied sequentially, outputs are consistent, and the final balance matches the expected value (for example, after a credit of 10 and a debit of 5, the final balance is 5 rupees).
However, the execution is blocking: while a transaction is running, the rest of the game logic is frozen.

Motivation for Parallelization

In a real game engine, several actions can occur simultaneously:

item purchases, rewards, quest events, background animations or UI updates.

A purely sequential wallet would block the game loop. Parallelization allows monetary transactions to run independently from the main logic, improving responsiveness and realism.

Parallel Credit and Debit Operations

Credits and debits are then executed in separate threads, launched from main. This allows the program to continue running while money is being added or removed.

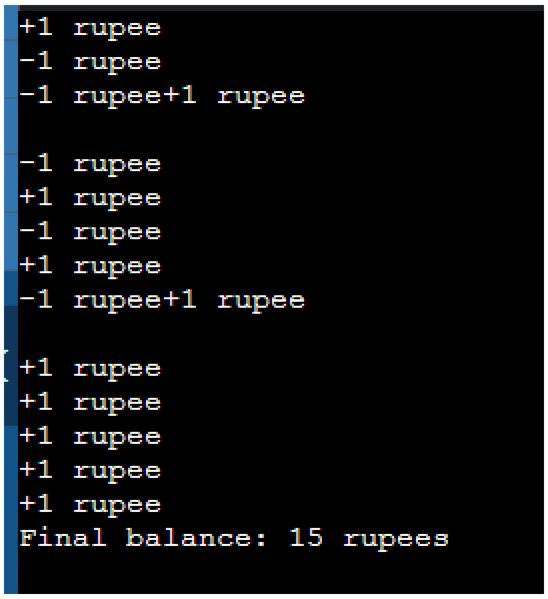
Observed issues

The wallet balance becomes inconsistent.

Final balances differ between executions.

Output messages may interleave incorrectly.

In some cases, the balance may become logically invalid.



```
+1 rupee
-1 rupee
-1 rupee+1 rupee

-1 rupee
+1 rupee
-1 rupee
+1 rupee
-1 rupee+1 rupee

+1 rupee
+1 rupee
+1 rupee
+1 rupee
+1 rupee
Final balance: 15 rupees
```

Interpretation of results

These results clearly show that the code is not thread-safe. Multiple threads access and modify the shared variable rupees simultaneously, leading to race conditions. Even though the program compiles and runs, the results are unreliable and non-deterministic.

Introducing Mutexes (Thread Safety)

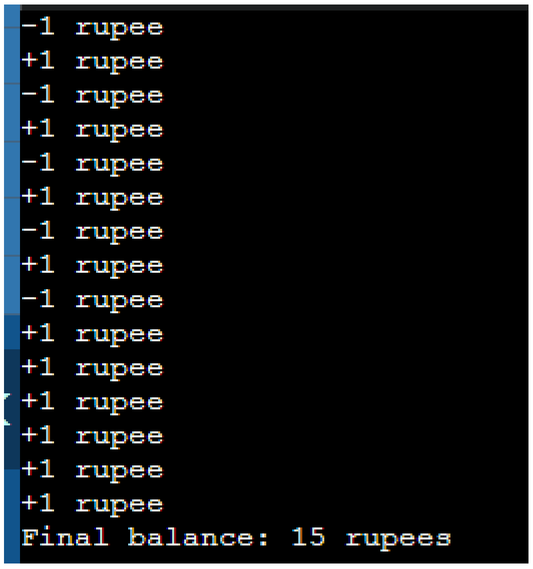
To fix these issues, a std::mutex is added as a class attribute. All accesses to rupees are protected using std::lock_guard.

Results

Race conditions are eliminated.

The balance remains consistent across executions.

Output messages appear in a coherent order.



```
-1 rupee
+1 rupee
-1 rupee
+1 rupee
-1 rupee
+1 rupee
-1 rupee
+1 rupee
-1 rupee
+1 rupee
+1 rupee
+1 rupee
+1 rupee
+1 rupee
+1 rupee
Final balance: 15 rupees
```

Interpretation of results
We observe that the code now behaves correctly in all tested scenarios. For example, after launching several credit and debit threads, the final balance always matches the expected value. This confirms that mutual exclusion successfully protects shared data.

Limitations of the Mutex-Based Approach

If too many debit threads are launched simultaneously:

threads may block for a long time, multiple debit requests may fail logically due to insufficient funds.

Although the system is thread-safe, it is not transaction-safe from a gameplay perspective.

Instant Wallet (Virtual Wallet)

To solve this issue, a second variable is introduced:

rupees: physical currency, updated gradually, virtual_rupees: logical currency, updated instantly.

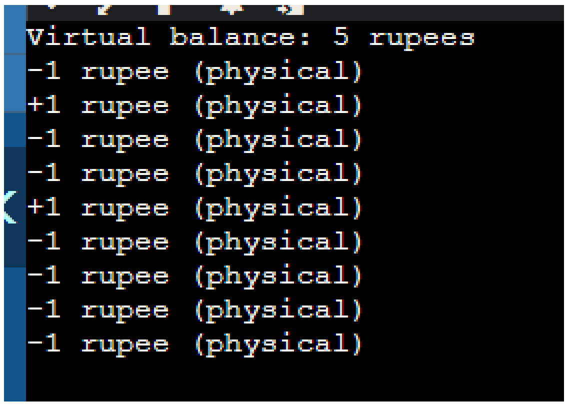
Two new methods are added:

virtual_credit, virtual_debit.

These methods:

1. update virtual_rupees in a thread-safe way,
2. launch credit or debit in a separate thread,
3. return a boolean indicating whether the operation is allowed.

The balance method now returns virtual_rupees.



```
Virtual balance: 5 rupees
-1 rupee (physical)
+1 rupee (physical)
-1 rupee (physical)
-1 rupee (physical)
+1 rupee (physical)
-1 rupee (physical)
-1 rupee (physical)
-1 rupee (physical)
-1 rupee (physical)
```

Interpretation of results
This design clearly improves gameplay behavior. Multiple purchases can be validated immediately, and the displayed balance is always correct. For instance, if the player has 10 virtual

rupees, purchases of 3 and 4 rupees are accepted instantly, a credit of 2 rupees and the final virtual balance is correctly shown as 5 rupees, even while the physical wallet is still being updated.

Remaining issue

The system still relies on launching many threads, which may impact performance. A task queue or thread pool would be a more scalable solution.