1 Introduction

In this lab, we were given the code base of a multi-threaded simulation program simulating production and sales between multiple entities. The job consists in implementing the logic governing the interactions between the various players, while paying attention to the critical sections of the program. The development of the resource management simulation has necessitated the implementation of several methods across the `Extractors`, `Wholesalers` and `Factories` classes to emulate the transactions. Our methodology emphasizes thread safety and transactional integrity ensuring that the simulation behaves predictably under concurrent conditions.

2 Concurrency

The program contains multiple methods that need to access the same resources of a class concurrently. This might affect the flow of the program as some accesses may be overwritten by another thread.

The `PcoMutex` class from the `pcosynchro` library is used to handle concurrency by protecting the so-called *critical section*. All methods that do read and/or write access to shared members of the class must acquire the mutual exclusion lock before processing the critical section to avoid the aforementioned concurrent access problem. The lock must be released on every exit path of a method.

Note that the code region that a lock protects may be extended to include some thread-safe statements. This practice is justified by the overall heavy cost of acquiring and releasing a lock.

3 Implementation

The `PcoMutex` object is used to guard against concurrent accesses to an instance. It is factorized as a private member of the `Seller` superclass.

The following code sections have been completed according to the lab instructions and using the previously detailed strategy.

3.1 The `trade` methods

The `trade` methods are implemented in the `Extractor`, `Wholesale` and `Factory` classes. The code is similar in each of the classes. It is designed to handle the transaction of a specified resource. A lock is first acquired to ensure thread safety, preventing simultaneous accesses that could lead to inconsistent state. Upon locking, the method verifies the validity of the trade, checking the requested quantity against the inventory and matching the item type to the extractor's resource. If the trade is invalid, it releases the lock and returns zero to indicate failure. For successful trades, it calculates the cost, adjusts the inventory and funds accordingly, and then releases the lock. This method provides a thread-safe means for extractors to participate in the market effectively.

The following code excerpt is taken from the `Extractor::trade()` method:

```
transactionMutex.lock();
if ( qty <= 0 || it != resourceExtracted || stocks[it] < qty) {
    transactionMutex.unlock();
    return 0;
}

int cost = qty * getMaterialCost();
money += cost;
stocks[it] -= qty;

transactionMutex.unlock();
return cost;
```

3.2 Wholesaler class

The `Wholesale::buyResources()` method is designed to purchase the resources of the game from other sellers. The implementation is straightforward: a lock protects the critical part of the transaction, and if the wholesaler has enough money, it buys from a random seller by decrementing the money and incrementing the stock. The mutex around the call `Seller:trade()` has been purposely place to prevent a deadlock situation. Indeed, if a given wholesaler tries to buy from a Factory and that specific factory tries to buy from the same wholesaler at the same time, a situation where both locks will never be released can occur.

```
void Wholesale::buyResources() {
    [...]

    transactionMutex.lock();
    if (price > money){
        transactionMutex.unlock();
        return;
    }
    transactionMutex.unlock();
    int bill = s->trade(i, qty); // Locking this section may cause a deadlock.
    transactionMutex.lock();
    if (bill > 0) {
        money -= bill;
        stocks[i] += qty;
    }
    transactionMutex.unlock();
}
```

3.3 Factory class

The `Factory::buildItem()` method simulates the processes of item assembly, resource consumption and labor allocation. If the factory can pay the employees, it produces an item, then pays the employees, and once the item is produced, increments its stocks by 1 unit. Note that the lock must be kept during the item assembly time, or it would risk creating contention.

```
void Factory::buildItem() {
    int salary = getEmployeeSalary(getEmployeeThatProduces(itemBuilt));

    transactionMutex.lock();
    // If we have enough money to pay salary
    if (money < salary) {
        transactionMutex.unlock();
        return;
    }

    // Produce 1 item
    for (ItemType item : resourcesNeeded) {
        stocks[item]--;
    }

    // Pay salary
    money -= salary;
    transactionMutex.unlock();

    // Temps simulant l'assemblage d'un objet.
    PcoThread::usleep((rand() % 100) * 100000);

    // Increment number of payed employee
    nbBuild++;

    // update item stock
    transactionMutex.lock();
    stocks[itemBuilt]++;
    transactionMutex.unlock();

    [...]
}
```

The `Factory::orderResources()` method maintains the supply of raw materials. It will order the resource the factory has the least of by looking at the lowest quantity of each resource in the stocks. For simplicity, only 1 resource is ordered at a time. The method seeks if any wholesaler can trade the needed resource. If one of the wholesalers possesses it, the resource is purchased by decrementing the money and incrementing the stock. The `std` containers (such as the one used to hold the stock) offer no thread-safe guarantee when read and write may happen at the same time. Note that the aforementioned deadlock situation between the factory and the wholesaler could also be avoided by refactoring this method.

```
void Factory::orderResources() {
    transactionMutex.lock();
    // Prioritizing resources the factory has the least of.
    ItemType resourceToBuy = std::min_element(stocks.cbegin(), stocks.cend(),
                                              [](const auto& l, const auto& r) {
                                                  return l.second < r.second;
                                              })->first;

    // Iterate over available wholesalers
    for (Wholesale* ws : wholesalers) {
        if (ws->getItemsForSale().contains(resourceToBuy)) {
            int cost = getCostPerUnit(resourceToBuy);
            if (cost > money)
                break;
            cost = ws->trade(resourceToBuy, 1);
            if (cost == 0)
                continue; // Trade did not work. Look at another wholeseller.
            stocks[resourceToBuy]++;
            money -= cost;
            break;
        }
    }
    transactionMutex.unlock();

    // Temps de pause pour viter trop de demande
    PcoThread::usleep(10 * 100000);
}
```

3.4 Extractor class

The `Extractor::run()` method is a continuous loop that simulate the extraction process, which includes paying the miner's salary and adding the extracted resource to the stockpile.

The call to `PcoThread::usleep()` is intentionally left unlocked because protecting this with a mutex would create unnecessary contention, as no shared resources are being accessed or modified during this time. Furthermore, since the sleeping period is meant to simulate the time taken by a miner to extract resources, it should not be subject to synchronization constraints.

```
void Extractor::run() {
    [...]

    int minerCost = getEmployeeSalary(getEmployeeThatProduces(resourceExtracted));
    transactionMutex.lock();
    if (money < minerCost) {
        transactionMutex.unlock();
        PcoThread::usleep(1000U);
        continue;
    }

    money -= minerCost;
    transactionMutex.unlock();
    PcoThread::usleep((rand() % 100 + 1) * 10000);
    nbExtracted++;

    transactionMutex.lock();
    stocks[resourceExtracted] += 1;
    transactionMutex.unlock();

    [...]
}
```

4 Program exit

When the user clicks on the *Close* button of the window, a signal that ultimately calls the `Utils::endService()` method is propagated. This exit-handling routine then executes a loop to call the `PcoThread::requestStop()` method of each instance.

```
// Ask the threads to stop
for (auto& thread : threads)
    thread->requestStop();
```

The `PcoThread::requestStop()` method requests the threads to stop by setting a boolean member that acts as a flag. This flag is the checked in the `run()` method of each `Seller` instance by calling the `PcoThread::stopRequested()` method. Once the flag is set, the instances will all stop their main run loop once their current task is finished. The program is then able to terminate gracefully.

Example taken from the `Factory::run()` method:

```
  while (!PcoThread::thisThread()->stopRequested()) {
        if (verifyResources()) {
            buildItem();
        } else {
            orderResources();
        }
        [...]
    }
```

Note that the simulation sometimes hangs for a little while before exiting due to the fact that the threads must all finish their current task before exiting.

5 Tests

The following tests have been performed to ensure the proper functioning of the program:

- Running the simulation and observing the behavior of the entities via the graphical interface - Verify that the game successfully terminates and shows the expected values in the popup - Voluntarily introduce inconsistencies by removing calls to `PcoMutex::lock()` to ensure that a section was indeed critical - Voluntarily introduce deadlocks by removing calls to `PcoMutex::unlock()` to ensure that mutual exclusion is indeed necessary - Modify the `usleep()` values to speed up the simulation. Note that the simulation ceases to function properly passed a certain speed threshold, leading to freezes during the execution and/or the inability to terminate the program properly. The causes of this behavior are unknown. - Modify the number of entities to make the simulation more complex - Modify various thresholds (such as the randomly generated values) to observe the effects on the simulation - Modify the number of cores available to the program using `taskset`

6 Conclusion

In summary, the features of this lab simulation's implementation are indicative of careful consideration of thread safety and transaction integrity. The designed `trade`, `buyResources`, `buildItem`, and `orderResources` methods function as intended, facilitating the simulation of economic interactions among extractors, wholesalers, and factories. By adhering to the principles of concurrent programming, we have ensured that the system operates consistently under various conditions.

We trust that the modifications put forth in this report meet the practical requirements of the simulation and contribute to the educational goals of the project.