1 Introduction

The following report describes the conception and implementation of a barbershop simulation designed to practice the Mesa monitor paradigm. The simulation involves a barber and multiple clients, each represented by a separate thread. The barber and clients operate independently but require careful synchronization to ensure correct behavior. The simulation is implemented in C++ using the PcoSynchro library for synchronization primitives and the Qt framework for the graphical user interface.

2 Analysis

Multiple threads representing clients and a barber operate independently but require careful synchronization to ensure correct behavior. The core challenge lies in managing the interactions between these threads, which include clients entering and exiting the salon, waiting for their turn, and the barber providing services and managing idle time. The barber and clients are modeled as separate threads, each executing its own logic. The salon operates with limited capacity, necessitating a mechanism to control client access and service order.

The Mesa monitor concept must be used to synchronize the barber and client threads. It involves condition variables, a synchronisation primitive used together with a mutex to block threads from accessing shared resources. Several condition variables are used to signal different states such as the availability of the barber, a client waiting for a haircut or the completion of a haircut. A mutex lock must be utilized to protect shared variables and ensure atomicity in operations. Threads acquire the lock before checking or modifying shared states and release it during the wait to avoid deadlock scenarios.

The incorporation of animations to reflect real-time actions add a layer of complexity to the synchronization scheme. Animations in the simulation are blocking, which means that the executing thread is held up until the animation completes. Because the mutex is systematically unlocked before the animation and re-locked afterwards, this mechanisms disrupts the synchronization flow, as the order of the execution is different before and after the animation.

3 Conception

3.1 Capacity and waiting seats management

It was decided that the salon's capacity should reflect the total number of clients that it can treat at any given time, which includes the number of waiting seats plus one additional seat for the barber chair. This approach ensures that the salon operates at full capacity, maximizing resource utilization.

As for the management of the waiting seats, we implemented a rotation system that does not strictly adhere to a first-come, first-sit basis. Instead, clients entering the salon may occupy any available waiting seat, determined by a rotating index. This index is managed using a modulo operation, which cycles through the waiting seats. It is important to note that this approach means clients might not always sit in the first free chair as no specific seating order was mandated.

3.2 Barber Cycle

The Barber class encapsulates the behavior of a barber in the simulation. The barber's responsibilities are to provide services to clients and manage idle time. The operational cycle is as follows:

```
1   void Barber::run() {
2       while (_salon->isInService() || _salon->getNbClient() > 0){
3           _interface->consoleAppendTextBarber("Je suis pret a accueillir un client");
4           if (_salon->getNbClient() == 0) {
5               _interface->consoleAppendTextBarber("Pas de client, je vais dormir");
6               _salon->goToSleep();
7               continue;
8           }else{
9               _interface->consoleAppendTextBarber("J'appelle le client suivant");
10              _salon->pickNextClient();
11          }
12
13          _interface->consoleAppendTextBarber("J'attends que le client vienne sur la chaise");
14          _salon->waitClientAtChair();
15          _interface->consoleAppendTextBarber("Je vais coiffer le client");
16          _salon->beautifyClient();
17      }
```

```
18          _interface->consoleAppendTextBarber("La journee est terminee, a demain !");
19      }
```

Listing 1: The barber operational cycle

- **Checking for Clients**: The barber continuously checks if clients are present in the salon. This check determines the barber's subsequent actions.
- **Sleeping when Idle**: If no clients are present, the barber goes to sleep. The `goToSleep()` method involves waiting on a condition variable until a client arrives and wakes the barber up.
- **Serving Clients**: When clients are present, the barber calls `pickNextClient()` to select the next client for service. The barber then waits for the client to be ready on the working chair (`waitClientAtChair()`) before proceeding with the haircut (`beautifyClient()`).

This cycle repeats until the salon is no longer in service, at which point the barber concludes the workday.

3.3 Clients Cycle

The Client class represents the behavior of clients visiting the salon. Clients will attempt to access the salon and receive a haircut :

```
1   void Client::run() {
2       while (_salon->isInService()) {
3           _interface->consoleAppendTextClient(_clientId, "Je regarde s'il y a de la place dans le
                salon");
4           if (!_salon->accessSalon(_clientId)) {
5               _interface->consoleAppendTextClient(_clientId, "Le salon est plein, je reviens plus
                    tard");
6               _salon->walkAround(_clientId);
7               continue;
8           }
9           _interface->consoleAppendTextClient(_clientId, "Je vais sur la chaise du barbier");
10          _salon->goForHairCut(_clientId);
11          _interface->consoleAppendTextClient(_clientId, "J'attends que mes cheveux repoussent");
12          _salon->waitingForHairToGrow(_clientId);
13      }
14      _interface->consoleAppendTextClient(_clientId, "Le salon est ferme... Zut !");
15      _salon->goHome(_clientId);
16  }
```

Listing 2: The clients operational cycle

- **Attempting Salon Access**: Each client checks if there is available space in the salon. If the salon is full, the client will walk around and retry after some time.
- **Receiving Haircut**: Once access to the salon is granted, the client either goes to the barber's chair for a haircut, or goes to a waiting chair. This involves waiting for the barber to complete the haircut, and the management of a ticketing system to keep the track of the order of arrival.
- **Post-Haircut**: After the haircut, the client simulates waiting for hair to grow before potentially attempting another visit.

The client's cycle continues as long as the salon is in service. Once the salon closes, the barber takes care of the remaining clients but doesn't accept any new ones. Once all the clients have been treated and gone home, the barber finishes his day.

3.4 Guarantee the client order

In the initial stages of designing the barbershop simulation, a queue data structure was considered to manage the order in which clients would be served, adhering to the FIFO principle. However, we realized that the waiting process in the Mesa monitor essentially forms a "virtual queue", where clients wait for their turn based on the condition variables. This insight led to the adoption of a simpler ticketing system.

The ticketing system works by assigning a unique ticket number to each client as they attempt to access the salon. This ticket number determines the client's position in the service order. Clients check if their ticket number matches the next number to be served (`_nextTicket`). If not, they wait in the `accessSalon` method, effectively queuing in a virtual waiting line. Once the barber completes a haircut, the `_nextTicket` number is incremented, and the corresponding client is notified to proceed for their service. This method ensures that clients are served in the exact order of their arrival, maintaining the FIFO. This ticketing approach takes advantage of the existing synchronization of the Mesa monitor without the need for an explicit queue data structure.

3.5 Mesa monitor implementation

The Mesa monitor paradigm is characterized by the use of condition variables along with a mutex to safely access and modify shared state. A mutual exclusion lock ensures atomic access to the shared state of the salon, such as the number of clients inside, the status of the barber chair, and the ticketing system.

Several condition variables are used to manage different synchronization aspects:

- `_barber`: Signaled when a client needs to wake up the sleeping barber.
- `_client`: Notified when it's time for the next client to proceed for their haircut.
- `_clientOnWorkingChair`: Used to synchronize the client's movement to the barber chair.
- `_beautifyDone`: Signaled when the barber completes a haircut.
- `_clientEnteringSalon`: Used for rare cases when the barber is awake, and a client is entering the salon.

3.6 Program termination

The simulation uses a shared boolean flag `_inService` to indicate whether the salon is still operational. When the program needs to end, this variable is set to false. If the barber is asleep (indicated by `_barberSleeping`), the salon must wake him up to process the termination signal. This is achieved by signaling the `_barber` condition variable. Upon waking up or after completing a haircut, the barber checks the `_inService` flag. If the flag is set to false, the barber concludes his work by treating the remaining clients. Clients, in their execution cycle, continuously check the `_inService` flag before attempting to access salon. If the salon is no longer in service, they go home and then exit their loop.

The main application ensures that all client and barber threads are joined after setting the `_inService` flag to false. This joining process waits for each thread to finish its execution, ensuring that all threads conclude properly before the program terminates.

4 Automated Testing of Barber and Clients Sequences

To ensure the correctness of the simulation, automated tests targeting the behavioral sequences of both the barber and the clients were implemented. The class can be found in the `utils/tests.cpp` file. These tests are executed at the end of every simulation where they assert that the sequence of events observed in the simulation matches the patterns passed in the instructions.

4.1 Defining Valid Sequences

The testing strategy involves the definition of "valid sequences" for both the barber and the clients. These sequences are chains of actions that the barber and clients are expected to perform in a specific order under normal operating conditions. These valid sequences are defined as arrays of enums, representing distinct actions. By using enums, we provide a readable representation of each action in the sequences.

4.2 Recording Sequences During Execution

As the simulation runs, each action taken by the barber or a client is recorded into a vector. This is accomplished by placing "push back" operations within the PcoSalon class's methods, which capture the sequence of actions as they occur in real-time. These vectors serve as the actual sequences to be tested against our predefined valid sequences at the end of the simulation.

4.3 Analyzing and Validating the Sequences

Once the simulation ends, the testing phase begins. Here, we analyze the recorded sequences to ensure they align with our predefined valid sequences. The analysis involves segmenting the sequences and then systematically comparing each segment against the valid sequences.

For the barber, whose sequences can vary based on situational factors (such as whether the barber was asleep or awake at the start of the sequence), the segmentation is based on the consistent ending action in each cycle. For clients, whose actions form repeatable loops, the segmentation is based on the start of each loop.

If a recorded sequence matches any of the valid sequences, it is deemed correct. If not, the sequence is flagged as incorrect, indicating a potential issue in the simulation logic or sequence implementation.

5 Manual testing

The automated tests are not sufficient to ensure the correctness of the simulation. Independent tests were also done to assert that the simulation behaves as expected in a given situation. All the checks described in these sections were manually operated by running the simulation and observing the behavior of the barber and the clients. All tests have successfully passed.

5.1 Synchronization tests

The following tests were done to assert that the synchronization between the barber and the clients in a given situation is correct :

- With more clients (8) than waiting seats (2)
- With more waiting seats (8) than chairs (2)
- With identical number of waiting seats and clients (4)
- With no waiting seats and a single client
- With no waiting seat and multiple clients (8)
- With no waiting seats and no clients
- With multiple waiting seats (4) and a single client
- With a single waiting seat and multiple clients (20)
- With a lot of clients (30) and waiting seats (20)

5.2 Program termination tests

The following tests were done to assert that the program terminates correctly in a given situation :

- While a client enters the salon
- While a client seats on a waiting chair
- While a client have his/her hair cut and other clients are waiting
- While a client have his/her hair cut and nobody is waiting
- While a client wait for his/her hair to grow
- While a client go for a walk
- While the barber is asleep
- While the barber is waking up

5.3 Client entrance tests

The following tests were done to assert that the client entrance in the salon is correct in a given situation :

- When there is a single client
- When there are multiple clients (8)

5.4 Hair cut order tests

The following tests were done to assert that the hair cut order is correct in a given situation :

- When there are multiple clients (20) and waiting seats (10)

5.5 Miscellaneous tests

The following tests were done to assert that the simulation is correct in a given situation :

- The clients go for a walk only when the salon is full
- The clients go home only when the salon is closed
- The barber goes to sleep when the salon is empty
- When the barber is asleep, the first client fully in the salon wakes him up
- The barber stops working when the salon closes and is empty

5.6 Edge case

Typically, at the start of the simulation, the barber enters a sleep state due to the absence of clients in the salon. However, there are infrequent situations where the barber is already awake when the first client enters the salon. In such a scenario, the barber must wait until the client has fully entered the salon and the entrance animation has completed before selecting the next client. If the barber were to select the next ticket prematurely, it would lead to a deadlock as no client has yet taken a ticket. To account for this edge case in our tests, we introduced a one-second delay before the main loop in the `barber::run` method. It's worth noting that when this edge case occurs, we've opted to seat the first client directly on the barber chair, rendering the waiting chair unnecessary. This decision also facilitates the entry of another client into the salon.

6 Conclusion

In conclusion, the simulation successfully implemented the Mesa monitor and ticket lock synchronization mechanisms to manage multiple threads accessing shared resources and needing to coordinate their actions. The simulation was validated by both manual and automated tests to ensure the correctness of the synchronization and the simulation logic.

Appendix

7 Appendix name