'Punta Cana Festival Simulation'

OS Project 2024

Group members: Jaime Berasategui Cabezas, Maria Evrydiki Kanellopoulou, José Urgal Saracho

TABLE OF CONTENTS

THE REAL-WORLD SCENARIO SIMULATED

The real-world scenario we are simulating is the worldwide known and admired ‘The Paradise’ festival in Punta Cana. The simulation is designed to model the dynamic and interactive atmosphere of a festival, complete with various activities, personnel roles, and attendee behaviors that mimic real-world scenarios.

This festival simulation integrates various core aspects of festival life, creating an immersive virtual experience that mirrors the real-world dynamics attendees encounter. At the heart of the festival are the bar and food truck areas, vital hubs where attendees like Jose Garcia Escandón can savor a variety of drinks and meals. The simulation ensures that these services are not just background elements but active parts of the experience.

The festival's infrastructure also includes bathroom facilities, separated by gender, to cater to the needs of attendees. The simulation intelligently links the consumption of beverages with the need to use these facilities. The more an attendee drinks, the more frequently they may need to visit the bathroom. This realistic feature underscores the practical aspects of festival-going and enhances the realism of the simulation.

Entertainment is provided on three main stages, each dedicated to a different genre, ensuring a variety of performances that cater to diverse musical tastes. Artists perform at scheduled times, creating a live event atmosphere where attendees like Jose can plan their itinerary around the performances of their favorite artists like Bad Bunny or Ariana Grande. Each stage hosts one artist at a time, allowing for simultaneous performances across the festival, thus offering attendees the chance to experience multiple shows. The festival simulation crafts a vivid, interactive panorama, blending entertainment, sustenance, and social interactions to deliver a truly memorable virtual event.

IMPLEMENTATION

Our implementation approach is based on Object Oriented Programming, where different classes are created to serve each activity that we would like to simulate as part of the Punta Cana Festival. The following section will give a brief description of all the classes in the simulation, in order to enhance understanding of the dynamics and interactions.

1. **Persons' classes:**

   Person: Base class for any individual at the festival
   - **Attributes:**
     - id: Unique identifier for the person.
     - age: Age of the person.

   Attendee:
   - **Attributes:**
     - Inherits id and age from Person.
     - ticket: Type of ticket held by the attendee.
     - total_drinks, total_foods, total_treatments, total_bathroom_visits, total_stage_visits: Counters for various activities.
     - gender: Gender of the attendee.
     - activities: List of possible activities attendee can engage in.
     - has_free_ticket: Boolean indicating if the attendee has a free ticket.
     - needs_emergency and needs_bathroom: Probabilities influencing certain needs based on activities.
   - **Methods:**
     - pass_check(): Validates entry based on the ticket.
     - decide_to_leave(): Determines if the attendee leaves based on time and activities.
     - place_drink(), place_food(): Methods to order drinks and food.
     - go_to_stage(), go_to_bathroom(), go_to_emergency(): Actions to attend a stage, use the bathroom, or visit the emergency truck.
     - do_activities(): Manages the random selection of activities and their execution.

   Barista and Cook:
   - **Attributes:**
     - name: The name of the barista or cook.
     - bar or food_truck: Reference to the bar or food truck where they work.
   - **Methods:**
     - run(): The main activity loop where the barista or cook processes orders. They check for new orders, prepare them, and notify the attendee when the order is ready. This method runs continuously in a thread as long as the festival is running.

Security Stuff:

- **Attributes:**
  - name: The name of the security staff member.
  - entrance: A reference to the entrance area where they operate.
- **Methods:**
  - run(): Checks attendees as they come in. If an attendee has a valid ticket, they are allowed inside and notified. This runs in a loop and each check simulates a delay to represent the time taken for security checks.

Artist:

- **Attributes:**
  - name: The name of the artist.
  - genre: Musical genre of the artist.
  - set_duration: Duration of the artist's set.
  - stage: Reference to the stage where they perform.
  - stage_index: The specific stage index assigned to the artist.
- **Methods:**
  - run(): Controls the artist's performance, ensuring they only perform when the stage is available (using a lock to prevent overlaps). The artist performs for a set duration, then finishes.

Doctor:

- **Attributes:**
  - name: The name of the doctor.
  - emergency_truck: Reference to the emergency medical services available at the festival.
- **Methods:**
  - run(): A loop where the doctor waits for patients, treats them, and sends them back to the festival with advice or warnings as necessary.

2. **Services classes:**

Entrance: Manages the entrance where security checks are conducted.

- **Attributes:**
  - security_count: Number of security staff at the entrance.
  - attendees: A queue of attendees waiting to be checked.
  - securities: A list of SecurityStaff instances.
- **Methods:**
  - add_check(attendee): Adds an attendee to the queue for security checking.
  - get_next_attendee(): Retrieves the next attendee from the queue for a security check.
  - start(): Starts all security staff threads to begin processing attendees at the entrance.

Bar:

- **Attributes:**
  - barista_count: The number of baristas working at the bar to serve drinks to attendees.
  - menu: An instance of the Menu_Bar class, which lists available drinks and their details.
  - orders: A list to manage drink orders placed by attendees.
  - baristas: A list of Barista instances, each responsible for serving drinks.
  - orders_lock: A threading lock to ensure that access to the orders list is thread-safe.
- **Methods:**
  - add_order(order): Adds a new drink order to the orders list. This method uses orders_lock to synchronize access to the list in a multi-threaded environment.
  - get_next_order(): Retrieves and removes the next order from the orders list for processing. This method is also synchronized using orders_lock.
  - start(): Starts the threads for all baristas, enabling them to begin processing orders. It ensures all barista threads are joined at the end of their execution.

Food Truck:

- **Attributes:**
  - cook_count: Number of cooks working in the food truck to prepare food orders.
  - menu: An instance of the Menu_FoodTruck class, detailing the food items available for order.
  - orders: A list to manage food orders placed by attendees.
  - cooks: A list of Cook instances, each responsible for preparing and serving food.
  - orders_lock: A threading lock used to manage access to the orders list safely in a multi-threaded environment.
- **Methods:**
  - add_order(order): Adds a new food order to the orders list. This method is synchronized with orders_lock to ensure thread safety.
  - get_next_order(): Retrieves the next order from the orders list for processing by the cooks. It is synchronized to prevent concurrent access issues.
  - start(): Initiates all cook threads allowing them to start processing food orders. Ensures all threads are properly joined after the festival ends or the service stops.

Stage:

- **Attributes:**

    - num_stages: Number of stages at the festival.
    - artists_info: Information about artists performing, such as name, genre, and performance duration.
    - stages: A list of dictionaries, each representing a stage equipped with a lock for concurrency control.
    - current_performers: A list tracking which artist (if any) is currently performing on each stage.
    - artists: A list of Artist instances assigned to perform on the stages.
- **Methods:**
    - start_show(): Starts all artist performances and ensures they perform sequentially or simultaneously as scheduled without overlap, managed by threading and locks.
    - get_current_performer(stage_index): Retrieves the current performer on a specified stage, if any.

Emergency Truck: Provides medical services at the festival.

- **Attributes:**
    - doctors_count: Number of doctors available in the emergency truck.
    - doctors: List of doctor instances working in the emergency truck.
    - patients: Queue of patients waiting for medical attention.
- **Methods:**
    - admit_patient(patient): Adds a patient to the queue for medical attention.
    - get_next_patient(): Retrieves the next patient needing care.
    - start(): Starts all doctor threads to begin treating patients.

Bathroom: Manages bathroom facilities, ensuring attendees can relieve themselves, especially after consuming beverages.

- **Attributes:**
    - stalls_per_gender: Specifies how many stalls are available per gender.
    - persons: Queues for males and females waiting to use the bathroom.
    - stalls: Actual bathroom stalls organized by gender.
- **Methods:**
    - request_use(person): Adds a person to the queue based on their gender.
    - get_next_person(gender): Retrieves the next person in line for the bathroom.
    - start(): Starts all stall threads, making them available for use.
    - stop(): Stops all stall threads when the festival ends.

3. **Additional classes to facilitate implementation:**

Menus & Menu Items: Managing the food and drink offerings available to attendees. These classes encapsulate the details and operations related to the menus at the festival's bars and food trucks respectively.

Ticket type: Designed to manage and define the types of tickets available to the attendees. This class encapsulates the ticket type as a singular attribute.

Festival Database class: Handles all database operations related to storing and retrieving festival data.

- **Attributes:**
  - conn: Database connection object.
  - cursor: Cursor object for executing SQL commands.
- **Methods:**
  - create_database(database): Creates the festival database if it does not exist.
  - create_attendees_table(), create_orders_table(): Create tables for storing data about attendees and their orders.
  - insert_attendee(attendee), insert_order(order): Insert records into the respective tables.
  - clear_orders_table(): Clears the orders table for initialization or cleanup.
  - close(): Closes the database connection and cursor, ensuring a clean shutdown of database operations.

4. **Final Festival simulation class:**
   - **Attributes:**
     - num_attendees, num_baristas, num_cooks, num_stalls, num_security, num_doctors, num_stages: These parameters define the scale of the festival, specifying the number of attendees, staff, facilities, and stages.
     - attendees: A list of Attendee instances representing all festival-goers.
     - bars, food_trucks: Lists containing instances of Bar and FoodTruck classes, respectively.
     - bathroom: An instance of the Bathroom class managing bathroom facilities.
     - stage: An instance of the Stage class managing all stage performances.
     - emergency_truck: An instance of the EmergencyTruck class providing medical services.
     - entrance: An instance of the Entrance class managing the entry of attendees.
     - festival_running: A boolean flag indicating if the festival is currently active.
     - all_orders: A list collecting all orders placed during the festival for later processing or analysis.
     - festival_db: An instance of the FestivalDatabase class for handling database operations related to the festival.
   - **Methods:**
     - collect_order(order): Adds an order to the all_orders list for processing.

- store_all_orders(): Stores all collected orders in the database, used for record-keeping and analysis post-festival.
- start(): Initiates the festival by starting all necessary threads for services like the entrance, bars, food trucks, bathroom, emergency services, and stage performances. This method also handles the lifecycle of festival activities and ensures all threads conclude properly.

Each attendee's experience is simulated as a distinct thread, starting from their arrival at the entrance where their ticket is checked by security. Once admitted, the attendee enters a continuous loop where they randomly select activities such as eating, drinking, watching performances, using the bathroom, or seeking emergency care—each represented by a specific interaction with festival infrastructure (bars, food trucks, stages, bathrooms, and emergency trucks). Their choices and frequency of activities are influenced by a dynamic model that considers their consumption and engagement, affecting probabilities like needing the bathroom or medical attention. Throughout their stay, attendees may decide to leave based on how long they've been at the festival and their interactions, with each activity potentially being their last. This decision-making loop continues until the attendee chooses to leave or the festival ends, at which point their thread is concluded and all activities are wrapped up. This threaded approach allows the simulation to dynamically handle multiple attendees simultaneously, each engaging in activities independently yet interacting with shared resources, managed through careful synchronization to prevent conflicts and ensure a smooth flow of operations within the simulated festival environment.

USE OF PARALLELISM AND THREADS

The festival simulation employs threading to manage simultaneous activities, mimicking the dynamic environment of a real festival.

Implementation of Threading

In the simulation, each festival attendee is represented by an individual thread, allowing for concurrent execution of multiple activities such as ordering food and drinks, go to the stages, and using restroom facilities. This design simulates the behavior of attendees in a real festival, where each individual operates independently yet interacts with shared services. Furthermore, specific service areas such as food trucks and bars do not have their own threads; instead, the focus is on the personnel within these services, such as baristas and cooks, who each operate in dedicated threads to manage tasks like preparing orders. Similarly, doctors in emergency medical services and staff managing bathroom stalls are each allocated separate threads to handle the specific demands of their services efficiently.

For artist performances and security checks at the entrance, the simulation further utilizes separate threads for each stage and continuous loops for security staff, respectively. This ensures that performances are managed without overlaps, allowing artists to perform independently according to their schedules, and that entrance checks are processed

continuously and efficiently. Moreover, the ThreadPoolExecutor is configured to have a number of workers that matches the number of attendees, which optimizes the handling of concurrent actions without delays, enhancing both the responsiveness and the realism of the simulation.

Management of Critical Regions

Critical regions in the simulation refer to sections of the code where shared resources are accessed. These areas include:

- **Order Lists**: Access to order lists in bars and food trucks where baristas and cooks retrieve and update order details.
- **Bathroom Facilities**: Management of access to bathroom stalls, which is particularly sensitive to concurrent access.
- **Emergency Services**: Coordination of patient treatment in the emergency truck where doctors access a shared list of patients.
- **Security checks at the entrance:** Verify attendee tickets and credentials, essential for controlling access and maintaining event security.
- **Stage Access**: Synchronization of performances on stages to prevent more than one artist from using the same stage at the same time.

To effectively manage shared resources within the festival simulation, we implemented locks to safeguard against data races, ensuring that critical operations were executed without conflicts. Additionally, these locks were integral in managing queuing functionalities using lists. For example, in managing the emergency truck and its associated medical staff, we established an empty list for patients and implemented a patient lock. This lock was used in conjunction with a method designed to admit patients, employing classic lock acquire, append to list, and release procedures to ensure orderly and safe admission. This method of managing access and operations through locks and lists was consistently applied across various other systems in the simulation, including bathrooms, stages, etc., thereby maintaining a coherent and controlled environment throughout the festival.

OPTIMIZATIONS & TRADE-OFFS

First of all, as usual in a project of threading and OOP, resource locking was employed as an optimization method to manage access to shared resources, such as the orders list in bars and food trucks. By implementing resource locking, we ensure data integrity and robustness of the system against concurrency issues. This method effectively prevents multiple threads from making conflicting modifications to shared resources, thus avoiding potential inconsistencies and ensuring that each interaction is processed reliably and accurately.

Moreover, throughout the whole process of making the simulation work, we made every change and optimization with one goal: Making the festival as real as possible. In this spirit, in a more specific optimization process, we applied the real-world festival dynamics through the use of time.sleep() and randomization of attendee activities. The time.sleep() function introduces realistic delays mimicking the actual time taken for events like performances, food preparation, and security checks. This not only adds a temporal dimension to the simulation but also helps manage the flow of operations by preventing simultaneous resource access, reducing the risk of system overloads, and ensuring smoother process transitions.

For randomization, each attendee randomly selects activities from options such as dining, drinking, watching performances, or visiting emergency services, influenced by previous activities and individual needs. Furthermore, inside all this randomization of activities, we also included probabilities to enhance realism. This means that if you go to the emergency truck, you will have a lower probability of drinking more. If you drink more, you are more likely to go to the bathroom; etc. This is why, in our final results of the simulation, usually attendees who drank more went more times to the bathroom.

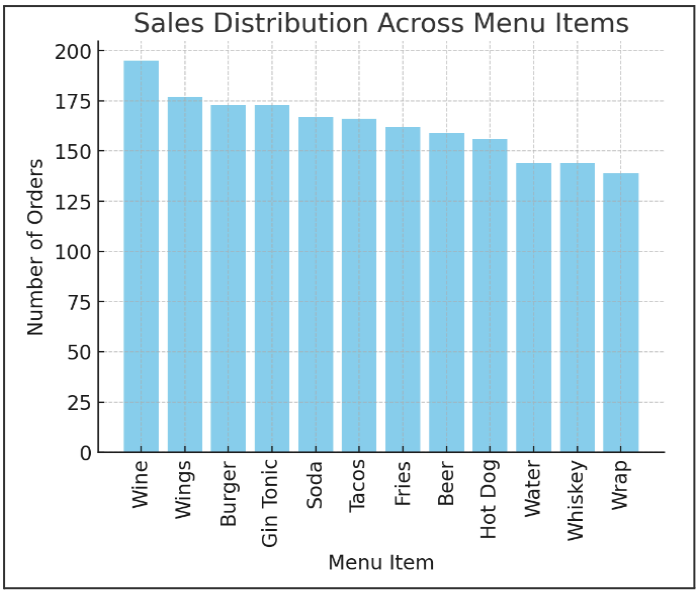
Furthermore, with this randomization and probabilities inputs, the trade-off between increasing realism and controlling the simulation's outcomes is particularly evident when it comes to modeling the decision-making process of attendees on when to leave the festival. The downside of this realistic approach is that it can lead to more attendees leaving the festival earlier than you might wish or expect. This early departure can skew the simulation results, making it challenging to assess other aspects of the festival, such as the efficiency of services. When many attendees leave early, it may also affect the simulated economic impact, as fewer people are consuming food, drinks, or merchandise.

Finally, we faced a trade-off concerning ticket distribution among attendees. To maintain a balanced representation of different ticket types, we chose not to use weighting when assigning tickets, resulting in each category being equally probable. This decision unexpectedly led to a high number of 'no ticket' assignments, substantially reducing the number of attendees who could enter the festival from the planned 500 to about 400 actual participants. This imbalance not only deviated from the typical ticket purchasing behavior observed at real festivals but also affected the simulation's realism. The substantial presence of attendees without tickets skewed operational and infrastructure testing, limiting our ability to assess the festival environment under fully expected conditions.

RESULTS

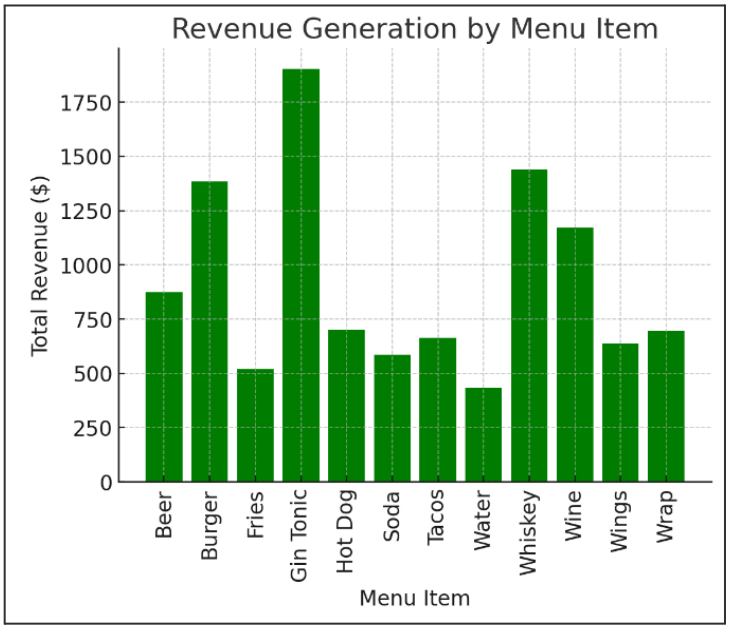
Throughout the simulation, the interactions between attendees and festival services were outputted, revealing patterns that align as closely as possible with anticipated real-world behaviors seen in a festival. The code correctly provided an idea of how individuals would function in the venue. Our initialization of our SQL database helped us organize data extracted from the simulation and analyze the results. We found out the mean age of our attendees was 28.8 years. All ticket types including no tickets had only a variation of 6 units, and that they were exactly the same amount of females and males. We deepen our analysis to better understand the business aspect of the event.

Figure 1



Initially "Sales Distribution Across Menu Items" and "Revenue Generation by Menu Item," we can draw several conclusions about customer preferences and the profitability of different offerings at the festival. The sales distribution graph shows that while many menu items have similar sales numbers, non-alcoholic beverages like Soda and Water have slightly lower sales compared to alcoholic beverages and fast-food items like Wings, Burgers, and Fries. The consistency across most items suggests a diverse range of preferences among attendees, and can help us understand in future events on how to plan for sales.

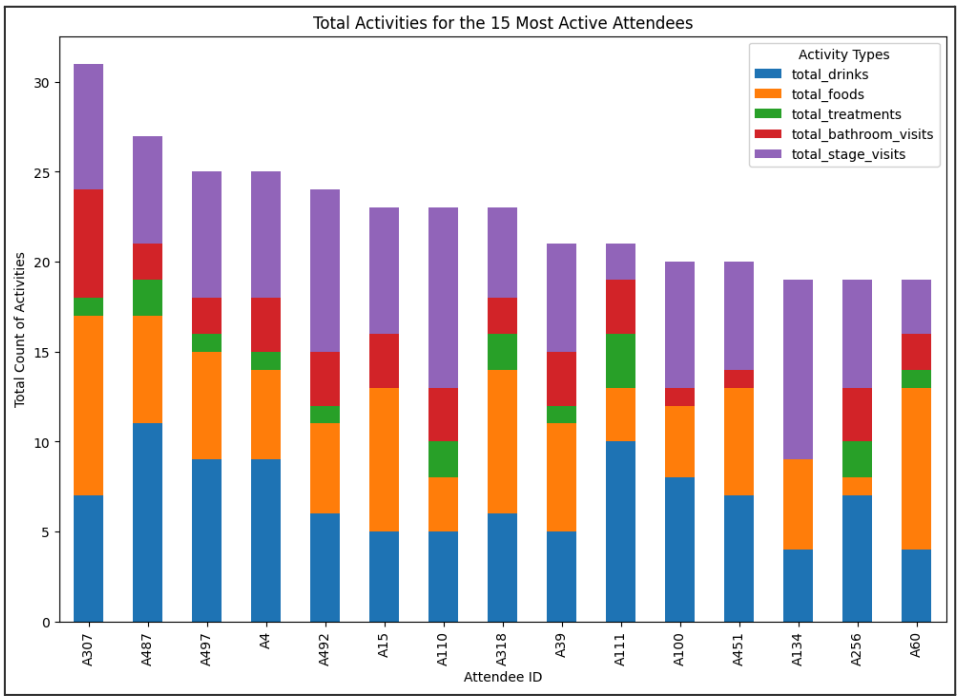
Figure 2



The revenue generation graph reveals more about the financial impact of each item. Although Beer and Wings are among the top revenue generators, higher-priced items like Whiskey and Gin Tonic also contribute significantly to total revenue despite their lower sales volumes. This indicates that the pricing strategy of the festival could have been better developed as there was a lot of lost revenue in items like wings. The whole

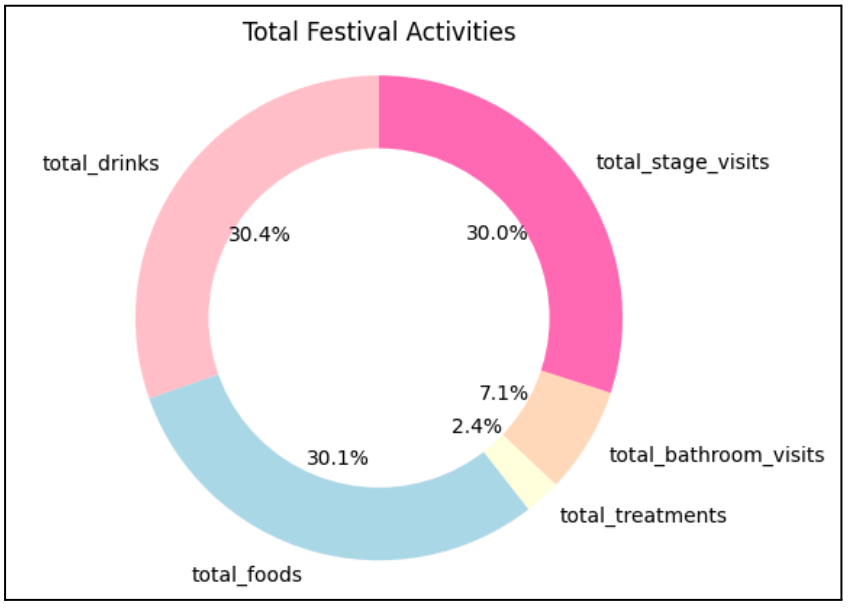
festival only generated 11004.6$ which is something that can be easily increased by changing the price strategy in certain items. Consumptions were very cheap.

Figure 3



We can also analyze the distribution of activities among all visitants and make strategies so that individuals engage more in certain practices. As we can see here, almost all of our top 15 most active attendees had to visit the medical room at least once, which for future events can be an interesting area to try and tackle. Guiding our attendees for a better time and healthier enjoyment.

Figure 4



Regarding the distribution of activities around the festival, initially we can see that all activities possible took place and that those that were significantly less done was because of the weights we put on each activity. Our main three activities were done out of pure randomness and thus being very balanced. This information can lead us to make more tailored activities and adapt weights to better represent realities.

The results of our simulation can lead us to better understand dynamics in such a massive event. The best way to further analyze our outputs would be to simulate with less balanced Attendees, taking into consideration past events to better direct the simulation.

CHALLENGES & LESSONS LEARNED

1. **Sequential vs. Parallel Entrance Processing:**

- Challenge: Initially, the goal was to simulate a continuous flow of attendees entering the festival concurrently with ongoing activities inside. However, managing the synchronization between the entrance checks and ongoing activities proved complex due to the need for precise control over shared resources and timing.

- Lesson Learned: We learned the importance of simplifying complex interactions within a simulation to ensure stability and predictability. By processing all entrance checks sequentially before starting other activities, we sacrificed a bit of realism but gained a more manageable and less error-prone system.

2. **Synchronization and Lock Management:**

- Challenge: Correctly implementing locks to manage access to shared resources such as order queues and bathroom facilities was a significant challenge. Early versions of the simulation encountered issues with deadlocks and race conditions, leading to inconsistent states.

- Lesson Learned: This challenge underscored the critical importance of meticulous lock management in multi-threaded environments. We improved our understanding of how locks can be used effectively to avoid concurrency problems and ensure data integrity.

3. **Multi-Activity Engagement per Attendee:**

- Challenge: Initially, attendees in the simulation were restricted to engage in one activity type once per festival visit. This limitation did not reflect the dynamic nature of real-world festival experiences where attendees might cycle through activities.

- Lesson Learned: We adjusted the simulation to allow attendees to randomly choose from a list of activities and repeat the selection process multiple times. This enhancement made the simulation more realistic and taught us about designing flexible systems that can handle varying user behaviors.

4. **Overall:**

- Challenge: Developing a simulation that accurately mirrors the complexity of a real-life festival with numerous simultaneous activities and interactions among attendees, all while maintaining optimal performance and responsiveness of the system, presented a significant challenge. Future work would include the use of the

ticket type actively in the simulation, more activities based on probabilistic models etc.

- Lesson Learned: This project underscored the critical importance of thorough testing and scalability planning. As the simulation grew more complex, it became evident that early testing phases and scalability assessments were crucial. These efforts helped identify bottlenecks and efficiency issues early, allowing for timely adjustments in the architecture and concurrency management strategies, ultimately leading to a more robust and reliable simulation.