

**Market Visit Problem**

**Project Description:**

This project simulates a real-life scenario where Alice, a software engineer, faces a dilemma while shopping for a unique, hand-painted vase at a local artisan market. Despite admiring the vase, Alice refrains from purchasing it due to its price. Later, she regrets her decision and searches for it online, hoping to find the same item. The problem involves managing inventory availability across physical and online platforms, ensuring synchronization, and handling customer competition. A C program is designed to address these challenges using semaphores, threads, and mutex locks, offering a solution for concurrency issues.

**Project Analysis:**

The simulation requires analyzing the following aspects:

**1.**

**Decision-Making Process:** Alice's hesitation and regret emphasize the importance of quick decision-making in competitive markets.

**2.**

**Inventory Synchronization**: The vase's availability needs to be accurately updated across the market and online platform.

**3.**

**Concurrency Control:** Multiple customers, including Alice, may compete for the vasesimultaneously, requiring a robust concurrency management system.

**4.**

**Resource Management:** Proper allocation and restriction of shared resources (e.g., vase availability) are critical.

The project focuses on implementing a system that resolves these challenges effectively.

**Abstract:**

In a rapidly evolving shopping ecosystem, where physical and online markets intersect, efficient inventory management and customer interaction are vital. This project simulates a scenario using semaphores, threads, and mutex locks to address the synchronization and concurrency issues in shared resource access. By modeling Alice's experience, the program demonstrates how technical solutions can resolve real-world problems involving resource contention and decision-making.

**Introduction:**

Modern markets often face challenges where customer actions, inventory synchronization, and competition overlap. This project explores such a scenario by implementing a simulation in C. Alice's decision to purchase a vase, the inventory management of the artist, and competition from other buyers form the basis of this simulation. Using advanced programming constructs, the project offers a practical approach to managingshared resources and ensuring fairness in customer interactions.

**Threads:**

1. **Customer Threads:**

Simulate individual customers visiting the market or searching for the vase online.

Handle scenarios where customers observe, decide, or regret their decisions.

1. **Restocker Thread:**

Periodically replenishes the inventory in both the market and online store.

Ensures continuous availability for customers while maintaining synchronization.

Threads enable concurrent simulation of actions, allowing multiple processes to occur simultaneously.

**Process:**

The program models the following processes:

1. **Market Visit:**

Alice or other customers visit the physical market to check vase availability.

Mutex locks are used to ensure safe access to shared resources during their visit.

1. **Online Search:**

Represents Alice's effort to find the vase on the artist's website or an e-commerce platform.

Manages inventory updates and prevents data inconsistencies.

1. **Competition Among Customers:**

Simulates other customers competing to buy the vase.

Semaphore control ensures no two customers purchase the same vase simultaneously.

**Semaphore:**

1. **Binary Semaphore:**

Ensures mutual exclusion for critical resources like the vase.

Allows only one customer at a time to access or modify the vase's availability.

1. **Counting Semaphore:**

Manages limited inventory availability in both the market and online store.

Tracks and updates the number of remaining vases across platforms.

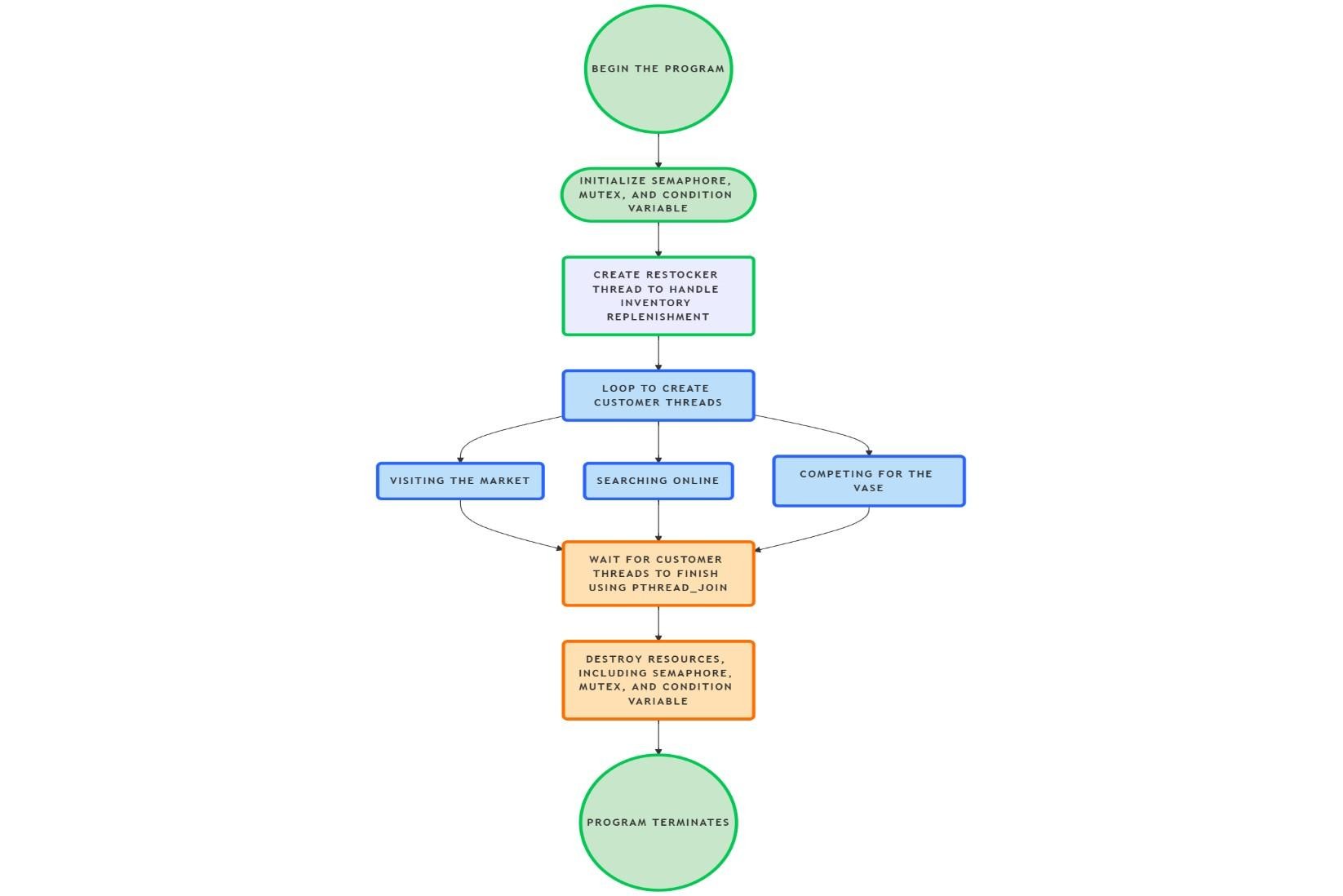
Semaphores and mutex locks together prevent race conditions and ensure smooth execution of the simulation.

1. **Main Function Flowchart**
2. Begin the program
3. Initialize Semaphore, Mutex , and Condition Variable.
4. Create Restocker Thread to handle inventory replenishment .
5. Loop to Create Customer Threads :

1. Create threads for :

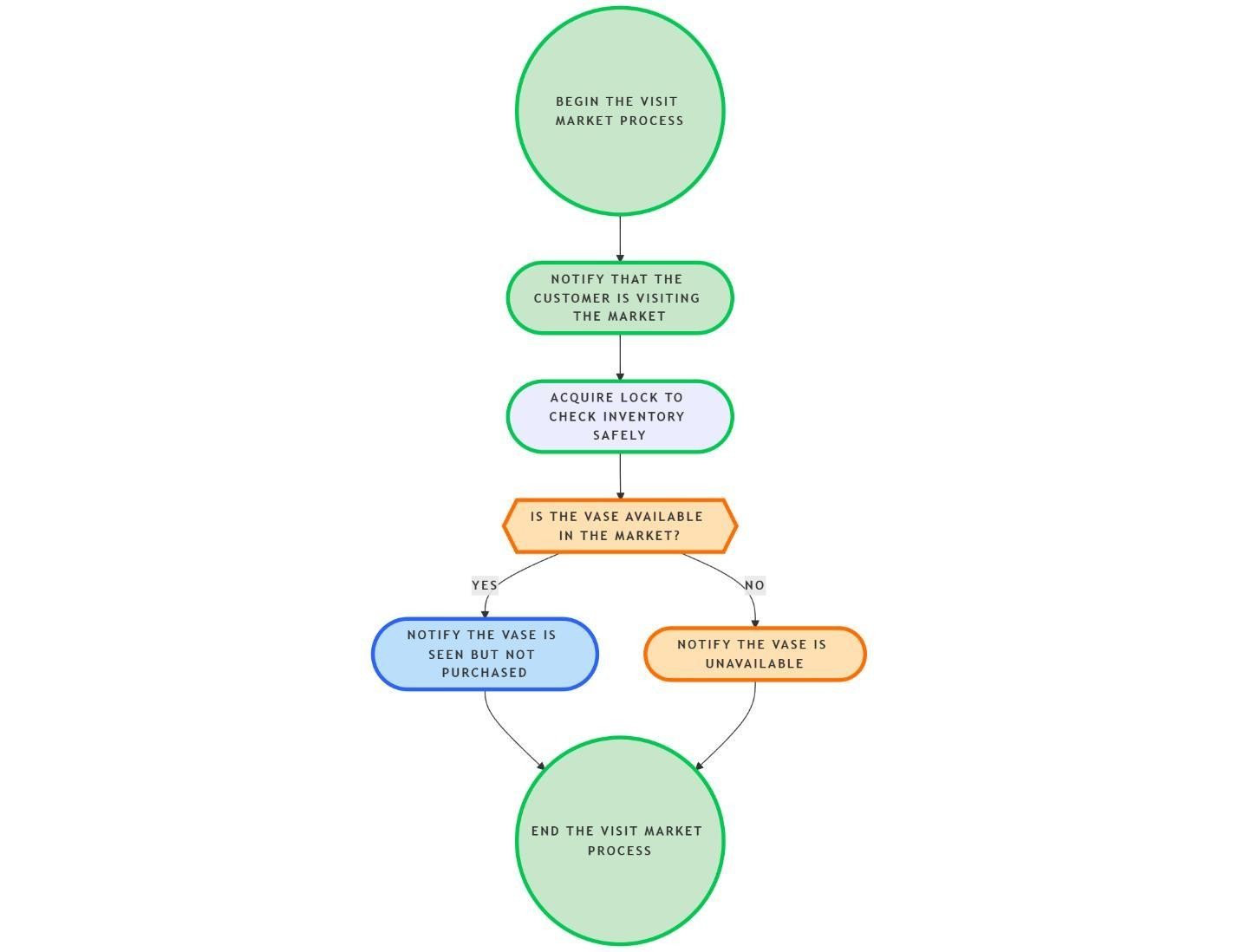
* 1. Visiting the Market .
  2. Searching Online .
  3. Competing for the Vase .

1. Wait for Customer Threads to Finish using pthread\_join to ensure all threads complete.
2. Destroy Resources, including semaphore, mutex, and condition variable.
3. Program terminates(End ) .



**2. Visit Market Flowchart**

* Begin the visit market process .
* Notify that the customer is visiting the market .
* Check Market Inventory:
  + If available, notify that the vase is seen but not bought .
  + If not available, notify that the vase is unavailable .
* Conclude the visit market process .



**3. Search Online Flowchart**

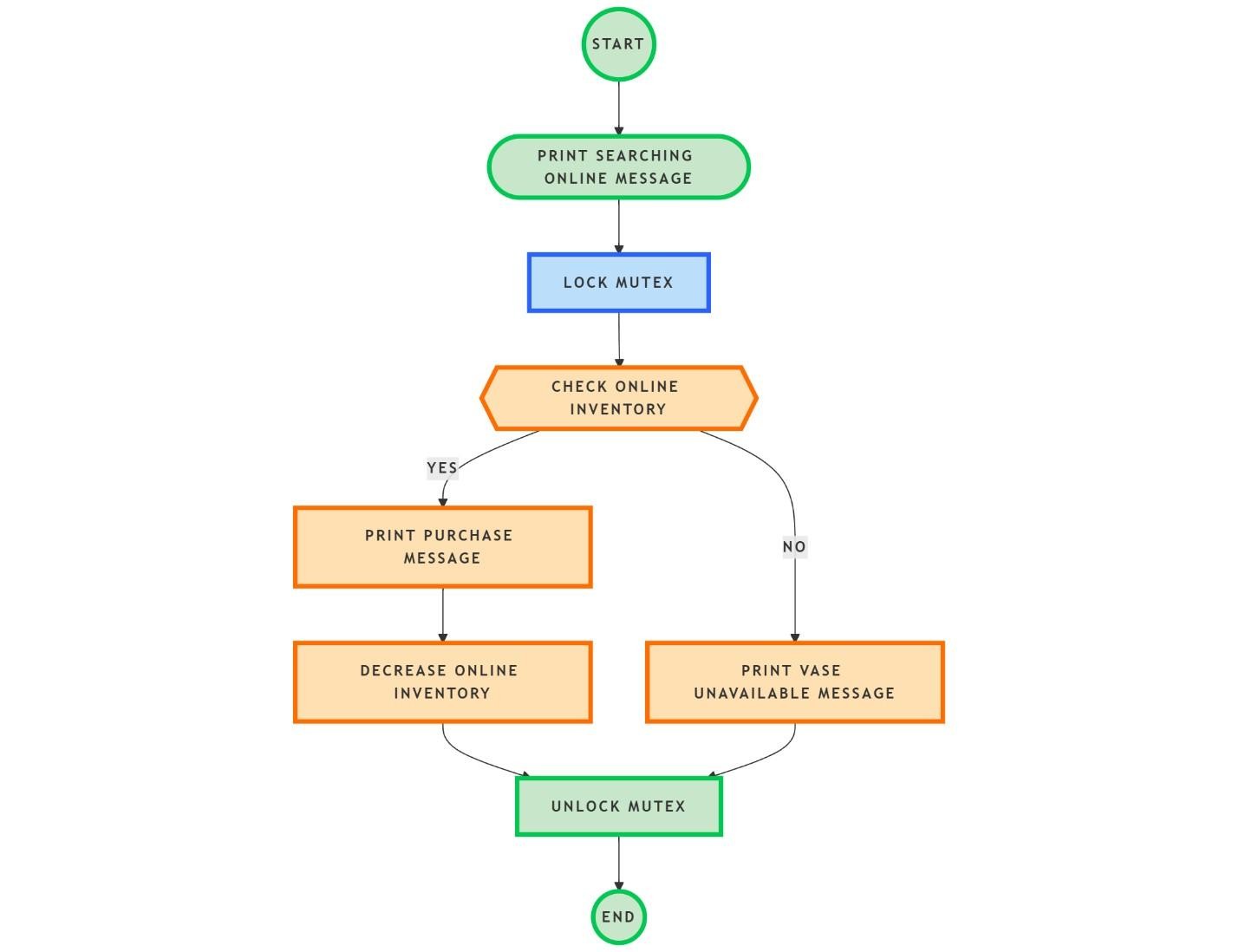
* Begin the search online process .
* Notify that the customer is searching online.
* Check Online Inventory :

o If available :

* + - * Notify that the vase was found and purchased .
      * Update the online inventory by decreasing the count .

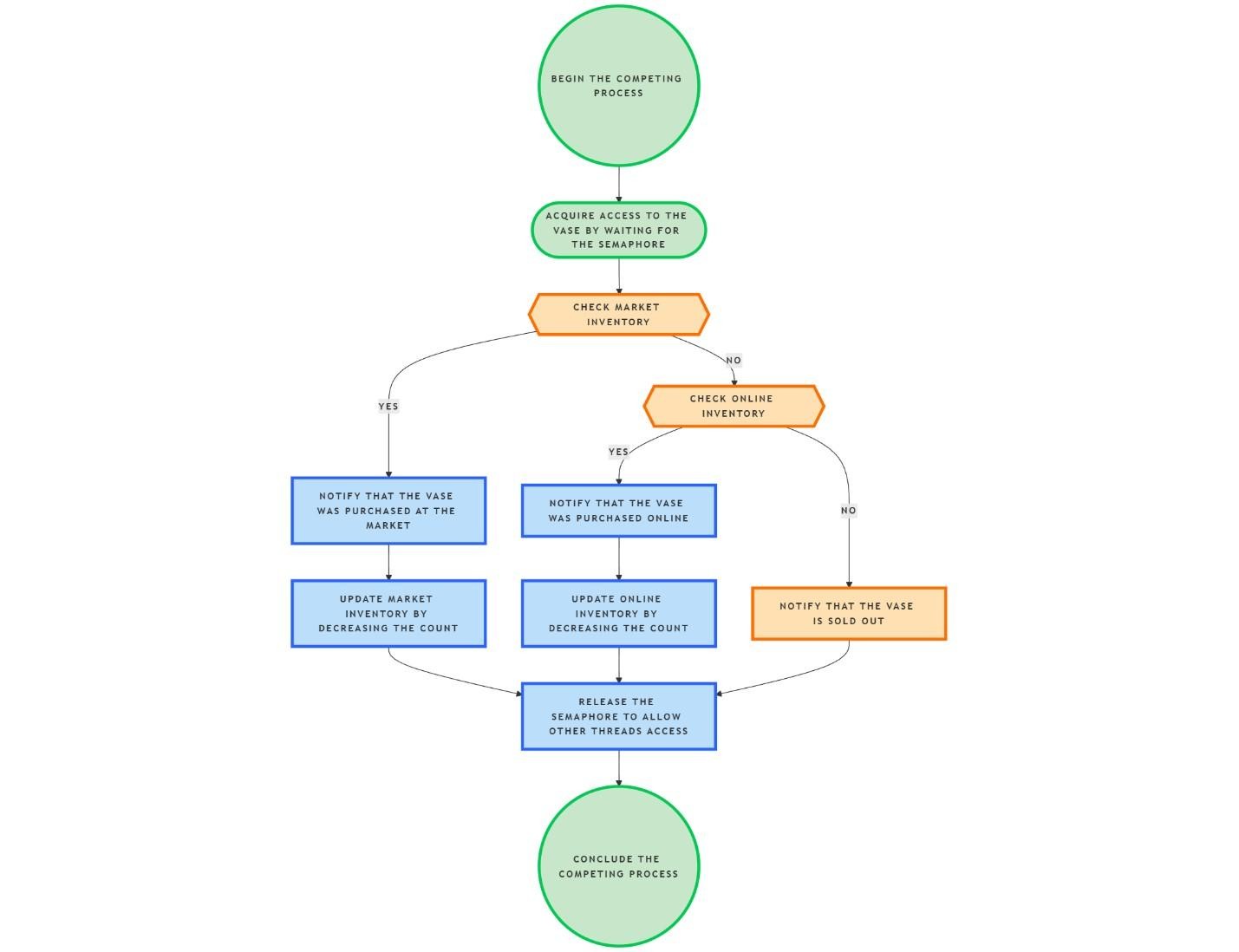
o If not available, notify that the vase is unavailable .

* Conclude the search online process .



**4. Other Customers Competing for the Vase Flowchart**

* Begin the competing process .
* Acquire access to the vase by waiting for the semaphore.
* Check Market Inventory :
  + If available :
    - Notify that the vase was purchased at the market.
    - Update market inventory by decreasing the count.
  + If not available , check Online Inventory:
    - If available:
    - Notify that the vase was purchased online.
    - Update online inventory by decreasing the count.
    - If not available, notify that the vase is sold out.
* Release the semaphore to allow other threads access.
* Conclude the competing process.



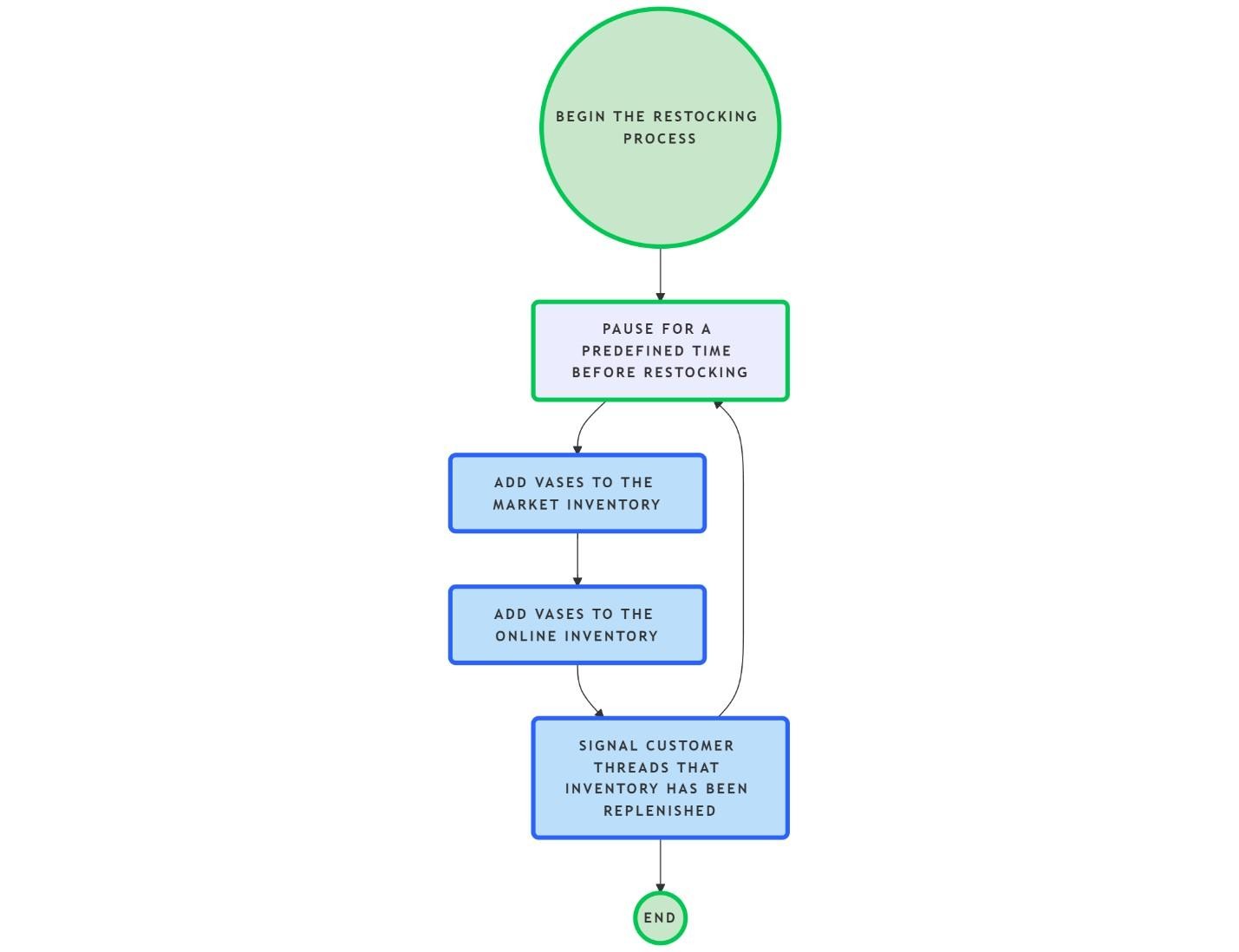
**5. Restocking Inventory Flowchart**

* Begin the restocking process .
* Continuously repeat the following steps: o Pause for a predefined time before restocking

.

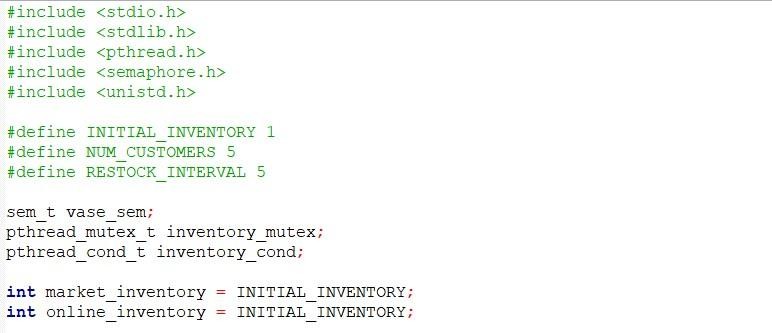
o Add vases to the market inventory . o Add vases to the online inventory . o Signal customer threads that inventory has been replenished.

* Return to the beginning of the loop to continue restocking.



MainCode:

1.



This part of the code is setting up some important variables and synchronization tools that will be used later in the program. Let me explain it in simple terms:

1. **#define Directives:** 
   * **INITIAL\_INVENTORY 1**: This defines the initial number of vases available at both the market and online. Right now, there is only **1 vase** available at each location when the program starts.
   * **NUM\_CUSTOMERS 5**: This sets the number of customers that will be simulated in the program. In this case, **5 customers** will be interacting with the vase inventory.
   * **RESTOCK\_INTERVAL 5**: This defines how often the inventory will be restocked. The restocking happens every **5 seconds**.
2. **Variables for Inventory:** 
   * **market\_inventory**: This variable keeps track of the number of vases available at the market. It starts with the initial value of **1** (as set by

INITIAL\_INVENTORY).

* + **online\_inventory**: This variable keeps track of the number of vases available online. Like market\_inventory, it also starts with **1**.

1. **Synchronization Tools:**

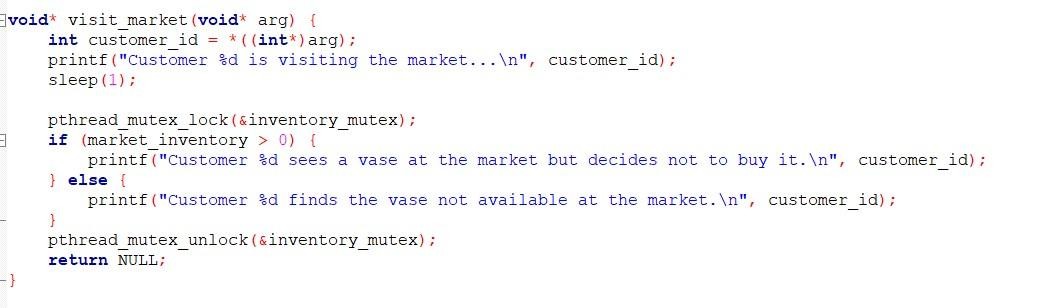
These are used to make sure that different parts of the program (like customers and restockers) don’t mess with the vase inventory at the same time.

* + **sem\_t vase\_sem**: This is a semaphore. A semaphore is like a gatekeeper. It controls access to the vase, allowing only one customer at a time to "interact" with the vase (e.g., buying it). Initially, it is set to **1**, meaning that one customer can access the vase at a time.
  + **pthread\_mutex\_t inventory\_mutex**: This is a mutex (short for "mutual exclusion"). It's like a lock for the inventory. Only one thread (like a customer or restocker) can access the inventory at a time. This prevents multiple threads from changing the inventory at the same time, which could cause problems. The mutex ensures safe access to shared resources like the inventory.
  + **pthread\_cond\_t inventory\_cond**: This is a condition variable. It is used for synchronization and communication between threads. For example, when the inventory is restocked, the restocker can signal other threads (like customers) that they can now proceed to check the inventory.

**In Summary:**

* + The program starts with 1 vase available at both the market and online.
  + **5 customers** will interact with the vase inventory.
  + A **semaphore** ensures that only one customer can access the vase at a time.
  + A **mutex** keeps the inventory changes safe and ensures that no two threads can modify the inventory at the same time.
  + A **condition variable** will be used to notify customers when the inventory has been restocked.

2.



This code defines a function called visit\_market, which simulates a customer visiting the market to check if a vase is available. Here's a breakdown of what it does, in simple terms:

**Steps:**

* 1. **Extract Customer ID:**
  + The function first gets the customer ID from the argument passed to it (the arg variable).
  + The customer\_id is the ID of the customer who is visiting the market.
  1. **Simulate Visiting:**
     + The function prints a message saying, "Customer [ID] is visiting the market...". o Then, it waits for 1 second (sleep(1)) to simulate the time the customer takes to visit the market.
  2. **Access Market Inventory:**
     + The function then "locks" the inventory\_mutex. This is like locking a door to make sure no one else is changing the inventory while the customer is checking it. o It checks if the vase is available in the market (market\_inventory > 0):
       - If the vase is available, it prints a message saying that the customer sees the vase but decides not to buy it.
       - If the vase is not available (inventory is 0), it prints a message saying the vase is not available at the market.

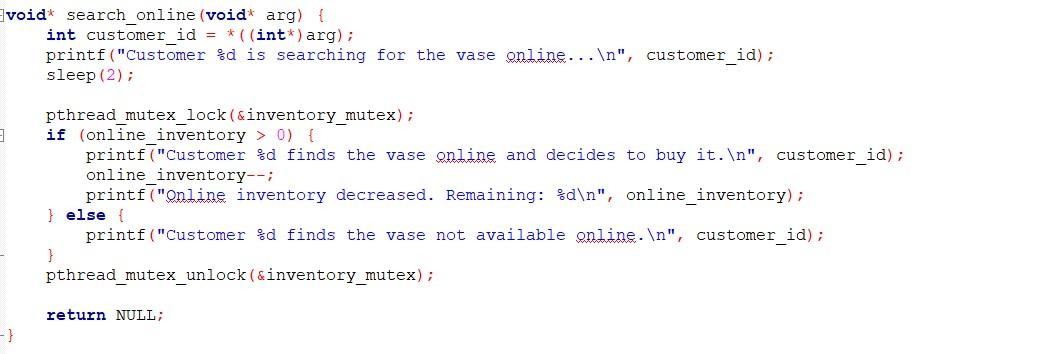
1. **Unlock Mutex:**

o After checking the inventory, the function "unlocks" the mutex, which is like opening the door again for others to access the inventory.

1. **Finish the Function:**

o Finally, the function ends by returning NULL, which means it’s done.

3.



This code defines a function called search\_online, which simulates a customer searching for a vase online and potentially buying it. Here's a breakdown of what the function does, in simple terms:

**Steps:**

* 1. **Extract Customer ID:**

The function gets the customer’s ID from the argument passed to it (the arg variable).

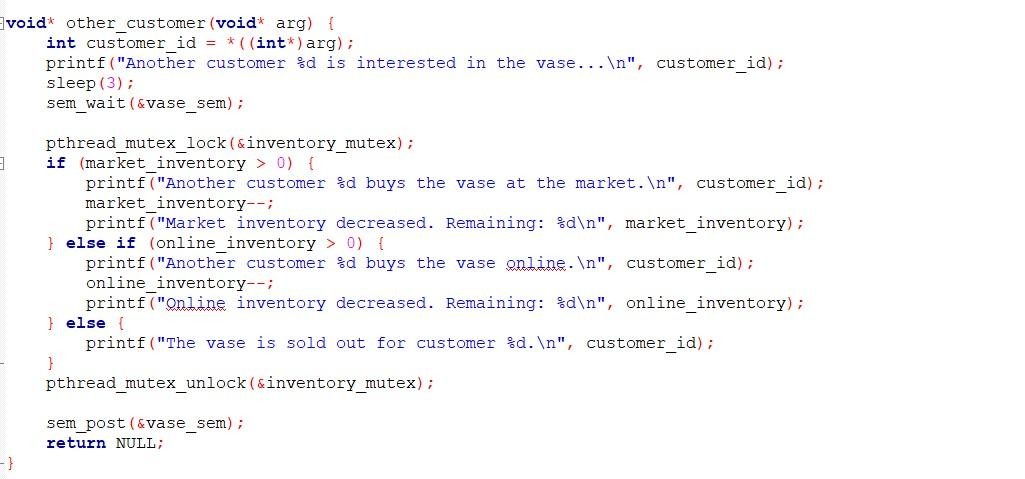
This ID is used to identify which customer is performing the online search.

* 1. **Simulate Search:**
     + The function prints a message saying, "Customer [ID] is searching for the vase online...". o Then, it waits for 2 seconds (sleep(2)) to simulate the time it takes for the customer to search online.
  2. **Access Online Inventory:**
     + The function then "locks" the inventory\_mutex. This lock ensures that no other threads (customers) can modify the inventory while the current customer is checking it. o The function checks if the vase is available online (online\_inventory > 0):
       - If the vase is available online (inventory is more than 0), it prints a message saying that the customer found the vase online and decided to buy it.
       - It then reduces the online\_inventory by 1 (meaning the vase is sold) and prints the updated inventory.
       - If the vase is not available online (inventory is 0), it prints a message saying the vase is not available online.

* 1. **Unlock Mutex:**
     + After checking and possibly updating the inventory, the function "unlocks" the mutex. This allows other customers to access the inventory if needed.
  2. **Finish the Function:**
     + Finally, the function ends by returning NULL, meaning the customer’s search process is finished.

**4.**

o



This code defines a function called other\_customer, which simulates another customer trying to buy a vase from either the market or online. It involves waiting for access to the vase and safely modifying the inventory. Here’s a simple explanation of each part:

**Steps:**

1. **Extract Customer ID:**

* + - The function gets the customer’s ID from the argument passed to it (arg), so it knows which customer is trying to buy the vase.
  1. **Simulate Delay:**

It prints a message saying, "Another customer [ID] is interested in the vase...".

Then, it waits for 3 seconds (sleep(3)) to simulate the time the customer takes to decide whether to buy the vase or not.

* 1. **Acquire Semaphore:**
     + The customer wants to access the vase, but only one customer can buy it at a time. So, the customer uses sem\_wait(&vase\_sem) to "wait" until the semaphore (which controls access to the vase) allows them to proceed. o This ensures that no two customers can try to buy the vase at the same time.

1. **Access and Modify Inventory:**

o The function "locks" the inventory\_mutex to safely check and modify the vase

inventory (this prevents other customers from changing it while the current customer is interacting with it). o It checks if there’s a vase available in the market (market\_inventory > 0):

* + - * + If a vase is available at the market, the customer buys it, and the market inventory decreases by 1. It prints the updated market inventory.
        + If there’s no vase at the market, it checks if there’s one available online (online\_inventory > 0):
        + If the vase is available online, the customer buys it, and the online inventory decreases by 1. It prints the updated online inventory.

o If the vase is sold out in both places (market and online), the function prints that the vase is sold out for this customer.

1. **Release Semaphore:**

o After modifying the inventory, the function "unlocks" the mutex, allowing other customers to check or modify the inventory.

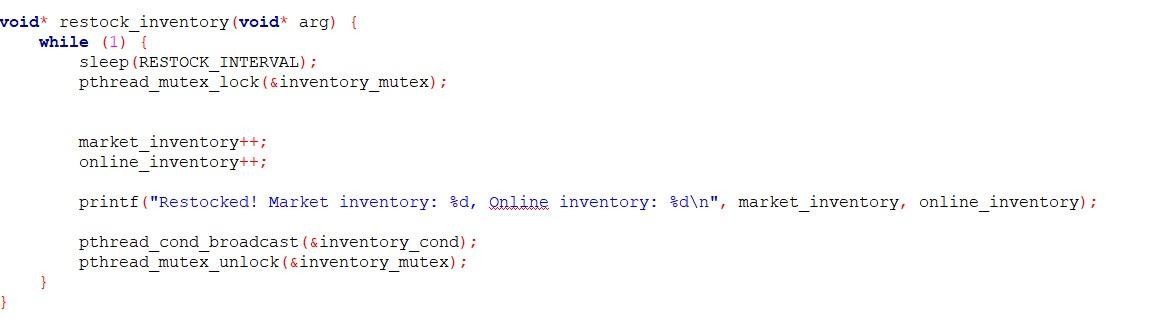
The customer then releases the semaphore with

sem\_post(&vase\_sem), which allows other customers to access the vase if they were waiting for it.

1. **Finish the Function:**

o Finally, the function ends and returns NULL, indicating that this customer’s process is complete.

**5.**



This code defines a function called restock\_inventory, which is responsible for restocking the vase inventory every set interval (defined by

RESTOCK\_INTERVAL).

**Steps:**

* 1. **Infinite Loop (while (1)):**

o The while (1) creates an infinite loop, meaning the function will keep running forever. o This is because the inventory needs to be restocked continuously.

* 1. **Wait for Restock Interval (sleep(RESTOCK\_INTERVAL)):**

The function waits for a certain amount of time

(RESTOCK\_INTERVAL), which is typically 5 seconds.

This simulates the time between each restocking of the vase.

* 1. **Lock Mutex (pthread\_mutex\_lock(&inventory\_mutex)):**

o Before modifying the inventory (market and online), the function locks a mutex (inventory\_mutex). o This "lock" ensures that no other thread (like a customer trying to buy a vase) is changing the inventory at the same time. o The mutex is like a lock on a door that keeps other people from entering while this function is working on the inventory.

* 1. **Restock the Inventory:**

o Once the mutex is locked, the function adds one vase to both the **market inventory** and **online inventory**. o This is the actual restocking process: it increases the count of vases available in both places by 1.

* 1. **Print the Updated Inventory:**

o After restocking, the function prints a message showing the new inventory levels for both the market and online. o For example: "Restocked! Market inventory: 2, Online inventory: 2".

* 1. **Notify Waiting Threads (pthread\_cond\_broadcast(&inventory\_cond)):**

o The function then sends a signal to all threads (like customers) that are waiting

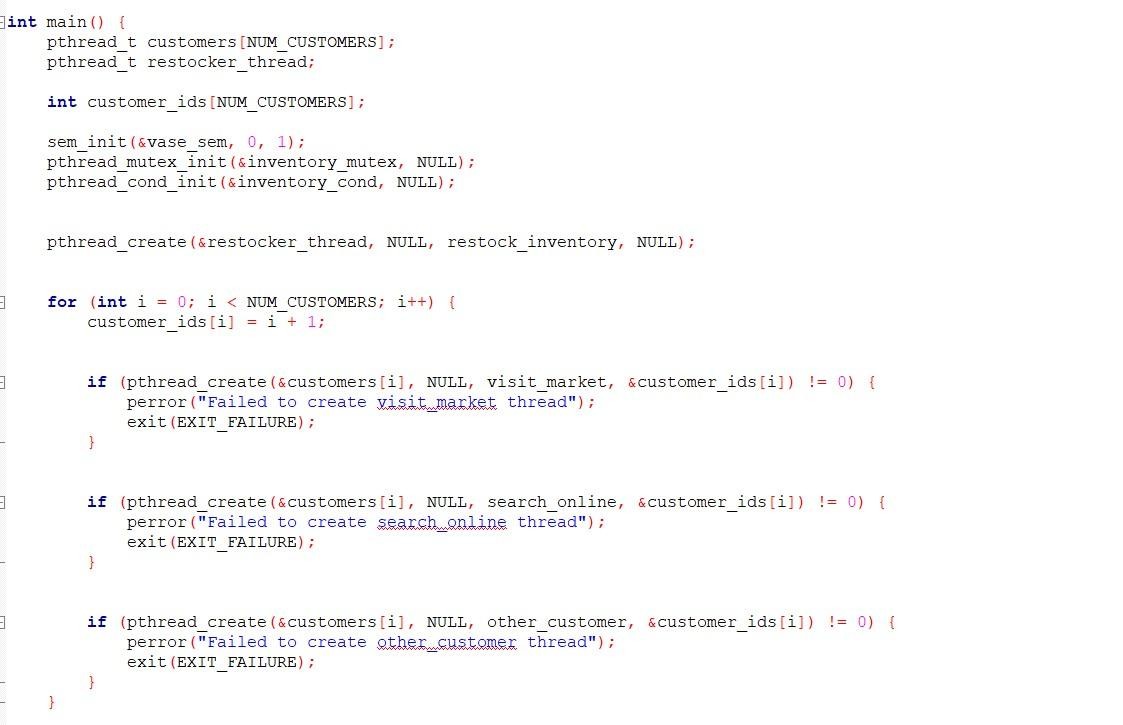
for the inventory to be restocked. o This is done using pthread\_cond\_broadcast, which notifies all waiting threads that they can now check or buy the vase if needed.

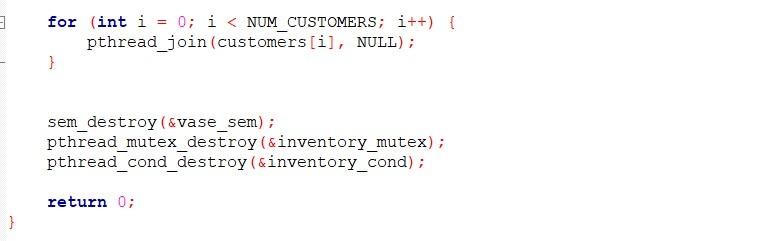
1. **Unlock Mutex (pthread\_mutex\_unlock(&inventory\_mutex)):**

o After the inventory is updated and other threads are notified, the function "unlocks" the mutex. o Unlocking the mutex allows other threads (like customers) to access and modify the inventory.

o

6.





This part of the code is the **main function**, which sets up and manages all the threads and resources for the vase-selling system.

1. **Setting Up Variables and Resources:** 
   * **Customer Threads Array (customers[NUM\_CUSTOMERS])**: An array to hold the threads for each customer. Each customer will have a separate thread.
   * **Restocker Thread (restocker\_thread)**: A thread that will be used to keep restocking the inventory (the vase).
   * **Customer IDs Array (customer\_ids[NUM\_CUSTOMERS])**: An array that holds unique IDs for each customer.

Next, the program prepares the resources that will be used by different threads:

* + **Semaphore (vase\_sem)**: Initializes a semaphore to control access to the vase. The value is set to 1, meaning only one customer can interact with the vase at a time.
  + **Mutex (inventory\_mutex)**: Initializes a mutex (mutual exclusion) lock to ensure that only one thread can access and modify the inventory at a time.
  + **Condition Variable (inventory\_cond)**: Initializes a condition variable, which will be used to notify customer threads when the inventory is restocked.

1. **Creating Threads:**

The program creates the threads that will perform different actions:

* + **Restocker Thread**: This thread runs the restock\_inventory function to continuously restock the vase inventory every few seconds.

Then, the program creates threads for customers. For each customer:

* + It **assigns a unique customer ID** (e.g., customer 1, customer 2, etc.).
  + It creates **three threads for each customer** to simulate three different actions:
    1. **Visiting the market** (visit\_market function).
    2. **Searching online** for the vase (search\_online function).
    3. **Competing to buy the vase** (other\_customer function).

If any thread fails to be created (e.g., if there is a problem with the system), the program will print an error message and exit.

1. **Waiting for Threads to Finish:**

After creating all the threads, the program waits for all the customer threads to finish using pthread\_join. This ensures that the main program does not exit until all customer actions are complete.

1. **Cleaning Up:**

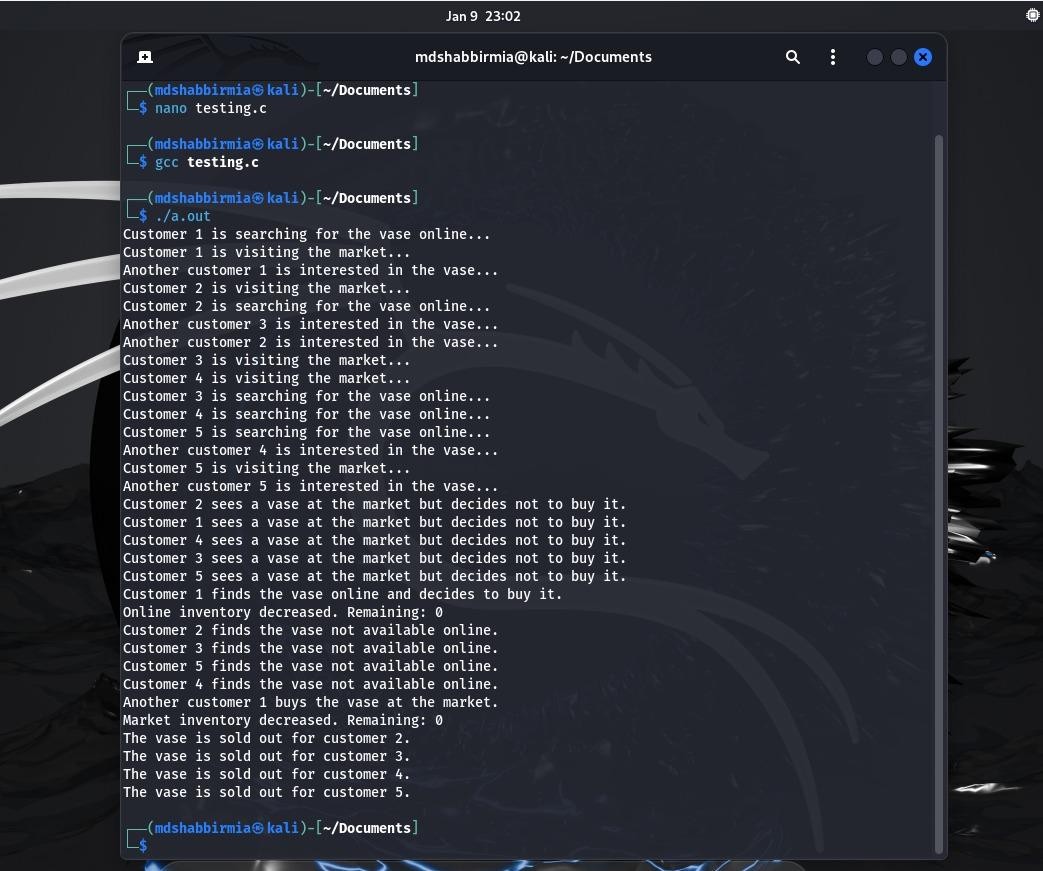
After all the threads have finished, the program **cleans up the resources** it used:

* + **Destroys the semaphore**: This frees up the semaphore resource.
  + **Destroys the mutex**: This frees up the mutex resource.
  + **Destroys the condition variable**: This frees up the condition variable resource.

1. **Program End:**

Finally, the program finishes and returns 0 to indicate that everything worked properly.

**Output:**



**Conclusion:**

Our project is about the "Market visit problem". Three members of our group solved it using C language and ran our project in the Ubuntu terminal—our honorable course instructor Md. Nawab Yousuf Ali has taught us how to implement every topic theoretically and practically. So we used the knowledge and implemented a synchronization method to solve the project. During the project solving, we found some issues multiple times and fixed them together. After this, we got a perfectly done project. Overall the project is very efficient and useful for users.