BLG 312E

COMPUTER OPERATING SYSTEMS

Homework-2

Course Instructor:

Prof. Dr. Kemal BIÇAKCI

Prepared by: Uğur AYVAZ

Contact me at *ayvaz18@itu.edu.tr*

April 2025

1. HOMEWORK-2: ONLINE MARKET CONCURRENCY CONTROL

1.1. ACADEMIC INTEGRITY AND PLAGIARISM WARNING

All work submitted must be your own. We apply **plagiarism checks** to both code and report files. Copying from other students or directly from online sources (code, text, etc.) is strictly prohibited. Violations will result in severe academic penalties, including a zero for the assignment and potential disciplinary action.

1.2. PROJECT OVERVIEW

This project simulates an online shopping platform in which multiple customers concurrently interact with a shared stock system. The goal is to investigate and manage concurrency-related challenges such as race conditions, timeout handling, and resource access control through the use of thread and the determined synchronization mechanism.

1.3. CONCURRENCY OBJECTIVES

- Prevent race conditions on stock access
- Isolate and protect critical sections
- Apply timeout-based soft reservation to simulate real-time purchasing
- Ensure fair access to shared resources (e.g., synchronized stock updates to prevent over-selling, enforcing the maximum concurrent payment limit through

cashier slot control, and protecting log file writes from concurrent access to maintain consistency and readability)

- If these shared resources are not properly synchronized, critical issues may arise, such as:
  - **Overselling** products due to race conditions in stock access.
  - More than two customers attempting payment simultaneously, **violating system constraints** (See 1.4.2).
  - **Corrupted or interleaved log entries** due to concurrent writes in the log file.
- Design real-world limitations such as restricting access to payment processing based on system capacity — for example, only two customers may proceed with payment concurrently, simulating limited physical cashiers in a store.

1.4. SYSTEM BEHAVIOR AND DESIGN

Important: Each thread represents a separate product request. A customer's cart is not unified (only one type of product in each cart). Each product is reserved, timed out, or purchased independently. This reflects a realistic, item-level concurrency scenario.

1.4.1. System Parameters

- Maximum number of concurrent customers: 5.
- Total number of distinct products: 3.
- Maximum stock per product: 5 units.
- Initial stock per product: provided inside the `input.txt`.
- Quantity per product request: provided inside the `input.txt`.
- Reservation timeout: 10,000 milliseconds (10 seconds)
- At most 2 customers can finalize their purchase at the same time (imagine there are only two cashiers as in a real market checkout scenario).

1.4.2. Rules and Restrictions

These parameters can be configured in the code but serve as a baseline for testing and simulation.

- The system limits the number of customers who can finalize a purchase simultaneously (e.g., proceed to payment) to simulate real-world backend constraints, such as limited payment processing capacity. For instance, only two customers may enter the payment section at the same time. For this simulation, *a maximum of 2 customers* can enter the payment section at the same time. This models limited backend resources or payment processing capacity.
- Each product has limited stock, accessed concurrently by multiple customers.
- Each customer may place multiple, independent product orders. For each order, *a separate thread is launched*.
- The thread attempts to reserve the requested quantity of a product. If available, it's moved into the customer's cart.
- From that moment (available and added to the cart), *a 10,000-millisecond timeout window* starts.
- If the customer does not finalize the purchase in time, the item is automatically returned to the stock.
- If another customer attempts to purchase a product while it is reserved, the system must check if any remaining stock exists. If no sufficient stock is left, the customer is notified of how much time in milliseconds remains until the reservation may expire.
- If the product is re-attempted after timeout expiration and not already purchased by another customer, it can be acquired.
- Each product request thread works independently. There is *no batching* per customer. This means that each product is ordered separately, and only one type of product is placed in the cart (no unified cart).

1.4.3. A Sample Data Structure for a Product Request

Note: This is just an example structure. You are free to modify or design your own as long as the core requirements (customer ID, product ID, quantity) are captured and handled correctly.

```
typedef struct {
    int customer_id;
    int product_id;
    int quantity;
} ProductRequest;
```

This structure represents a single product request initiated by a customer. Each request is managed independently and tracked with its own timeout window.

1.4.4. Example Thread Usage

Note: This is a simplified illustration. You are encouraged to design your own threading logic as long as each product request is managed independently.

```
// A separate dedicated thread is created for each product request.
// For each product the customer wants to buy:
//   - Create a new thread to process the request
//   - The thread handles product ID and quantity
//   - It attempts to reserve the product and waits for timeout or purchase
// Pseudo Code Example:
//   thread1 → requests (product_id = 1, quantity = 2)
//   thread2 → requests (product_id = 0, quantity = 1)
```

Each thread represents a single product order attempt by a customer and should be evaluated in real time against the state of the system.

1.4.5. Timeout Management

- Each thread is responsible for tracking its own timeout window after the product is added to the cart. No global timer or shared timeout manager is required.

- The timeout is handled ***per product***, not per customer or per cart.
- When a thread places a product in the cart, the current timestamp is recorded. If the customer doesn't complete the purchase before the timeout, the item is automatically removed from the cart and **returned to the stock**.
- In order to simulate real-world behavior, not every customer is guaranteed to complete their purchase. Each product thread can randomly decide whether the customer will proceed to payment or allow the timeout to expire.
- Each customer has a 50% chance of completing the purchase.

```
int will_purchase = rand() % 2;  //For each reserved product, we randomly
    determine whether it will result in a purchase or timieout expire and
    return to stock.
```

- If `will_purchase` is 0 (false), the thread simply waits until the timeout expires without initiating payment.
- Upon payment attempt or background timeout check, the product's timeout is validated.
- **Important:** Even if a customer decides to purchase (`will_purchase` is 1), the system enforces a limit of 2 concurrent payments. If the customer cannot acquire a payment slot in time, the thread may remain blocked until the timeout expires — resulting in a failed purchase due to unavailability of checkout resources.
- If a customer initiates a purchase but all cashier slots are occupied (maximum concurrent payments (#2) reached), the thread may block or retry automatically. As long as the timeout window has not expired, the purchase may still succeed once a slot becomes available. This reflects realistic queueing behavior under resource constraints.
- the timeout expires, the item is removed from the cart and **stock is replenished**.

1.4.6. Important Behavior

- In concurrent systems, the first thread to access shared stock reserves the item. Others must wait, retry, or receive a reservation notice.

- A product in the cart is temporarily deducted from stock, even if not yet purchased.
- If a single *product reservation retry* is scheduled and another thread reserves the product first, the retry must be invalidated.
- **Note:** Retry logic during the payment phase is different from *product reservation retry*. In the reservation stage, a customer only retries **once** if the product is temporarily unavailable. However, during payment, a customer thread may automatically retry or block until a cashier slot becomes available — as long **as the reservation timeout has not expired**.

1.4.7. Reservation Awareness and Retry

- If a customer attempts to purchase a product that is out of stock due to another reservation:
  - The system checks the remaining timeout "Y" of the reserved product "X".
  - A message like the following is shown: "Product X is currently reserved. Try again in Y milliseconds."
  - The system may also queue a retry. If the product becomes available again (not purchased), the retry can succeed.
  - This retry request is registered with a timer.
  - If another customer successfully reserves and purchases the product before the retry (while waiting the timeout), the retry must be cancelled and the event logged as a failed retry attempt.
  - Each customer can only be queued for *`a single retry per product request`*. If the retry fails, no further waiting is scheduled for the same customer.
- This ensures fairness and prevents late retries from overriding successful purchases.

1.4.8. Input File Specification

The simulation will use a fixed input file named `input.txt`, which is provided within Homework-2. Your C program must read this file at runtime to configure the system (i.e.,

number of customers, products, timeout, and product requests). **You must not modify or generate this file.** Instead, use its contents to initialize data structures and create request threads.

1.4.8.1. Expected Format

- `num_customers`: Number of customers in the simulation
- `num_products`: Number of distinct products
- `reservation_timeout_ms`: Timeout duration in milliseconds
- `max_concurrent_payments`: Max number of concurrent payments (to be enforced via condition variables)
- `initial_stock`: Comma-separated list of initial stock values for each product
- Customer requests: one per line, in the format `customer_id, product_id, quantity`

1.4.8.2. Sample `input.txt`

The input file has two parts:

1. **System Configuration** The first five lines define the simulation setup:
   - `initial_stock=5,5,5` means:
     - Product 0 starts with 5 units,
     - Product 1 starts with 5 units,
     - Product 2 starts with 5 units.
2. **Customer Requests** Each line after that follows the format: `customer_id, product_id, quantity`

   For example, `1,1,1` means that Customer_1 requests 1 unit of Product_1.

   ### Concurrent Request Groups (Important)

   `input.txt` includes customer requests in groups. Requests in the same group are intended to be triggered **concurrently**. Groups are separated by a single blank line in the `input.txt`.

   Example from input.txt:

```
1,1,1   // these three requests
1,0,2   // were triggered
2,0,1   // simultaneously.
        //You must treat blank lines as group separators and ignore them
            when parsing.
3,0,3   // these two requests were
4,0,1   // triggered simultaneously.
```

1.5. STUDENT RESPONSIBILITIES

- Identify and resolve concurrency conflicts.
- Implement chosen synchronization mechanisms.
- Implement time-based logic and per-product cart handling.

- Ensure safe access to logs and shared memory.

1.6. TECHNICAL CONSTRAINTS

- Programming language: C
- Each code must be tested on the Docker container.
- Allowed libraries: ‘pthread.h‘, ‘time.h‘, ‘stdio.h‘
- External concurrency frameworks (OpenMP, Boost) are **not allowed**.
- Shared memory or IPC is not required.

  Important: You must implement this assignment *without using semaphores*. Instead, use `mutex` and `condition variables` to implement synchronization logic. This includes managing critical sections, customer retry behavior, and limiting concurrent payments.

1.7. QUESTIONS TO BE ANSWERED IN THE REPORTS

1. **How did you ensure that no more than two customers could make a payment at the same time?** Explain how you implemented this constraint using `mutex` and `condition variables`. Additionally, discuss an alternative synchronization mechanism that could have been used for this purpose, and compare its behavior with your current implementation.
2. **What strategy did you use to manage reservation timeouts, and how did your implementation ensure thread safety during stock access?** Describe how you handled conflicts in stock availability and ensured consistent behavior across threads.
3. **Under what conditions does a product request thread terminate?** Explain how you designed thread exit conditions and how you ensured proper resource cleanup and logging.

1.8. EVALUATION CRITERIA

Your project will be evaluated based on:

Evaluation Item	Points
Identification of concurrency issues and risks	10
Correct application of synchronization techniques	20
Logical and modular thread implementation	15
Handling of reservation, purchase, and timeout (including reservation retry and purchase retries in before timeout expires)	10
Functional logging and output clarity	10
Code compiles and runs cleanly (e.g., tested in Docker Container)	10
Report quality and explanation depth (including answers of the questions)	25
Total	**100**

1.9. SAMPLE LOG OUTPUT

Timestamps represent milliseconds since UNIX epoch. Students may convert them to a readable time format with millisecond precision for clarity and debugging.

Log output should show which thread (customer) performed what action, including:

- Initial stock
- Requested product and quantity
- Cart placement and stock change
- Timeout and return to stock
- Retry or conflict situations
- Successful purchases
- etc.

Sample Log Output:

```
[1713209805] Initial Stock: [product 0: 5, product 1: 5, product 2: 5]
[1713209809] Customer 1 tried to add product 1 (qty: 1) to cart | Stock: [product 0: 5,
    product 1: 4, product 2: 5]   // succeed
[1713209810] Customer 1 tried to add product 0 (qty: 2) to cart | Stock: [product 0: 3,
    product 1: 4, product 2: 5]   // succeed
[1713209811] Customer 2 tried to add product 0 (qty: 1) to cart | Stock: [product 0: 2,
    product 1: 4, product 2: 5]   // succeed
[1713209813] Customer 1 attempted to purchase product 0 (qty: 2) | Stock: [product 0: 2,
    product 1: 4, product 2: 5]
[1713209813] Customer 1 attempted to purchase product 1 (qty: 1) | Stock: [product 0: 2,
    product 1: 4, product 2: 5]
[1713209813] Customer 2 attempted to purchase product 0 (qty: 1) | Stock: [product 0: 2,
    product 1: 4, product 2: 5]
[1713209814] Customer 1 purchased product 0 (qty: 2) | Stock: [product 0: 2, product 1:
    4, product 2: 5]
[1713209814] Customer 1 purchased product 1 (qty: 1) | Stock: [product 0: 2, product 1:
    4, product 2: 5]
[1713209814] Customer 2 couldn't purchase product 0 (qty: 1) and had to wait! (maximum
    number of concurrent payments reached!)
[1713209815] Customer 3 tried to add product 0 (qty: 3) but only 2 units were available
    | Stock: [product 0: 2, product 1: 4, product 2: 5]    // failed
[1713209815] Customer 4 tried to add product 0 (qty: 1) to cart | Stock: [product 0: 1,
    product 1: 4, product 2: 5]    // succeed
[1713209815] Customer 2 (automatically) retried purchasing product 0 (qty: 1) | Stock:
    [product 0: 1, product 1: 4, product 2: 5]   //(checked for available cashier slot
    before timeout expired.)
[1713209815] Customer 4 attempted to purchase product 0 (qty: 1) | Stock: [product 0: 1,
    product 1: 4, product 2: 5]
[1713209815] Customer 2 purchased product 0 (qty: 1) | Stock: [product 0: 1, product 1:
    4, product 2: 5]
[1713209815] Customer 5 tried to add product 2 (qty: 2) to cart | Stock: [product 0: 1,
    product 1: 4, product 2: 3]    // succeed
[1713209816] Customer 4 purchased product 0 (qty: 1) | Stock: [product 0: 1, product 1:
    4, product 2: 3]
[1713209827] Customer 5 attempted to purchase product 2 (qty: 2) | Stock: [product 0: 1,
    product 1: 4, product 2: 3]
[1713209827] Customer 5 could not purchase product 2 (qty: 2) in time.  Timeout is
    expired!!! Product 2 (qty: 2) returned to the stock! | Stock: [product 0: 1, product
    1: 4, product 2: 5]
[1713209827] Customer 3 retry attempt failed - product already reserved (or purchased)
    by another customer | Stock: [product 0: 1, product 1: 4, product 2: 5] // no more
    retry for this thread
```

NOT THAT: To check available cashier slots the customer retries automatically many times until the timeout expires. However, the customer only tries once for adding a product to the cart. Here the thread checks the timeout of the reservation.

1.10. SUBMISSION CONTENTS

All submission materials must be compressed into a single `.zip` archive. The file should be named as: `studentID_HW2.zip`. This archive must include all of the following files:

File	Description
`studentID_market_sim.c`	Complete application in *C*.
`*.h`	Header files (if any used in your project).
`Makefile`	Compilation automation using *make*.
`README.txt`	Compilation and usage instructions.
`log.txt`	Logged actions from concurrent system events
`input.txt`	Instructor-provided configuration file, including grouped concurrent requests. Your code must read and parse it. Do not modify `input.txt`.
`studentID_Fullname_report.pdf`	A 2–3 page report that documents the student's implementation details, design decisions, and reflections on concurrency handling in the project.