GEBZE TECHNICAL UNIVERSITY

CSE 344
SYSTEM PROGRAMMING

Final Report

PideShop Server-Client Application

BARIŞ YURDAKUL
1801042620

June 15, 2024

Contents

1 Problem Definition

In the context of a modern food delivery system, particularly a PideShop, managing order processing, cooking, and delivery poses several challenges. The primary objective of this project is to design and implement a server-client application capable of efficiently handling these tasks.

1.1 Challenges

- **Order Management**: Handle orders arriving asynchronously from multiple clients, ensuring each order is queued and processed in sequence.
- **Cooking Management**: Assign orders to cooks, manage cooking time and oven capacity, and ensure optimal use of resources.
- **Delivery Management**: Manage a queue of cooked orders, assign couriers, and ensure timely and efficient delivery.
- **Concurrency**: Handle multiple clients and workers concurrently, using robust synchronization mechanisms to avoid race conditions.
- **Logging and Reporting**: Maintain detailed logs and generate summary reports upon shutdown, recognizing top-performing workers.

1.2 Requirements

- **Scalability**: Handle varying numbers of clients, orders, cooks, and couriers without performance degradation.
- **Fault Tolerance**: Handle unexpected conditions without crashing.
- **Efficiency**: Optimize resource use to minimize wait times and ensure timely deliveries.
- **User Feedback**: Provide feedback to clients about their order status and maintain logs for administrative purposes.

By addressing these challenges, this project aims to develop a comprehensive server-client application for a PideShop, ensuring efficient order processing, cooking, and delivery, while maintaining logs and generating reports.

2 Implementation

2.1 Order Queue Management

The server maintains two queues: one for orders waiting to be cooked and another for orders ready for delivery.

```
typedef struct Order {
    int client_socket;
    int order_id;
    int x, y;
    pid_t client_pid;
    struct Order* next;
} Order;

typedef struct {
    Order* front;
    Order* rear;
    int size;
    pthread_mutex_t mutex;
} OrderQueue;

OrderQueue order_queue;
OrderQueue delivery_queue;
```

- **Order**: Represents a client's order, including its ID, coordinates, client's PID, and socket information.
- **OrderQueue**: A linked list structure for managing orders, protected by a mutex for thread safety.

Functions:

- `void enqueue(OrderQueue* queue, Order* order)`: Adds an order to the queue.
- `Order* dequeue(OrderQueue* queue)`: Removes an order from the queue.

2.2 Cooking and Delivery Thread Management

Cooks and couriers are represented by worker threads, each processing orders in their respective pools.

```
typedef struct {
    int id;
    bool available;
    pthread_cond_t cond;
    int orders_processed;
} Worker;

Worker* cooks;
Worker* couriers;
int cook_thread_pool_size;
int delivery_thread_pool_size;
```

- **Worker**: Represents a cook or a courier, including ID, availability status, condition variable for synchronization, and the number of processed orders.

Thread Functions:

- `void* cook thread(void* arg)`: Manages cooking process.
- `void* delivery thread(void* arg)`: Manages delivery process.

Cook Thread Logic:

1. Wait for an order in the queue.
2. Process the order (simulate cooking).
3. Move the order to the delivery queue.

Delivery Thread Logic:

1. Wait for cooked orders in the delivery queue.
2. Deliver the orders to the client's location.
3. Update the delivery count and free resources.

2.3 Client Handling

Clients connect to the server, send their orders, and wait for the delivery. The server handles incoming connections and assigns orders to the appropriate queues.

```
typedef struct {
    pid_t pid;
    int numberOfClients;
    bool announced;
    int orders_to_serve;
} ClientInfo;

ClientInfo clients[MAX_CLIENTS];
int client_count = 0;
```

Client Connection Handling:

1. Accept incoming client connections.
2. Receive order details from clients.
3. Enqueue orders into the order queue.
4. Maintain client information for logging and tracking.

2.4 Logging and Reporting

The server maintains a log file to record activities and provides a summary report upon shutdown.

```
FILE* log_file;

void handle_sigint(int sig);
void thank_most_orders(Worker* workers, int size, const char* role);
```

- **Log File**: Records each significant event, such as new orders, cooking start and end, deliveries, and shutdown summary.
- **Signal Handling**: Captures SIGINT for graceful shutdown, ensuring all threads are terminated and resources are cleaned up.
- **Thank You Messages**: Displays and logs thank you messages for the most productive workers.

Functions:

- `void thank most orders(Worker* workers, int size, const char* role)`: Identifies and thanks the workers with the highest order counts.

3 Synchronization Mechanisms

Synchronization is crucial for managing multiple clients and workers concurrently. We use mutexes and condition variables to ensure thread safety and coordination.

3.1 Mutexes

Mutexes are used to protect shared data structures, such as order queues.

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);

// Locking
pthread_mutex_lock(&mutex);
// Critical section
pthread_mutex_unlock(&mutex);
```

3.2 Condition Variables

Condition variables are used to block threads until a particular condition is met, such as waiting for new orders in the queue.

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);

// Waiting for a condition
pthread_mutex_lock(&mutex);
while (condition_not_met) {
    pthread_cond_wait(&cond, &mutex);
}
// Critical section
pthread_mutex_unlock(&mutex);

// Signaling a condition
pthread_mutex_lock(&mutex);
condition_met = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

3.3 Example: Cook Thread

The cook thread waits for an order to be available, processes it, and then moves it to the delivery queue.

```
void* cook_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&order_queue.mutex);
        while (order_queue.size == 0) {
            pthread_cond_wait(&order_queue.cond, &order_queue.mutex);
        }
        Order* order = dequeue(&order_queue);
        pthread_mutex_unlock(&order_queue.mutex);

        // Simulate cooking
        sleep(cooking_time);

        pthread_mutex_lock(&delivery_queue.mutex);
        enqueue(&delivery_queue, order);
        pthread_cond_signal(&delivery_queue.cond);
        pthread_mutex_unlock(&delivery_queue.mutex);
    }
}
```

3.4 Example: Delivery Thread

The delivery thread waits for cooked orders and delivers them to clients.

```
void* delivery_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&delivery_queue.mutex);
        while (delivery_queue.size == 0) {
            pthread_cond_wait(&delivery_queue.cond, &delivery_queue.mutex);
        }
        Order* order = dequeue(&delivery_queue);
        pthread_mutex_unlock(&delivery_queue.mutex);

        // Simulate delivery
        sleep(delivery_time);

        // Update delivery count
        pthread_mutex_lock(&stats_mutex);
        deliveries++;
        pthread_mutex_unlock(&stats_mutex);
    }
}
```

4 Result and Test Cases

4.1 Results

The implementation successfully handles multiple client connections, processes orders efficiently using cook and courier threads, and maintains a log of all activities. Upon shutdown, a summary report of total orders and deliveries, along with a thank you message to the most productive workers, is generated.

4.2 Test Cases

- **Single Client Order**: Connect a single client to the server, place an order, verify processing and delivery, and check log entries.
- **Multiple Client Orders**: Connect multiple clients, place orders, verify processing and delivery for all, and check log entries.
- **High Load**: Simulate a high number of clients and orders, ensure stability, and verify logs.
- **Graceful Shutdown**: Send SIGINT to the server, ensure all threads terminate gracefully, and check the summary report and logs.

4.3 Screenshots

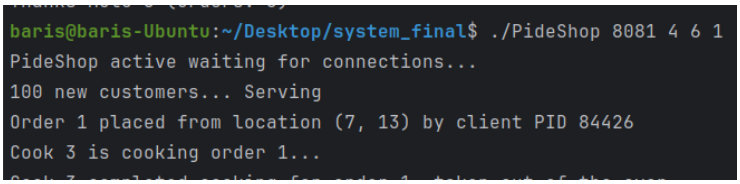

Figure 1: Server initialization and handling multiple client orders.

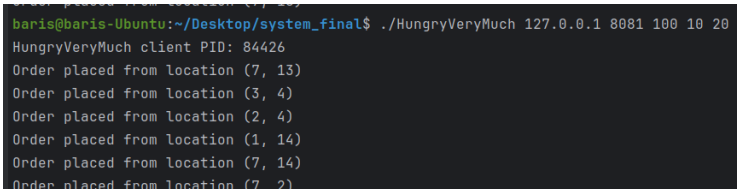

Figure 2: Client connections and order placements from various locations.

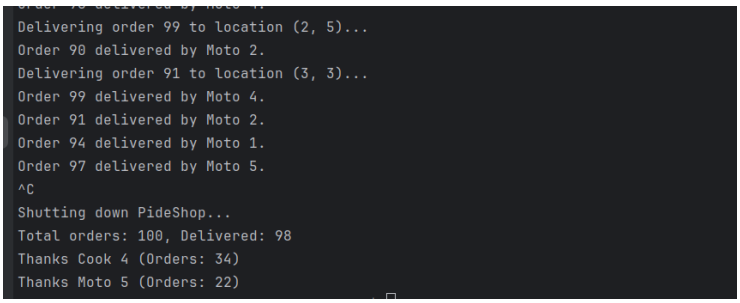

Figure 3: Server processing orders and delivering them, with shutdown summary.

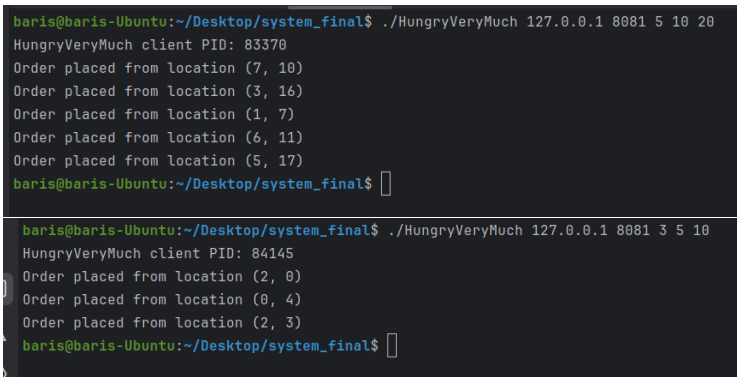

Figure 4: Multiple clients placing orders simultaneously.

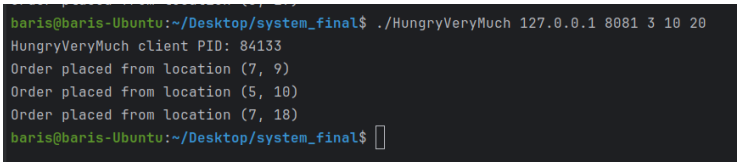

Figure 5: Client placing a few orders for testing.

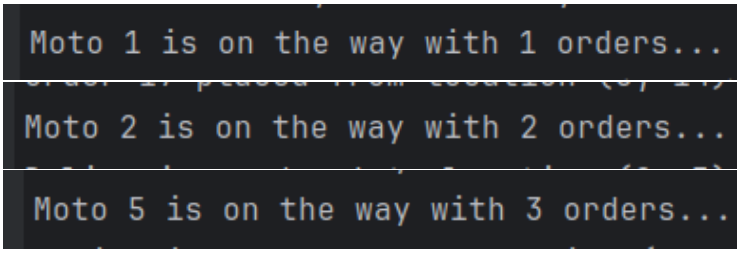

Figure 6: Moto could carry 1, 2 or 3 orders at the same time.

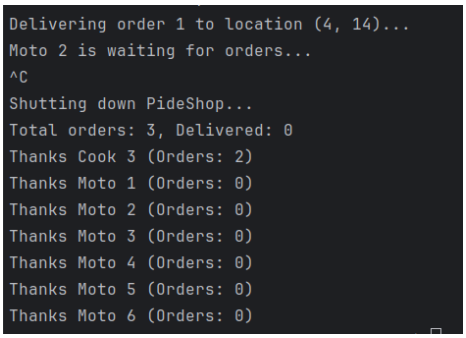

Figure 7: Graceful shutdown with summary report and thank you messages for top performers.

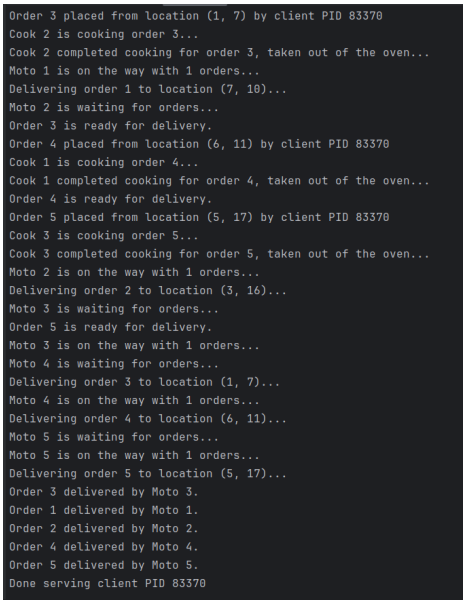

Figure 8: Order placement and processing by different cooks and couriers.

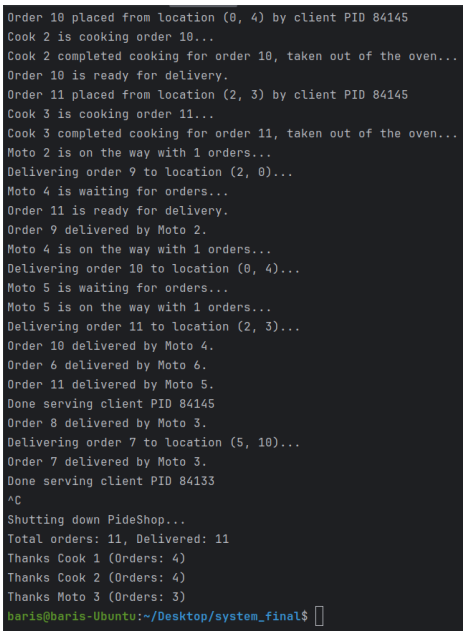

Figure 9: Server handling orders, deliveries, and shutdown summary.

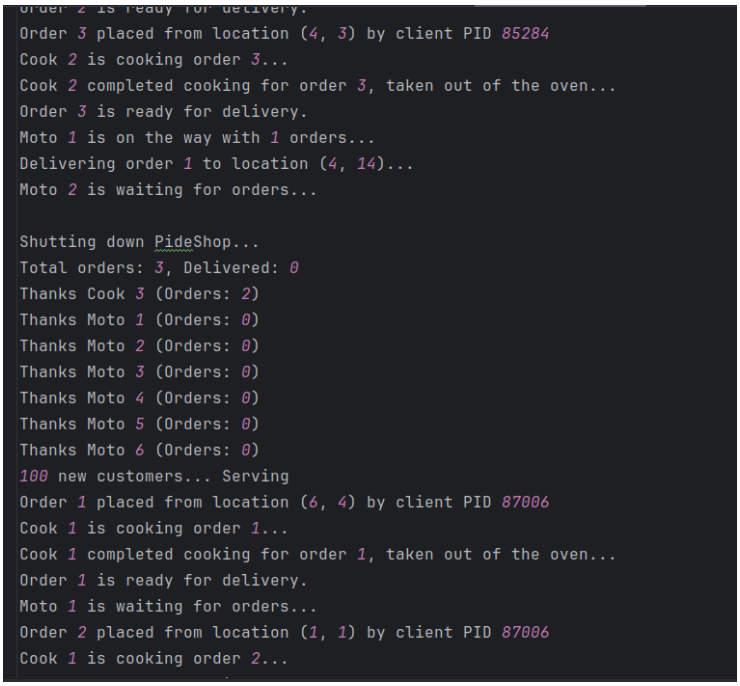

Figure 10: Part of Log file.