

\_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp

**Gebze Technical University**  
**Computer Engineering**

---

# SYSTEM PROGRAMMING

EMRE YILMAZ: 1901042606

HOMEWORK FINAL PROJECT

---

**Professor: Erkan Zergeroglu**

1901042606

June 15, 2024

# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>General Notes</b>                              | <b>3</b> |
| <b>2</b> | <b>How to Run</b>                                 | <b>4</b> |
| <b>3</b> | <b>Server Side</b>                                | <b>4</b> |
| 3.1      | Connection . . . . .                              | 4        |
| 3.1.1    | Server Initialization . . . . .                   | 4        |
| 3.1.2    | Handling Client Connections . . . . .             | 6        |
| 3.1.3    | Handle Client Thread . . . . .                    | 6        |
| 3.1.4    | Summary . . . . .                                 | 7        |
| 3.2      | Orders . . . . .                                  | 7        |
| 3.3      | Cooks . . . . .                                   | 8        |
| 3.4      | Couriers . . . . .                                | 9        |
| 3.5      | Manager Thread . . . . .                          | 9        |
| 3.5.1    | Monitor Shutdown and Cancellation Flags . . . . . | 9        |
| 3.5.2    | Order Assignment to Delivery Personnel . . . . .  | 9        |
| 3.5.3    | Handle All Orders Delivered . . . . .             | 10       |
| 3.5.4    | Wait for New Orders . . . . .                     | 10       |
| 3.5.5    | Assign Orders to Cooks . . . . .                  | 10       |
| 3.5.6    | Assign Orders to Delivery Personnel . . . . .     | 10       |
| 3.6      | Cook Thread . . . . .                             | 10       |
| 3.6.1    | General Flow . . . . .                            | 10       |
| 3.6.2    | Shutdown and Cancellation Handling . . . . .      | 11       |
| 3.6.3    | Waiting for Orders . . . . .                      | 11       |
| 3.6.4    | Process Orders . . . . .                          | 11       |
| 3.6.5    | Simulation of Preparing Time . . . . .            | 11       |
| 3.6.6    | Managing Apparatus . . . . .                      | 12       |
| 3.6.7    | Oven Opening Management . . . . .                 | 12       |
| 3.6.8    | Placing Order in Oven . . . . .                   | 12       |
| 3.6.9    | Simulating Cooking Time . . . . .                 | 12       |
| 3.6.10   | Removing Order from Oven . . . . .                | 12       |
| 3.6.11   | Marking Order as Completed . . . . .              | 12       |
| 3.7      | Courier Thread . . . . .                          | 13       |
| 3.7.1    | General Flow . . . . .                            | 13       |
| 3.7.2    | Monitor Shutdown and Cancellation Flags . . . . . | 13       |
| 3.7.3    | Wait for Orders . . . . .                         | 13       |
| 3.7.4    | Collect Orders for Delivery . . . . .             | 14       |
| 3.7.5    | Simulate Delivery Process . . . . .               | 14       |
| 3.7.6    | Update Delivery Status . . . . .                  | 14       |
| 3.8      | CTRL-C Signal Handling . . . . .                  | 14       |
| 3.9      | Canceling Orders . . . . .                        | 14       |

|          |   |           |
|----------|---|-----------|
| <b>4</b> | <b>Client Side</b>  | <b>15</b> |
| 4.1      | Connection . . . . .  | 15        |
| 4.2      | Threads . . . . .   | 15        |
| 4.3      | Why Prefer Separate Threads for Each Client . . . . .                               | 16        |
| 4.4      | Order Cancellation . . . . .  | 16        |
| 4.5      | CTRL-C Signal Handling . . . . .  | 16        |
| <b>5</b> | <b>Test Cases</b>   | <b>17</b> |
| 5.1      | 7 Cook, 3 Courier, 10 Orders . . . . .  | 17        |
| 5.2      | 5 Cook, 5 Courier, 20 Orders . . . . .  | 17        |
| 5.3      | 5 Cook, 5 Courier, 50 Orders . . . . .  | 19        |
| 5.4      | 6 Cook, 4 Courier, 100 Orders . . . . .   | 19        |
| 5.5      | Order Cancel Test . . . . .   | 20        |
| 5.6      | Order Cancel Test - Repeated Runs . . . . .   | 20        |
| 5.7      | CTRL-C to Server Side . . . . .   | 21        |
| 5.8      | Valgrind Result in Normal Flow of Program . . . . .                                 | 21        |
| 5.9      | Valgrind Result Before All Orders Delivered . . . . .                               | 22        |
| 5.10     | Valgrind Result of Client Side . . . . .  | 22        |
| 5.11     | Valgrind Result of Closing Server After Order Cancellation in Server Side . . . . . | 23        |

# 1 General Notes

1. I strongly suggest firstly check Test Cases section of report.
2. I tried test cases for memory leak as much possible. I put them to Test results. I do not encounter any leaks.
3. Valgrind output may take time. Be patient please. Signals and outputs may come late a little bit.
4. For Matrix calculation, I get the implementation from external resources, I do not want to use a library because of probable compatibility issues. This calculated time is fast comparing delivery times of Pide Shop. If you want to update those matrix calculation line, go to line 471 of src/Server/PideHouse.c. You can change sleep time what you wish. I tried, it works for all cases.
5. You should look at How to Run section to learn how you should give inputs. I also put usage to code. In test cases, there was no `port` input, I added it later, you should give `port` input in addition to inputs you saw in test cases.
6. To make the system more realistic, I create threads for each connection in client side. They communicate with the server separately. Server also creates threads, read message and terminate threads.
7. I have 2 thread pools as cook and delivery men. Also, 1 thread for manager.
8. Manager checks orders and assigns them to convenient personel.
9. Server will be open until you send CTRL-C signal.
10. You can examine logs to understand the system.
11. The most difficult part was the memory leaks and signals. I gave my best those parts. You can examine report and code.
12. I developed the project in Ubuntu 18.04 system.
13. For any problem, feel free to call me whenever you want. I am ready for a demo. [eyilmaz2019@gtu.edu.tr](mailto:eyilmaz2019@gtu.edu.tr), +905319346629
14. Best Regards.

## 2 How to Run

For server side:

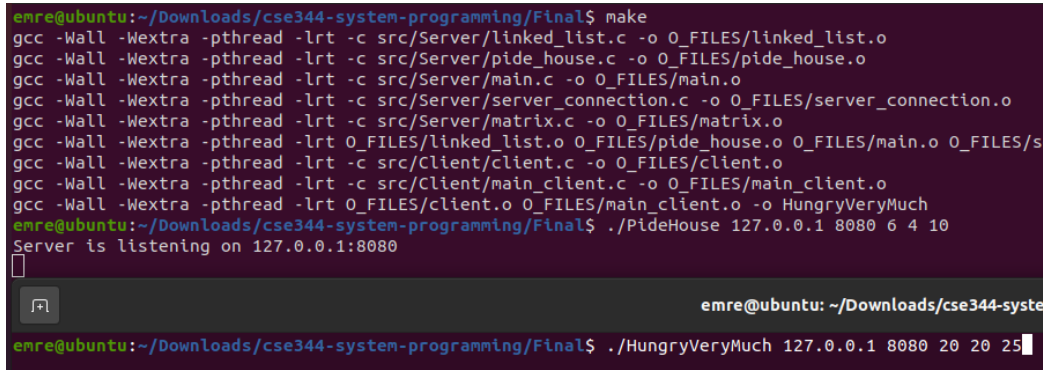
```
./PideHouse 127.0.0.1 8080 6 4 10
```

Usage: ./PideHouse <ip\_address> <port> <n\_cooks> <m\_delivery\_persons> <delivery\_speed>

For client side:

```
/HungryVeryMuch 127.0.0.1 8080 20 20 25
```

Usage: ./HungryVeryMuch <server\_ip> <port> <num\_clients> <p> <q>



```
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ make
gcc -Wall -Wextra -pthread -lrt -c src/Server/linked_list.c -o O_FILES/linked_list.o
gcc -Wall -Wextra -pthread -lrt -c src/Server/pide_house.c -o O_FILES/pide_house.o
gcc -Wall -Wextra -pthread -lrt -c src/Server/main.c -o O_FILES/main.o
gcc -Wall -Wextra -pthread -lrt -c src/Server/server_connection.c -o O_FILES/server_connection.o
gcc -Wall -Wextra -pthread -lrt -c src/Server/matrix.c -o O_FILES/matrix.o
gcc -Wall -Wextra -pthread -lrt O_FILES/linked_list.o O_FILES/pide_house.o O_FILES/main.o O_FILES/s
gcc -Wall -Wextra -pthread -lrt -c src/Client/client.c -o O_FILES/client.o
gcc -Wall -Wextra -pthread -lrt -c src/Client/main_client.c -o O_FILES/main_client.o
gcc -Wall -Wextra -pthread -lrt O_FILES/client.o O_FILES/main_client.o -o HungryVeryMuch
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 8080 6 4 10
Server is listening on 127.0.0.1:8080
[ ]
emre@ubuntu: ~/Downloads/cse344-syste
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./HungryVeryMuch 127.0.0.1 8080 20 20 25
```

Fig-0: How to Run

## 3 Server Side

### 3.1 Connection

In the server-side implementation of our project, the connection management system is responsible for handling incoming client connections, maintaining active connections, and ensuring efficient communication between the server and the clients. This subsection provides an in-depth overview of the connection handling mechanisms based on the provided source code.

#### 3.1.1 Server Initialization

The server initialization process sets up the essential components required to handle client connections efficiently. The following steps are involved in server initialization:

- **Socket Creation:** A TCP socket is created using the `socket()` function with the `AF_INET` address family and `SOCK_STREAM` type. This socket serves as the endpoint for accepting client connections.

```
server_socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket_fd < 0) {
    perror("Error creating socket");
}
```

```

        exit(EXIT_FAILURE);
    }

```

- **Socket Options:** The `SO_REUSEADDR` socket option is set to allow the server to reuse the address, which is particularly useful for rapid restarts.

```

    int opt = 1;
    if (setsockopt(server_socket_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
        perror("setsockopt failed");
        close(server_socket_fd);
        exit(EXIT_FAILURE);
    }

```

- **Address Configuration:** The server's IP address and port are configured using the `sockaddr_in` structure. This allows the server to bind to a specific IP address and port for listening to incoming connections.

```

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    if (inet_pton(AF_INET, ip_address, &server_addr.sin_addr) <= 0) {
        perror("Invalid address/Address not supported");
        close(server_socket_fd);
        exit(EXIT_FAILURE);
    }

```

- **Binding and Listening:** The server binds the socket to the configured address and port using the `bind()` function. After binding, the server starts listening for incoming connections with a maximum queue length defined by `MAX_CONNECTIONS`.

```

    if (bind(server_socket_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Error binding socket");
        close(server_socket_fd);
        exit(EXIT_FAILURE);
    }

    if (listen(server_socket_fd, MAX_CONNECTIONS) < 0) {
        perror("Error listening on socket");
        close(server_socket_fd);
        exit(EXIT_FAILURE);
    }

```

### 3.1.2 Handling Client Connections

The server handles client connections in a continuous loop, accepting new connections and delegating each to a dedicated thread for individual management:

- **Accepting Connections:** The server uses the `accept()` function to accept new incoming connections. This function blocks until a connection is available, at which point it creates a new socket for communication with the client. It reads the order and terminates. It is efficient.

```
client_connection->socket_fd = accept(server_socket, (struct sockaddr *)&client_connection->addr, &client_connection->addr_len);
if (client_connection->socket_fd < 0) {
    perror("Error accepting connection");
    free(client_connection);
    continue;
}
```

- **Client Management:** Each client connection is encapsulated in a `ClientConnection` structure, which includes details such as the client's socket descriptor and address.

```
client_connections[num_connections++] = client_connection;
```

- **Multi-threading:** A new thread is created for each client connection using `pthread_create()`. This thread handles the communication with the client, including reading client data, processing orders, and sending responses.

```
pthread_create(&client_threads[client_thread_count - 1], NULL, handle_client, client_connection);
```

- **Synchronization:** To ensure thread-safe operations, the server uses mutexes to protect shared resources. This prevents race conditions when multiple threads access or modify global variables.

```
pthread_mutex_lock(&connection_mutex);
// Critical section
pthread_mutex_unlock(&connection_mutex);
```

### 3.1.3 Handle Client Thread

When a server receives a connection request from a client, it creates a socket and then spawns a thread to handle that client. This thread essentially waits for an order from the client. It listens and blocks until a message is received. Upon receiving the message, it parses the message and stores it in an `Order` struct. This `Order` is then added to a linked list that holds all the orders and a signal is sent to manager that an order has come. A message is sent to the client confirming that the order has been received, and the thread terminates.



```
// Place order to Order Linked List
place_order(&order);
```

```
// Send response to client
send_response(client_socket, "Order received and being processed.\n");
```

There's an additional step at this point. If any of the clients send a cancel order signal, a thread is created to handle this signal successfully. I'll explain this in more detail in a later section.

### 3.1.4 Summary

The connection management system of the server is designed to efficiently handle a large number of client connections simultaneously. It employs multi-threading to ensure concurrent client handling, uses robust synchronization techniques to maintain thread safety, and includes comprehensive signal handling for graceful shutdown. This design ensures that the server can manage high traffic efficiently while maintaining stability and responsiveness.

## 3.2 Orders

```
// Structure to represent an order
typedef struct {
    int order_id;
    int customer_id;
    double preparation_time;
    double cooking_time;
    int delivery_time;
    int is_cancelled;
    int cook_id; // ID of the cook who prepared the order
    int delivery_person_id; // ID of the delivery person who delivered the order
    int x; // x-coordinate of the customer
    int y; // y-coordinate of the customer
    int taken; // 1 if the order is taken by a cook, 0 otherwise
    int ready; // 1 if the order is ready for delivery, 0 otherwise
    int socket_fd;
} Order;
// Structure to represent a node in the linked list
typedef struct ListNode {
    Order order;
    struct ListNode *next;
} ListNode;

// Structure to represent the linked list
typedef struct {
    ListNode *head;
    int size;
} LinkedList;
```

Each order in the system is stored in a linked list, which provides the flexibility to dynamically manage orders. The choice of a linked list as the data structure is crucial due to the frequent addition and removal of orders, which requires efficient operations.

Each order contains various fields that keep track of essential information, such as:

**Preparation Time:** The duration it takes for the order to be prepared. **Delivery Time:** The time taken for the order to be delivered to the customer. **Assigned Cook:** The cook responsible for preparing the order. **Assigned Delivery Person:** The delivery person responsible for delivering the order. **Status Flags:** Indicate whether the order has been taken by a cook and assigned to a delivery person. **Socket Information:** Contains the customer's socket address. **Customer Location:** Includes the coordinates of the customer's location. This structure allows for effective management and easy access to the relevant fields of each order. Orders are added to the list by connection threads, which handle the communication with clients. These threads take the order information from the client, load it into an Order struct, and then add it to the linked list. When an order is added to the list, a signal is sent to the manager thread. This signaling ensures that if the manager is waiting on a condition (using `cond_wait`), it wakes up to continue processing, knowing that a new order has arrived.

Overall, this setup enables the system to manage orders efficiently and ensures seamless coordination between different components involved in order handling.

### 3.3 Cooks

```
// Node structure for the queue
typedef struct QueueNode {
    Order* order;
    struct QueueNode* next;
} QueueNode;

// Queue structure
typedef struct {
    QueueNode* front;
    QueueNode* rear;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} OrderQueue;

// Structure to represent a cook
typedef struct {
    int cook_id;
    int total_deliveries;
    double total_earnings;
    int order_id;
    int busy;
    OrderQueue *order_queue;
    pthread_mutex_t cookMutex;
    pthread_cond_t cookCond;
} Cook;
```

This is how cook is represented. Let me mention a few key points: Each cook has a variable that indicates which order they are currently handling. This is shown by `order_id`. Similarly, `busy` shows whether the cook is available. The Manager can make new assignments by checking these fields.

Most importantly, each cook can learn which orders have been assigned to them through a queue

named `order_queue`. This queue holds references to the actual orders, and orders can be accessed via this queue.

Cooks are created at the start of the program and are not destroyed until the program ends.

### 3.4 Couriers

```
// Structure to represent a delivery person
typedef struct {
    int delivery_person_id;
    int total_deliveries;
    double total_earnings;
    int order_id;
    OrderQueue *order_bag;
    int busy;
    int current_delivery_count; // Add this to track the number of orders in the bag
} DeliveryPerson;
```

Here's the explanation of how a delivery person is represented, similar to the cook structure.

This section describes how a delivery person is represented in the server system. The logic is similar to that of a cook. A delivery person has a field (`order_id`) that keeps track of the current order they are handling. Additionally, they have an `order_queue`, which is a queue of orders assigned to them. This structure allows the delivery person to access and manage the orders they are responsible for, ensuring efficient tracking and processing of their assigned deliveries.

This structure enables the delivery personnel to effectively manage their tasks and helps the server system coordinate between the delivery personnel and the orders.

Delivery personels are created at the start of the program and are not destroyed until the program ends.

### 3.5 Manager Thread

The `manager_function` thread is responsible for managing the overall workflow of the system, particularly focusing on assigning orders to cooks and delivery persons. It continuously monitors the state of orders and handles their distribution and progression through the system.

#### 3.5.1 Monitor Shutdown and Cancellation Flags

The thread periodically checks the `shutdown_flag` and `cancel_order_flag`. If either flag is set, the wakeup function is called, and the thread exits its loop, preparing for a safe shutdown or order cancellation.

#### 3.5.2 Order Assignment to Delivery Personnel

If the number of remaining orders is fewer than three and the number of prepared orders is not equal to the total order count (`order_count`), the manager signals all delivery personnel by calling `pthread_cond_signal` on their respective condition variables. This triggers the delivery personnel to start processing the available orders.

### 3.5.3 Handle All Orders Delivered

When all orders have been delivered (`total_delivered_orders == order_count`) and there are orders in the system, the manager sends a "All orders are delivered." message to the client, prints the statistics of the operations, logs the disconnection, and exits the loop.

### 3.5.4 Wait for New Orders

If there are no new orders (`order_count == 0`), the thread waits on the `order_cond` condition variable, unlocking the mutex during the wait. This effectively puts the manager thread in a wait state until new orders arrive or a shutdown or cancel signal is received.

### 3.5.5 Assign Orders to Cooks

The manager searches for an appropriate order using `findAppropriateOrder`. If an order is found, it looks for an available cook using `findAppropriateCook`. If a cook is available, the order is enqueued into the cook's order queue, and the cook is signaled to start processing the order. The order is marked as taken, and a message is sent to the client indicating the order has been sent to the cook.

### 3.5.6 Assign Orders to Delivery Personnel

The thread traverses the list of orders and checks if there are any orders ready (`ready == 1`), taken (`taken == 1`), but not yet assigned to a delivery person (`delivery_person_id == -1`). For such orders, it looks for an available delivery person using `findAppropriateDeliveryPerson`. If found, the order is enqueued into the delivery person's order bag. If the delivery person reaches their delivery capacity (`DELIVERY_CAPACITY = 3` in our case), they are signaled to start delivering orders. Messages are sent to clients as orders are assigned to delivery personnel.

## 3.6 Cook Thread

The `cook_function` is a continuous loop where each cook waits for orders, processes them, and manages shared resources like apparatus and oven space. The function also handles shutdown and cancellation signals.

### 3.6.1 General Flow

1. First, it checks if there is an assigned order for it. If not, it blocks and waits.
2. When an order arrives, it prepares the food first.
3. Then it takes the paddle (kürek). If it can't take the paddle, it blocks and waits.
4. After taking the paddle, it checks the oven entry. If no suitable entry is available, it waits for an entry.
5. After acquiring both the entry and the paddle, it checks if there is space in the oven. If there is no space, it releases both the paddle and the entry and waits for space to open up in the oven.
6. Once space is available, it places the food in the oven.
7. When the food is cooked, it takes the paddle, acquires a suitable entry, and removes the food

### 3.6.2 Shutdown and Cancellation Handling

This explanation will be much more wider in later sections. This is just how it is handled in thread.

- Shutdown: If shutdown\_flag is set, the cook wakes up from any waiting state, logs a message, and exits.
- Cancellation: If cancel\_order\_flag is set, the cook wakes up, handles cancellation, and exits.

```
    if (shutdown_flag) {
        wakeup();
        sprintf(buffer, "Cook %d exiting...\n", cook->cook_id);
        printf("%s", buffer);
        writeLog(buffer);
        pthread_exit(NULL);
    }
    if (cancel_order_flag) {
        wakeup();
        pthread_exit(NULL);
    }
}
```

### 3.6.3 Waiting for Orders

The cook locks its mutex (cookMutex) and waits for an order to be available in the order\_queue. If no orders are available and no shutdown or cancel signals are set, it waits on a condition variable (cookCond).

```
    pthread_mutex_lock(&cook->cookMutex);
while (cook->order_queue->front == NULL && shutdown_flag == 0 && cancel_order_flag == 0) {
    pthread_cond_wait(&cook->cookCond, &cook->cookMutex);
}
```

### 3.6.4 Process Orders

Once an order is available, it is dequeued from order\_queue. If the order is null (possibly due to spurious wakeups), the cook continues to wait for valid orders.

```
    Order *order = dequeue(cook->order_queue);
pthread_mutex_unlock(&cook->cookMutex);
if (order == NULL) {
    continue;
}
```

### 3.6.5 Simulation of Preparing Time

The cook logs that it is preparing the order and then simulates the preparation time by calculating the psuedo-inverse of a 30 by 40 matrix having complex elements. I got that matrix calculation from external resources. I do not prefer library since compatibility issues.

### 3.6.6 Managing Apparatus

The cook locks the `apparatus_mutex` and waits for an available apparatus. If no apparatus is available, it waits on a condition variable (`apparatus_cond`).

### 3.6.7 Oven Opening Management

The cook waits for an opening to put the order into the oven. It waits using a condition variable (`oven_putting_opening_cond`) for oven openings

### 3.6.8 Placing Order in Oven

After acquiring an oven opening, the cook checks for available oven space and places the order in the oven. If there is no space, it blocks and wait for place.

### 3.6.9 Simulating Cooking Time

The cook simulates the cooking time by sleeping for half the initial preparation time.

### 3.6.10 Removing Order from Oven

The cook waits for an opening to remove the order from the oven, then removes it, and releases the apparatus and oven space.

### 3.6.11 Marking Order as Completed

The cook marks the order as completed, increments counters for available cooks and prepared orders, and sends a response to the client.

- Resource Management: Cooks wait for orders and manage shared resources (apparatus, oven space) using mutexes and condition variables.
- Order Handling: They handle orders by simulating preparation and cooking times, ensuring synchronization with other cooks.
- Signal Handling: The function handles shutdown and order cancellation by exiting threads safely.
- Order Completion: Upon finishing an order, the cook updates system state and informs the client.
- This process ensures that cooks operate efficiently, handle orders properly, and maintain synchronization with other cooks and resources in the system.

```
pthread_mutex_lock(&order_mutex);
order->ready = 1;
available_cooks++;
prepared_order++;
cook->total_deliveries++;
pthread_cond_signal(&cooked_cond);
total_prepared_orders++;
send_response(order->socket_fd, buffer);
```

```
cook->busy = 0;
pthread_mutex_unlock(&order_mutex);
```

### 3.7 Courier Thread

The `delivery_function` thread is responsible for managing the delivery of orders by each delivery person in the system. Each delivery person runs this function to handle the collection, delivery, and reporting of orders assigned to them.

#### 3.7.1 General Flow

- Manager looks for couriers that have empty bags.
- When it find an empty bag, it assigns the order to that delivery.
- When that delivery has 3 bags, manager sends a signal to courier, and it is triggered. After that, it starts to delivery

#### 3.7.2 Monitor Shutdown and Cancellation Flags

The thread continuously checks the `shutdown_flag` and `cancel_order_flag`. If either flag is set, the `wakeup` function is called to notify all waiting threads, and the thread exits after logging the shutdown event.

Particularly in very critical areas, cancel order and shutdown signals are checked by cooks. These signals are checked before and after `cond wait`, at the beginning of the loop, and in critical places such as just before leaving from shop, to see if the order has been canceled or if the server has shut down.

**This explanation will be much more wider in later sections. This is just how it is handled in thread.**

#### 3.7.3 Wait for Orders

The thread locks the mutex associated with the delivery person's order bag and waits for orders to be added to the bag if the conditions are met: either the bag is empty or the current delivery count is less than the delivery capacity and there are at least 3 remaining orders:

```
while ((delivery_person->order_bag->front == NULL
|| (delivery_person->current_delivery_count < DELIVERY_CAPACITY &&
countRemainingOrders(&orders) >= 3))
&& shutdown_flag == 0
&& cancel_order_flag == 0)
```

In this way, they are waiting for bag has 3 orders. If bag does not have 3 orders and remaining order is less than 3, then it goes for delivery. No need to wait.

It uses `pthread_cond_wait` to block until a condition signal is received indicating that new orders are available. It re-checks the shutdown and cancel flags after waking up.

### 3.7.4 Collect Orders for Delivery

Once woken up and there are orders in the bag, the delivery person collects up to the `DELIVERY_CAPACITY` number of orders from the order bag. This is done by dequeuing orders from the bag into a local array `orders_to_deliver`.

### 3.7.5 Simulate Delivery Process

For each collected order, the thread calculates the Euclidean distance from the Pide House to the customer's location to determine the delivery time. It simulates the delivery time using `usleep` and logs the delivery process, including the distance and the order being delivered. After delivery, it updates the order's delivery time, increments the delivery person's earnings, and sends a response to the client indicating that their order has been delivered. Also, it updates the current location for next delivery.

### 3.7.6 Update Delivery Status

After delivering the collected orders, the delivery person resets their delivery count and marks themselves as not busy. It also updates the total number of delivered orders and logs the current delivery statistics.

## 3.8 CTRL-C Signal Handling

CTRL-C signal is one of the most critical aspects of this project. There are many running threads and extensive dynamic memory allocation.

First, the signal handler sends signals to all condition variables so that all sleeping threads are awakened. All threads have checks for a flag indicating that the CTRL-C signal has arrived. These checks are placed in the most critical areas: before and after `cond_wait`, at the beginning of loops, and at every point where a new task is started. Threads that see the flag set in this check immediately stop their work and terminate. Subsequently, all memory is freed one by one, sockets are closed, and the program exits. As you can see the output in the figure below, you can also test this yourself.

Particularly in very critical areas, cancel order and shutdown signals are checked by cooks. These signals are checked before and after `cond wait`, at the beginning of the loop, and in critical places such as just before placing the food in the oven, to see if the order has been canceled or if the server has shut down.

## 3.9 Canceling Orders

When a client sends a cancel order message, the server catches this in the `handle_client` thread. As soon as this message is detected, the `is_cancelled` flag is set, and the system waits for all threads to join.

All cooks and delivery personnel who receive the message, as well as the manager, stop their work as soon as possible and terminate. Couriers who are already on the road complete their last deliveries, and items in the oven are taken out but not given to the courier. The `handle_client` thread then clears the order lists and the order reference lists of cooks and couriers and reinitializes the



system to prepare everyone for new orders. I would like to carefully note that the statistics are not reset.

## 4 Client Side

This section details the client-side architecture, which is responsible for managing connections to the server, handling multiple client threads, and ensuring proper signal handling for shutdown procedures.

### 4.1 Connection

The client connection process involves initializing multiple clients, each assigned a unique identifier and connection parameters. These clients establish a connection with the server using a predefined IP address and port.

Each client creates a socket and attempts to connect to the server. If the connection fails, an error is reported and handled gracefully.

```
void initialize_clients(int num_clients, char *server_ip, int p, int q) {
    clients = (Client *)malloc(num_clients * sizeof(Client));
    // Error handling and socket creation
    for (int i = 0; i < num_clients; i++) {
        clients[i].socket_fd = socket(AF_INET, SOCK_STREAM, 0);
        clients[i].server_addr.sin_family = AF_INET;
        clients[i].server_addr.sin_port = htons(8080);
        inet_pton(AF_INET, server_ip, &clients[i].server_addr.sin_addr);
        // Random coordinates
        clients[i].x = rand() % p;
        clients[i].y = rand() % q;
    }
}
```

### 4.2 Threads

To handle multiple clients concurrently, each client operates in its own thread. This approach allows for independent management of each client's connection and communication with the server.

Each client thread establishes the connection, sends orders, and listens for responses from the server. The threads can also handle specific responses, such as delivery confirmations, independently.

```
void *client_function(void *arg) {
    Client *client = (Client *)arg;
    client->socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    connect(client->socket_fd, (struct sockaddr *)&client->server_addr, sizeof(client->server_addr));
    send_order(client);
    // Receiving responses
}
```

### 4.3 Why Prefer Separate Threads for Each Client

I thought this approach was more realistic. This way, messages can be sent to and received from each client individually. It is suitable for future work. It might cause the program to run slowly, but considering the scenario, I found it more appropriate to have different threads. Using separate threads for each client has several advantages:

- **Concurrency:** Clients can send and receive messages simultaneously, enhancing system responsiveness.
- **Isolation:** Issues in one client's communication do not affect others, allowing for independent handling of each client.
- **Scalability:** The system can dynamically accommodate multiple clients by creating new threads as needed.

This design ensures that each client manages its connection lifecycle independently, improving overall system robustness and flexibility.

### 4.4 Order Cancellation

Order cancellation is handled by checking a flag that indicates whether the client should stop sending orders and terminate the connection gracefully. When this flag is set, the client sends a special message to the server indicating disconnection.

```
void client_function2(const char *server_ip) {
    Client client;
    client.socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    inet_pton(AF_INET, server_ip, &client.server_addr.sin_addr);
    connect(client.socket_fd, (struct sockaddr *)&client.server_addr, sizeof(client.server_addr));
    send(client.socket_fd, "-1 -1 -1 -1 -1 -1 -1 -1 -1", strlen("-1 -1 -1 -1 -1 -1 -1 -1 -1"), 0);
    close(client.socket_fd);
}
```

Server gives information about order is cancelled and it could not deliver all orders.

After getting this message, system initially make the `cancel_order_flag = 1`. Then, send signals to all condition variables to wake all threads up. Threads wake up and terminate. Then, we join all threads. After that, we destroy and free cooks' and delivery men's mutexes, cond vars and orders. After that, we reinitilize the system. Thread memories are not freed. They just terminated. And then, threads are created again. In this way, we can continue with new customer.

### 4.5 CTRL-C Signal Handling

Signal handling is implemented to manage graceful shutdowns when the program receives a termination signal (such as CTRL-C). The signal handler sets a flag to stop running and ensures all client threads are joined before the program exits.

```
void handle_signal(int sig) {
    printf("\nCaught signal %d (SIGINT), cleaning up...\n", sig);
    keep_running = 0;
    client_function2(server_ip);
}
```

```

    for (int i = 0; i < num_clients; i++) {
        pthread_join(threads[i], NULL);
    }
    free(threads);
    cleanup();
    exit(EXIT_SUCCESS);
}

```

This handler ensures that all client threads and sockets are properly closed, preventing resource leaks and ensuring a clean shutdown process. It waits for all threads and after that frees and releases all the resources.

## 5 Test Cases

### 5.1 7 Cook, 3 Courier, 10 Orders

```

emre@ubuntu:~/Downloads/cse344-system-programming/Final$ make
gcc -Wall -Wextra -pthread -lrt -c src/Server/server_connection.c -o 0_FILES/server_connection.o
gcc -Wall -Wextra -pthread -lrt 0_FILES/linkedList.o 0_FILES/pide_house.o 0_FILES/main.o 0_FILES/server_connection.o 0_FILES/matrix.o -ln -o PideHouse
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 7 3 10
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

All orders are delivered successfully!
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 0 orders
Cook 1 cooked 0 orders
Cook 2 cooked 0 orders
Cook 3 cooked 1 orders
Cook 4 cooked 1 orders
Cook 5 cooked 2 orders
Cook 6 cooked 1 orders
Delivery Person Statistics:
Delivery Person 0 delivered 5 orders and earned 100.00 TL
Delivery Person 1 delivered 0 orders and earned 0.00 TL
Delivery Person 2 delivered 0 orders and earned 0.00 TL
Total orders: 5
Total prepared orders: 5
Total delivered orders: 5
Thanks our best cook-5 has cooked 2 orders
Thanks our best delivery person-0 has delivered 5 orders
Active for new clients...

emre@ubuntu:~/Downloads/cse344-system-programming/Final$
Client 0 received: Order received and being processed.
Client 3 received: Order 2 is prepared by cook 6
Client 4 received: Delivery person 0 delivering order 1 over a distance of 8.06 km
Client 0 received: Order 4 is prepared by cook 3
Client 2 received: Order 3 is prepared by cook 5
Client 4 received: Your order is delivered.
Client 1 received: Delivery person 0 delivering order 0 over a distance of 1.41 km
Client 1 received: Your order is delivered.
Client 3 received: Delivery person 0 delivering order 2 over a distance of 10.82 km
Client 3 received: Your order is delivered.
Client 0 received: Delivery person 0 delivering order 4 over a distance of 12.17 km
Client 0 received: Your order is delivered.
Client 2 received: Delivery person 0 delivering order 3 over a distance of 12.65 km
Client 2 received: Your order is delivered.
All customers served. Client generator program will be closed successfully. You can start it again!
emre@ubuntu:~/Downloads/cse344-system-programming/Final$

```

Fig-1: Test-1

### 5.2 5 Cook, 5 Courier, 20 Orders

```

emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 5 5 5
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

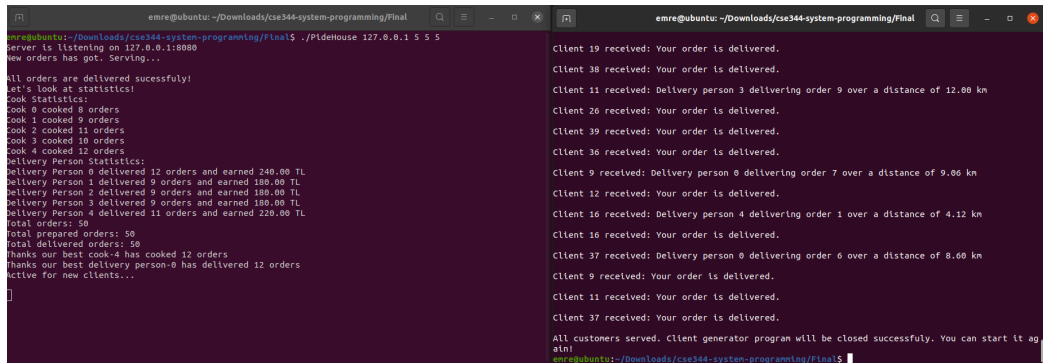
All orders are delivered successfully!
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 4 orders
Cook 1 cooked 3 orders
Cook 2 cooked 4 orders
Cook 3 cooked 4 orders
Cook 4 cooked 5 orders
Delivery Person Statistics:
Delivery Person 0 delivered 6 orders and earned 120.00 TL
Delivery Person 1 delivered 6 orders and earned 120.00 TL
Delivery Person 2 delivered 6 orders and earned 120.00 TL
Delivery Person 3 delivered 2 orders and earned 40.00 TL
Delivery Person 4 delivered 2 orders and earned 0.00 TL
Total orders: 20
Total prepared orders: 20
Total delivered orders: 20
Thanks our best cook-4 has cooked 5 orders
Thanks our best delivery person-0 has delivered 6 orders
Active for new clients...

emre@ubuntu:~/Downloads/cse344-system-programming/Final$
Client 15 received: Delivery person 1 delivering order 14 over a distance of 3.16 km
Client 13 received: Your order is delivered.
Client 12 received: Delivery person 0 delivering order 6 over a distance of 7.00 km
Client 16 received: Delivery person 1 delivering order 15 over a distance of 9.90 km
Client 15 received: Your order is delivered.
Client 12 received: Your order is delivered.
Client 10 received: Delivery person 0 delivering order 19 over a distance of 13.00 km
Client 10 received: Delivery person 2 delivering order 5 over a distance of 9.22 km
Client 7 received: Your order is delivered.
Client 10 received: Your order is delivered.
Client 10 received: Your order is delivered.
Client 4 received: Delivery person 2 delivering order 4 over a distance of 6.00 km
Client 10 received: Your order is delivered.
Client 4 received: Your order is delivered.
All customers served. Client generator program will be closed successfully. You can start it again!
emre@ubuntu:~/Downloads/cse344-system-programming/Final$

```

Fig-2: Test-2

## 5.3 5 Cook, 5 Courier, 50 Orders



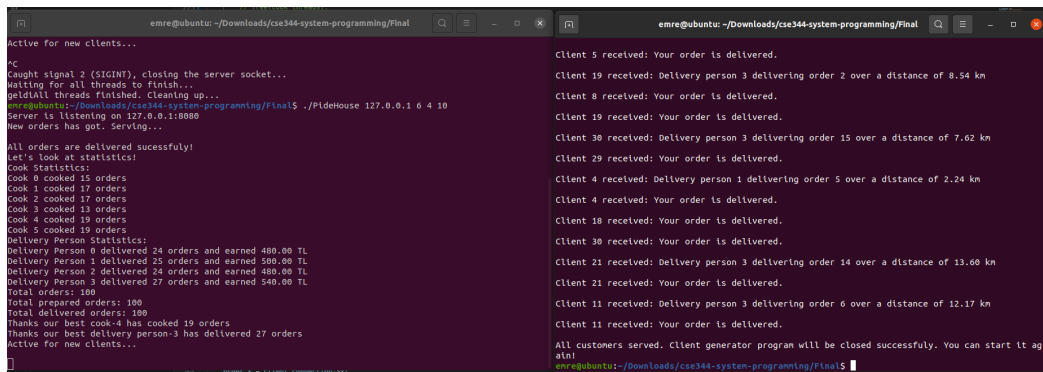
```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 5 5 5
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

All orders are delivered successfully!
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 8 orders
Cook 1 cooked 9 orders
Cook 2 cooked 11 orders
Cook 3 cooked 10 orders
Cook 4 cooked 12 orders
Delivery Person Statistics:
Delivery Person 0 delivered 12 orders and earned 240.00 TL
Delivery Person 1 delivered 9 orders and earned 180.00 TL
Delivery Person 2 delivered 9 orders and earned 180.00 TL
Delivery Person 3 delivered 9 orders and earned 180.00 TL
Delivery Person 4 delivered 11 orders and earned 220.00 TL
Total orders: 50
Total prepared orders: 50
Total delivered orders: 50
Thanks our best cook-4 has cooked 12 orders
Thanks our best delivery person-0 has delivered 12 orders
Active for new clients...

Client 19 received: Your order is delivered.
Client 38 received: Your order is delivered.
Client 11 received: Delivery person 3 delivering order 9 over a distance of 12.00 km
Client 26 received: Your order is delivered.
Client 39 received: Your order is delivered.
Client 36 received: Your order is delivered.
Client 9 received: Delivery person 0 delivering order 7 over a distance of 9.00 km
Client 12 received: Your order is delivered.
Client 16 received: Delivery person 4 delivering order 1 over a distance of 4.12 km
Client 16 received: Your order is delivered.
Client 37 received: Delivery person 0 delivering order 6 over a distance of 8.60 km
Client 9 received: Your order is delivered.
Client 11 received: Your order is delivered.
Client 37 received: Your order is delivered.
All customers served. Client generator program will be closed successfully. You can start it again!
emre@ubuntu:~/Downloads/cse344-system-programming/Final$
```

Fig-3: Test-3

## 5.4 6 Cook, 4 Courier, 100 Orders



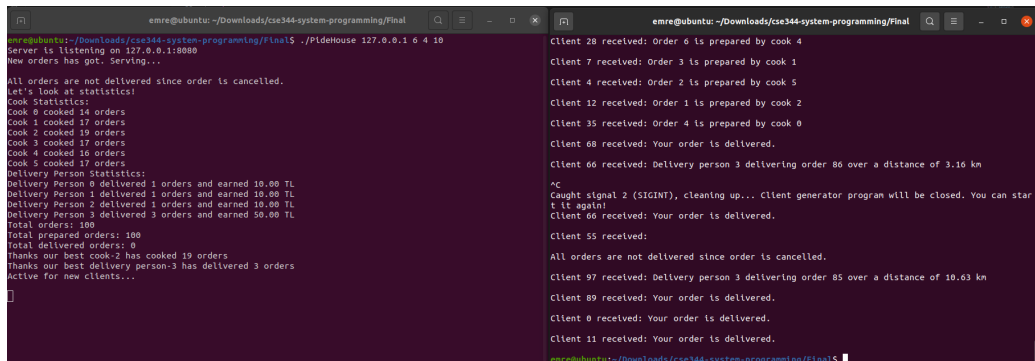
```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 6 4 100
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

All orders are delivered successfully!
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 15 orders
Cook 1 cooked 17 orders
Cook 2 cooked 17 orders
Cook 3 cooked 13 orders
Cook 4 cooked 19 orders
Cook 5 cooked 19 orders
Delivery Person Statistics:
Delivery Person 0 delivered 24 orders and earned 480.00 TL
Delivery Person 1 delivered 25 orders and earned 500.00 TL
Delivery Person 2 delivered 24 orders and earned 480.00 TL
Delivery Person 3 delivered 27 orders and earned 540.00 TL
Total orders: 100
Total prepared orders: 100
Total delivered orders: 100
Thanks our best cook-4 has cooked 19 orders
Thanks our best delivery person-3 has delivered 27 orders
Active for new clients...

Client 5 received: Your order is delivered.
Client 19 received: Delivery person 3 delivering order 2 over a distance of 8.54 km
Client 8 received: Your order is delivered.
Client 19 received: Your order is delivered.
Client 30 received: Delivery person 3 delivering order 15 over a distance of 7.62 km
Client 29 received: Your order is delivered.
Client 4 received: Delivery person 1 delivering order 5 over a distance of 2.24 km
Client 4 received: Your order is delivered.
Client 19 received: Your order is delivered.
Client 30 received: Your order is delivered.
Client 21 received: Delivery person 3 delivering order 14 over a distance of 13.60 km
Client 21 received: Your order is delivered.
Client 11 received: Delivery person 3 delivering order 6 over a distance of 12.17 km
Client 11 received: Your order is delivered.
All customers served. Client generator program will be closed successfully. You can start it again!
emre@ubuntu:~/Downloads/cse344-system-programming/Final$
```

Fig-4: Test-4

## 5.5 Order Cancel Test



```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 6 4 10
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

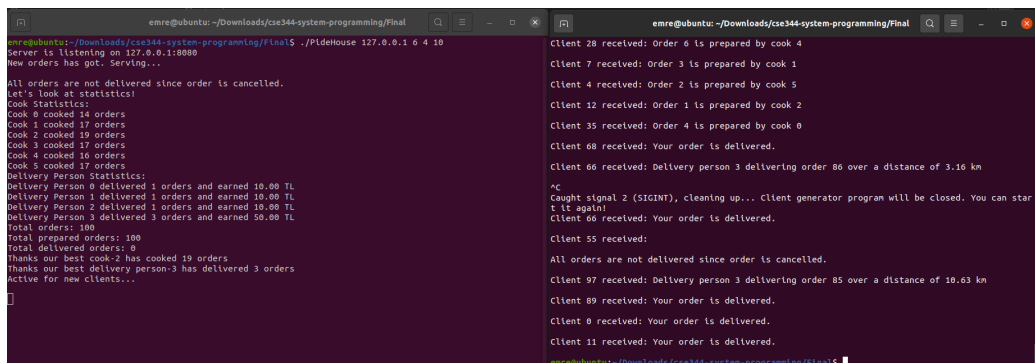
All orders are not delivered since order is cancelled.
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 14 orders
Cook 1 cooked 17 orders
Cook 2 cooked 19 orders
Cook 3 cooked 17 orders
Cook 4 cooked 16 orders
Cook 5 cooked 17 orders
Delivery Person Statistics:
Delivery Person 0 delivered 1 orders and earned 10.00 TL
Delivery Person 1 delivered 1 orders and earned 10.00 TL
Delivery Person 2 delivered 1 orders and earned 10.00 TL
Delivery Person 3 delivered 3 orders and earned 50.00 TL
Total orders: 100
Total prepared orders: 100
Total delivered orders: 0
Thanks our best cook-2 has cooked 19 orders
Thanks our best delivery person-3 has delivered 3 orders
Active for new clients...

Client 28 received: Order 6 is prepared by cook 4
Client 7 received: Order 3 is prepared by cook 1
Client 4 received: Order 2 is prepared by cook 5
Client 12 received: Order 1 is prepared by cook 2
Client 35 received: Order 4 is prepared by cook 0
Client 68 received: Your order is delivered.
Client 66 received: Delivery person 3 delivering order 86 over a distance of 3.16 km
^C
Caught signal 2 (SIGINT), cleaning up... Client generator program will be closed. You can start it again!
Client 66 received: Your order is delivered.
Client 55 received:
All orders are not delivered since order is cancelled.
Client 97 received: Delivery person 3 delivering order 85 over a distance of 10.63 km
Client 89 received: Your order is delivered.
Client 0 received: Your order is delivered.
Client 11 received: Your order is delivered.
```

Fig-5: Cancel Test-1

As you see, until cancel signal came, cooks did their work. However, delivery men cannot deliver since orders are cancelled. Also, it waits for new order. Next test case will be an example of sending new clients after an order cancelation.

## 5.6 Order Cancel Test - Repeated Runs



```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 6 4 10
Server is listening on 127.0.0.1:8080
New orders has got. Serving...

All orders are not delivered since order is cancelled.
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 14 orders
Cook 1 cooked 17 orders
Cook 2 cooked 19 orders
Cook 3 cooked 17 orders
Cook 4 cooked 16 orders
Cook 5 cooked 17 orders
Delivery Person Statistics:
Delivery Person 0 delivered 1 orders and earned 10.00 TL
Delivery Person 1 delivered 1 orders and earned 10.00 TL
Delivery Person 2 delivered 1 orders and earned 10.00 TL
Delivery Person 3 delivered 3 orders and earned 50.00 TL
Total orders: 100
Total prepared orders: 100
Total delivered orders: 0
Thanks our best cook-2 has cooked 19 orders
Thanks our best delivery person-3 has delivered 3 orders
Active for new clients...

Client 28 received: Order 6 is prepared by cook 4
Client 7 received: Order 3 is prepared by cook 1
Client 4 received: Order 2 is prepared by cook 5
Client 12 received: Order 1 is prepared by cook 2
Client 35 received: Order 4 is prepared by cook 0
Client 68 received: Your order is delivered.
Client 66 received: Delivery person 3 delivering order 86 over a distance of 3.16 km
^C
Caught signal 2 (SIGINT), cleaning up... Client generator program will be closed. You can start it again!
Client 66 received: Your order is delivered.
Client 55 received:
All orders are not delivered since order is cancelled.
Client 97 received: Delivery person 3 delivering order 85 over a distance of 10.63 km
Client 89 received: Your order is delivered.
Client 0 received: Your order is delivered.
Client 11 received: Your order is delivered.
```

Fig-6: Cancel Test-2

As you see, first order is cancelled. Only 84 of total orders had been prepared. Then, new program is run and send new clients. Server runs successfully, again. Then, orders cancelled one more time and server handled cancelled orders successfully.

## 5.7 CTRL-C to Server Side

```

emre@ubuntu: ~/Downloads/cse344-system-programming/Final
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 6 4 10
Server is listening on 127.0.0.1:10000
New orders has got. Serving...
^C
Caught signal 2 (SIGINT), closing the server socket...
Waiting for all threads to finish...
All threads finished. Cleaning up...
emre@ubuntu:~/Downloads/cse344-system-programming/Final$

Client 4 received: Order 13 is prepared by cook 2
Client 46 received: Order 11 is prepared by cook 3
Client 30 received: Order 8 is prepared by cook 5
Client 27 received: Order 7 is prepared by cook 4
Client 45 received: Order 6 is prepared by cook 1
Client 33 received: Order 3 is prepared by cook 3
Client 11 received: Order 4 is prepared by cook 4
Client 44 received: Order 5 is prepared by cook 5
Client 89 received: Your order is delivered.
Client 71 received: Delivery person 2 delivering order 95 over a distance of 7.00 km
Client 0 received: Your order is delivered.
Client 76 received: Your order is delivered.
Client 24 received: Your order is delivered.
Client 71 received: Your order is delivered.
Client generator program will be closed successfully. You can start it again!

```

Fig-7: SIGINT Test

As you see, server is terminated successfully. It just burned as it is said. No statistics or no information.

## 5.8 Valgrind Result in Normal Flow of Program

I did not send any signals until all orders are delivered. Then, I sent a CTRL-C signal to server. The valgrind output is below.

```

emre@ubuntu: ~/Downloads/cse344-system-programming/Final
emre@ubuntu:~/Downloads/cse344-system-programming/Final$ ./PideHouse 127.0.0.1 6 4 10
Server is listening on 127.0.0.1:10000
New orders has got. Serving...
^C
All orders are delivered successfully!
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 0 orders
Cook 1 cooked 1 orders
Cook 2 cooked 2 orders
Cook 3 cooked 1 orders
Cook 4 cooked 1 orders
Cook 5 cooked 1 orders
Delivery Person Statistics:
Delivery Person 0 delivered 5 orders and earned 100.00 TL
Delivery Person 1 delivered 0 orders and earned 0.00 TL
Delivery Person 2 delivered 0 orders and earned 0.00 TL
Delivery Person 3 delivered 0 orders and earned 0.00 TL
Total orders: 5
Total prepared orders: 5
Thanks our best cook 1 has cooked 1 orders
Thanks our best delivery person 0 has delivered 5 orders
Active for new clients...
^C
Caught signal 2 (SIGINT), closing the server socket...
Waiting for all threads to finish...
All threads finished. Cleaning up...
emre@ubuntu:~/Downloads/cse344-system-programming/Final$

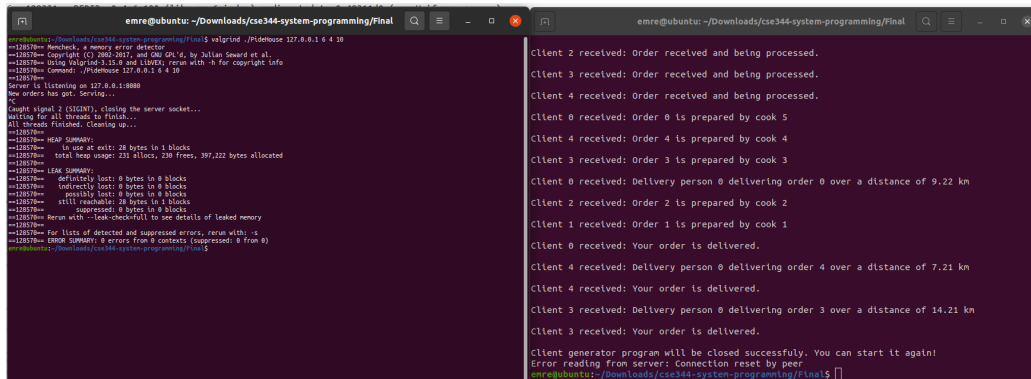
Client 3 received: Order 4 is prepared by cook 4
Client 2 received: Order 3 is prepared by cook 5
Client 4 received: Order 0 is prepared by cook 1
Client 3 received: Delivery person 0 delivering order 4 over a distance of 3.61 km
Client 1 received: Order 2 is prepared by cook 3
Client 0 received: Order 1 is prepared by cook 2
Client 3 received: Your order is delivered.
Client 2 received: Delivery person 0 delivering order 3 over a distance of 8.94 km
Client 2 received: Your order is delivered.
Client 4 received: Delivery person 0 delivering order 0 over a distance of 9.43 km
Client 4 received: Your order is delivered.
Client 1 received: Delivery person 0 delivering order 2 over a distance of 7.81 km
Client 1 received: Your order is delivered.
Client 0 received: Delivery person 0 delivering order 1 over a distance of 8.94 km
Client 0 received: Your order is delivered.
Client generator program will be closed successfully. You can start it again!
emre@ubuntu:~/Downloads/cse344-system-programming/Final$

--126316-- HEAP SUMMARY:
--126316--   in use at exit: 28 bytes in 1 blocks
--126316-- total heap usage: 299 allocs, 298 frees, 310,186 bytes allocated
--126316--
--126316-- LEAK SUMMARY:
--126316--   definitely lost: 0 bytes in 0 blocks
--126316--   possibly lost: 0 bytes in 0 blocks
--126316--   still reachable: 28 bytes in 1 blocks
--126316--   suppressed: 0 bytes in 0 blocks
--126316--
--126316-- For lists of detected and suppressed errors, rerun with: -s
--126316-- ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Fig-8: Valgrind Result - 1

## 5.9 Valgrind Result Before All Orders Delivered



```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$ valgrind ./PrideHouse 127.0.0.1 6 4 10
==128078== Memcheck, a memory error detector
==128078== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==128078== Using Valgrind-3.15.0 and LIBVEX, rerun with -h for copyright info
==128078== Command: ./PrideHouse 127.0.0.1 6 4 10
==128078==
Server is listening on 127.0.0.1:8080
New orders has got. Serving...
^C
Caught signal 2 (SIGINT), cleaning the server socket...
Waiting for all threads to finish...
All threads finished. Cleaning up...
==128078==
==128078== HEAP SUMMARY:
==128078==   in use at exit: 28 bytes in 1 blocks
==128078==   total heap usage: 231 allocs, 230 frees, 397,222 bytes allocated
==128078==
==128078== LEAK SUMMARY:
==128078==   partially lost: 0 bytes in 0 blocks
==128078==   indirectly lost: 0 bytes in 0 blocks
==128078==   possibly lost: 0 bytes in 0 blocks
==128078==   still reachable: 28 bytes in 1 blocks
==128078==   suppressed: 0 bytes in 0 blocks
==128078==
==128078== Error with --leak-check=full to see details of leaked memory
==128078==
==128078== For lists of detected and suppressed errors, rerun with: -s
==128078== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$

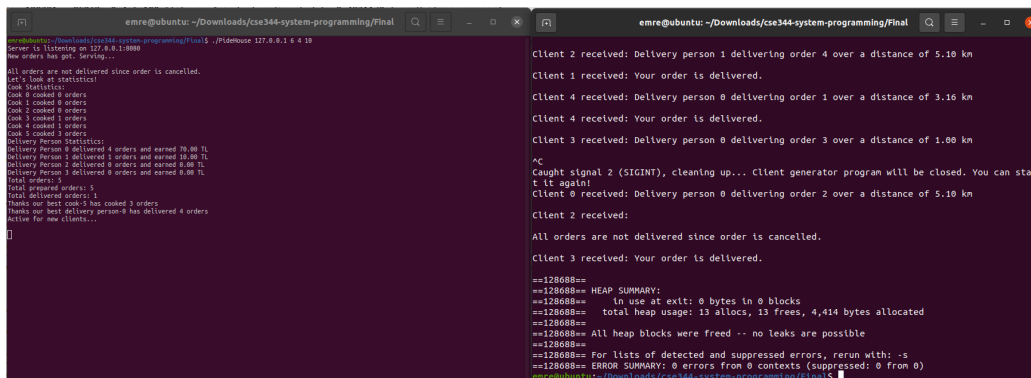
Client 2 received: Order received and being processed.
Client 3 received: Order received and being processed.
Client 4 received: Order received and being processed.
Client 0 received: Order 0 is prepared by cook 5
Client 4 received: Order 4 is prepared by cook 4
Client 3 received: Order 3 is prepared by cook 3
Client 0 received: Delivery person 0 delivering order 0 over a distance of 9.22 km
Client 2 received: Order 2 is prepared by cook 2
Client 1 received: Order 1 is prepared by cook 1
Client 0 received: Your order is delivered.
Client 4 received: Delivery person 0 delivering order 4 over a distance of 7.21 km
Client 4 received: Your order is delivered.
Client 3 received: Delivery person 0 delivering order 3 over a distance of 14.21 km
Client 3 received: Your order is delivered.
Client generator program will be closed successfully. You can start it again!
Error reading from server: Connection reset by peer
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$
```

Fig-9: Valgrind Result - 2

I sent a CTRL-C signal before all orders are delivered. It again shows no leak.

## 5.10 Valgrind Result of Client Side

This is CTRL-C test of client side (Order Cancellation). As you see in server side in the output, it cancels order successfully. Also, there is no leak in client side.



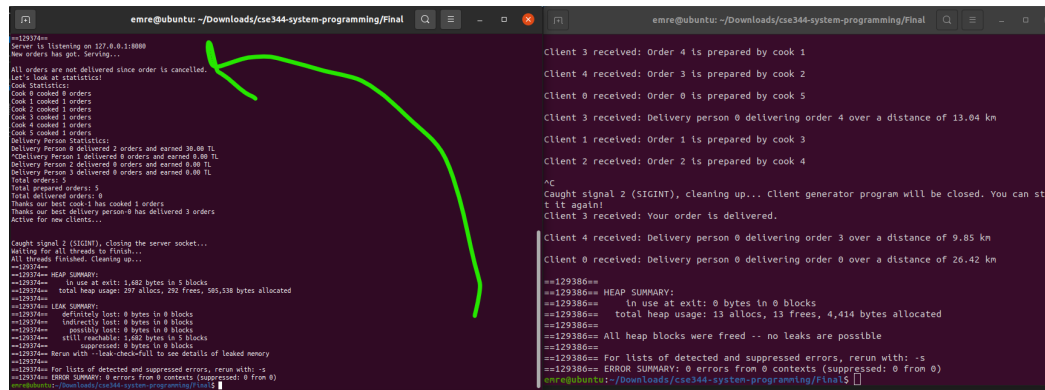
```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$ valgrind ./PrideHouse 127.0.0.1 6 4 10
Server is listening on 127.0.0.1:8080
New orders has got. Settings...
All orders are not delivered since order is cancelled.
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 0 orders
Cook 1 cooked 0 orders
Cook 2 cooked 1 orders
Cook 3 cooked 1 orders
Cook 4 cooked 1 orders
Cook 5 cooked 3 orders
Delivery Person Statistics:
Delivery Person 0 delivered 4 orders and earned 70.00 TL
Delivery Person 1 delivered 1 orders and earned 10.00 TL
Delivery Person 2 delivered 0 orders and earned 0.00 TL
Delivery Person 3 delivered 0 orders and earned 0.00 TL
Total ordered orders: 5
Total delivered orders: 1
Thanks our best cook 5 has cooked 3 orders
Thanks our best delivery person 0 has delivered 4 orders
Active for new clients...
^C
Client 2 received: Delivery person 1 delivering order 4 over a distance of 5.10 km
Client 1 received: Your order is delivered.
Client 4 received: Delivery person 0 delivering order 1 over a distance of 3.16 km
Client 4 received: Your order is delivered.
Client 3 received: Delivery person 0 delivering order 3 over a distance of 1.00 km
^C
Caught signal 2 (SIGINT), cleaning up... Client generator program will be closed. You can start it again!
Client 0 received: Delivery person 0 delivering order 2 over a distance of 5.10 km
Client 2 received:
All orders are not delivered since order is cancelled.
Client 3 received: Your order is delivered.
==128088==
==128088== HEAP SUMMARY:
==128088==   in use at exit: 0 bytes in 0 blocks
==128088==   total heap usage: 13 allocs, 13 frees, 4,414 bytes allocated
==128088==
==128088== All heap blocks were freed -- no leaks are possible
==128088==
==128088== For lists of detected and suppressed errors, rerun with: -s
==128088== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
emre@ubuntu: ~/Downloads/cse344-system-programming/Final$
```

Fig-10: Valgrind Result - 3



## 5.11 Valgrind Result of Closing Server After Order Cancelation in Server Side

This was the most difficult part since when orders has cancelled, I refresh everything. So, it is difficult to track threads and dynamic memories. But I handle it. As you see, even after order cancelization, there is no any leaks.



```
emre@ubuntu: ~/Downloads/cse344-system-programming/Final
==129374==
Server is listening on 127.0.0.1:8080
New orders has get. Serving...
All orders are not delivered since order is cancelled.
Let's look at statistics!
Cook Statistics:
Cook 0 cooked 0 orders
Cook 1 cooked 1 orders
Cook 2 cooked 1 orders
Cook 3 cooked 1 orders
Cook 4 cooked 1 orders
Cook 5 cooked 1 orders
Delivery Person Statistics:
Delivery Person 0 delivered 2 orders and earned 30.00 TL
Delivery Person 1 delivered 8 orders and earned 8.00 TL
Delivery Person 2 delivered 8 orders and earned 0.00 TL
Delivery Person 3 delivered 0 orders and earned 0.00 TL
Total orders: 5
Total prepared orders: 5
Total delivered orders: 8
Thanks our best cook-1 has cooked 1 orders
Thanks our best delivery person-0 has delivered 3 orders
Active for new clients...

Caught signal 2 (SIGINT), closing the server socket...
Waiting for all threads to finish...
All threads finished. Cleaning up...
==129374== HEAP SUMMARY:
==129374==    in use at exit: 1,682 bytes in 9 blocks
==129374==    total heap usage: 207 allocs, 207 frees, 565,538 bytes allocated
==129374== LEAK SUMMARY:
==129374==    definitely lost: 0 bytes in 0 blocks
==129374==    indirectly lost: 0 bytes in 0 blocks
==129374==    possibly lost: 0 bytes in 0 blocks
==129374==    still reachable: 1,682 bytes in 9 blocks
==129374==    suppressed: 0 bytes in 0 blocks
==129374== Run with --leak-check=full to see details of leaked memory
==129374== For lists of detected and suppressed errors, rerun with: -s
==129374== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Client 3 received: Order 4 is prepared by cook 1
Client 4 received: Order 3 is prepared by cook 2
Client 0 received: Order 0 is prepared by cook 5
Client 3 received: Delivery person 0 delivering order 4 over a distance of 13.04 km
Client 1 received: Order 1 is prepared by cook 3
Client 2 received: Order 2 is prepared by cook 4
^C
Caught signal 2 (SIGINT), cleaning up... Client generator program will be closed. You can st
t it again!
Client 3 received: Your order is delivered.

Client 4 received: Delivery person 0 delivering order 3 over a distance of 9.85 km
Client 0 received: Delivery person 0 delivering order 0 over a distance of 26.42 km

==129380== HEAP SUMMARY:
==129380==    in use at exit: 0 bytes in 0 blocks
==129380==    total heap usage: 13 allocs, 13 frees, 4,414 bytes allocated
==129380== All heap blocks were freed -- no leaks are possible
==129380== For lists of detected and suppressed errors, rerun with: -s
==129380== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
emre@ubuntu:~/Downloads/cse344-system-programming/Final$
```

Fig-11: Valgrind Result - 4