CSE344
Final Report

Eren Çakar
1901042656

June 15, 2024

Contents

1 Introduction

1.1 Project Overview

This project involves simulating a food production and delivery system for a Pide Shop. The system includes three main components: the pide house, delivery personnel, and customers.

1.2 Objectives

The objectives of this project are to design and implement a multi-threaded server and client system that simulates the operation of a pide shop, including order handling, preparation, cooking, and delivery.

2 System Design

2.1 Architecture

The system consists of two separate programs which represent the shop and the group of customers which behave as a server and a client that communicate on local machine respectively.

2.1.1 Program Executions

The client side program is executed with arguments

- **port:** The port to use to send connection request to the server.
- **number of customers:** Number of unique clients to give order for.
- **p:** The first position value of the client on a two dimensional plane.
- **q:** The second position value of the client on a two dimensional plane.

The server side program is executed with arguments

- **port:** The port to listen to when waiting for clients.
- **cookPoolSize:** Essentielly number of cooks in the shop.
- **deliveryPoolSize:** Essentially number of couriers of the shop.
- **deliverySpeed:** The constant which affects the time a courier takes to traverse some distance, greater the value given for this argument shorter the waits for courier deliveries.

2.1.2 Client

The client program works following the steps explained below

Signal Handling Setup

1. The program sets up a signal handler for `SIGINT` using `sigaction`.
2. The signal handler function `sigintHandler` checks if any orders have been sent. If not, it prints a message and exits the program.
3. If orders have been sent, it sets a flag `ordersCancelled` to 1 and prints "Orders cancelled".

Command Line Argument Parsing

1. The program checks if the correct number of command line arguments are provided. If not, it prints a usage message and exits.
2. It parses the command line arguments to extract the port number, number of customers, and dimensions `p` and `q`.
3. It validates the port number to ensure it is between 1024 and 65535.
4. It validates the number of customers to ensure it is between 1 and 1024.
5. It validates `p` and `q` to ensure they are between 1 and 1000.

Socket Setup and Connection

1. The program creates a socket using `socket`.
2. It initializes the server address structure with the IP address "127.0.0.1" and the port number.
3. It attempts to connect to the server using `connect`. If the connection fails, it prints an error message and exits.

Order Preparation and Sending

1. The program sets the `sentOrders` flag to 1 to indicate that orders are being sent.
2. It initializes a buffer to hold the order data for all customers.
3. For each customer, it generates a random position within the town dimensions `p` and `q`, and stores this information in the buffer.
4. It prints the order information for each customer.
5. If orders have not been cancelled, it sends the buffer to the server using `write`.

Server Response Handling

1. The program initializes a flag `connectionSafelyClosed` to 0.
2. It enters a loop to read responses from the server using `read`.
3. For each response, it prints the server's response character by character.

4. If the response is "Thank you for your order", it sets `connectionSafelyClosed` to 1 and breaks the loop.

5. If the `read` operation fails or the connection is not safely closed, it prints an error message and exits.

Order Cancellation Handling

1. If orders have been cancelled, the program sends a "CANCEL" message to the server using `write`.

Socket Closure

1. The program closes the connection socket using `close`.

Program Termination

1. The program exits with a status code of 0 indicating successful execution.

2.1.3 Server

The client program works following the steps explained below

Command Line Argument Parsing

1. The program checks if the correct number of command line arguments are provided. If not, it prints a usage message and exits.

2. It parses the command line arguments to extract the port number, cook pool size, delivery pool size, and delivery speed.

3. It validates the port number to ensure it is between 1024 and 65535.

4. It validates the cook pool size to ensure it is between 1 and 1024.

5. It validates the delivery pool size to ensure it is between 1 and 1024.

6. It validates the delivery speed to ensure it is between 1 and 1024.

Signal Handling Setup

1. The program sets up `SIGINT` signal handling.

Log File Setup

1. The program opens a log file named "server.log" for writing, creating it if it does not exist, and appending to it if it does.

2. If opening the log file fails, it prints an error message and exits.

Socket Setup and Binding

1. The program creates a socket using `socket`.

2. It sets socket options to reuse the address and port using `setsockopt`.

3. It initializes the server address structure with `INADDR_ANY` and the port number.

4. It binds the socket to the address and port using `bind`. If binding fails, it prints an error message and exits.

5. It starts listening for incoming connections using `listen`. If listening fails, it prints an error message and exits.

Server Operation

1. The program prints and logs that the server is listening on the specified port.

2. It enters a loop to accept incoming client connections.

3. For each client connection:

   (a) It prints and logs the acceptance of the connection along with the client's address and port.

   (b) It initializes various counters and semaphores for order management.

   (c) It creates and initializes mutexes for synchronization.

   (d) It initializes circular queues for managing orders.

   (e) It creates the manager thread using `pthread_create`.

   (f) It creates the cook threads based on the cook pool size.

   (g) It creates the courier threads based on the delivery pool size.

   (h) It waits for all threads to finish using `pthread_join`.

   (i) It prints and logs that all threads are done.

   (j) It prints and logs statistics about the orders and performance of cooks and couriers.

   (k) It frees dynamically allocated memory for the threads and their arguments.

   (l) It destroys the semaphores and mutexes.

   (m) It clears the circular queues.

   (n) It prints and logs the completion of serving the client.

   (o) It sends a "Thank you for your order" message to the client.

   (p) It closes the client socket.

Server Shutdown

1. The program closes the server socket.

2. It returns 0 indicating successful execution.

2.1.4 Actors: Manager

After client program is connected to the server program the manager takes from there. The manager's life cycle is that

1. Pick up the phone.

2. Note down each of the orders.

3. Put each of the order note to the orders list for cooks to take from.

4. Until every order is delivered

   (a) In the case of a change in state in any order, print the number of orders in each state.

   (b) In the case of a new order is cooked and ready to be delivered, take it from the cooks' counter to the couriers' counter.

   (c) In the case of phone is ringing, pick it up and if the caller says to cancel the orders, cancel the remaining orders.

2.1.5 Actors: Cook

Cooks are the main pillar of the shop, and therefore have a complex life cycle throughout the order cooking cycle.

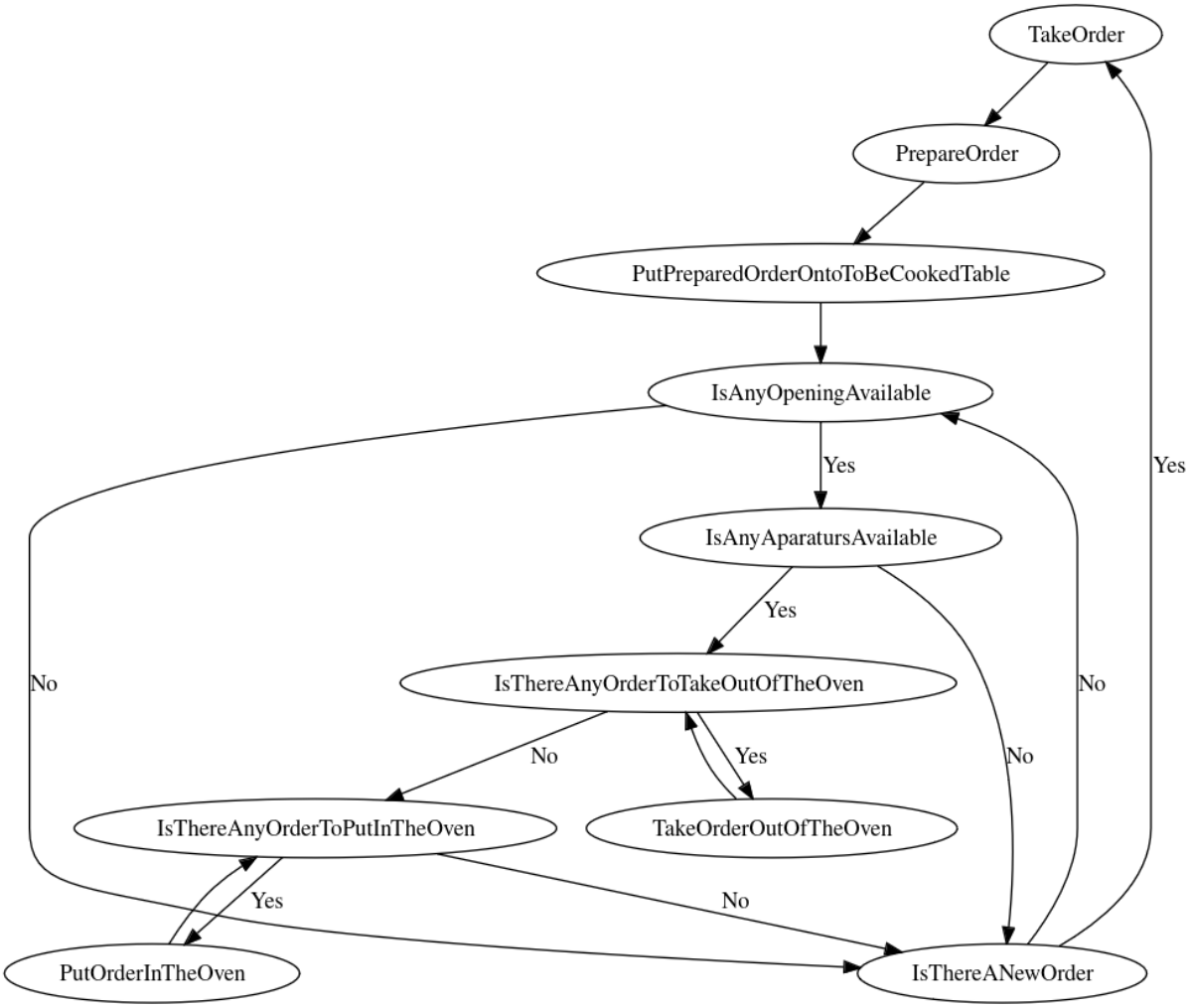

Figure 1: The life cycle of a cook

In Figure 1 the complex life cycle of a cook is shown as a flowchart. To explain its inner workings in detail:

The `cook` function simulates the behavior of a cook in a pide shop. The function is designed to be run as a separate thread, and it takes a single argument, a void pointer `arg`, which is expected to be a pointer to an integer representing the ID of the cook.

The function begins by initializing two circular queues, `ordersToPutInOven` and `ordersToTakeOutFromOven`, and a counter `ordersCount` to keep track of the number of orders processed by this cook. It then enters a loop that continues until the number of cooked orders is less than the total orders.

In each iteration of the loop, the function dequeues an order from the global `orders` queue. If the order is not `NULL`, it decreases the number of orders to be prepared, increases the number of orders in preparation, and logs that the cook is preparing the order. It then simulates the time it takes to prepare the order using the `pseudoInverse` function, decreases the number of orders in preparation, and logs that the order has been prepared. The order is then enqueued in the `ordersToPutInOven` queue, and the number of orders waiting for the oven is increased.

Next, the function checks if the oven and the oven apparatus are available using semaphores. If either is not available, it releases the oven doors semaphore if it was acquired and continues to the next iteration of the loop.

The function then checks the `ordersToTakeOutFromOven` queue for orders that are ready to be taken out of the oven. If an order's time is less than or equal to the current time, it increases the number of free ovens, dequeues the order from the `ordersToTakeOutFromOven` queue, enqueues it in the `readyOrders` queue, logs that the order has been put in the delivery queue, increases the number of cooked orders, and frees the memory allocated for the order.

The function then checks the `ordersToPutInOven` queue for orders to put in the oven. If an order is found and there is a free oven, it decreases the number of free ovens, decreases the number of orders waiting for the oven, updates the order's time, enqueues it in the `ordersToTakeOutFromOven` queue, and logs that the order has been put in the oven.

Finally, the function releases the oven apparatus and oven doors semaphores, and if all orders have been cooked, it logs that the cook is done and updates the integer pointed to by `arg` with the number of orders processed by this cook.

2.1.6 Actors: Courier

The life cycle of a courier is far more simple. It is basically:

1. Wait until you have 3 orders in your bag to deliver or there is no orders that are prepared.
2. Start delivering by the oldest order and deliver orders from oldest to most recent.
3. Return to the shop when there is no orders to deliver any more.

To explain the thread function in detail:

The thread function `courier` simulates the behavior of a courier in a pizza delivery system. The function is designed to be run as a separate thread, and it takes a single argument, a void pointer `arg`, which is expected to be a pointer to an integer representing the ID of the courier.

The function begins by initializing a circular queue `myOrders` to keep track of the orders assigned to this courier, and variables `myP` and `myQ` to keep track of the courier's

current position. It also initializes a counter `ordersCount` to keep track of the number of orders delivered by this courier. It then enters a loop that continues until the number of delivered orders is less than the total orders.

In each iteration of the loop, the function first checks if the size of `myOrders` is equal to `DELIVERY_ORDER_COUNT` or if the remaining orders to be delivered are less than `DELIVERY_ORDER_COUNT`. If either condition is true, it enters a nested loop that continues until `myOrders` is empty. In each iteration of the nested loop, the function dequeues an order from `myOrders`, logs that the courier is delivering the order, simulates the time it takes to deliver the order using the `sleep` function, logs that the order has been delivered, updates `myP` and `myQ` with the order's position, frees the memory allocated for the order, increases `ordersCount`, decreases the number of orders in delivery, and increases the number of delivered orders. After the nested loop, if the courier is not at the shop, it logs that the courier is returning to the shop, simulates the time it takes to return to the shop, logs that the courier has returned to the shop, and resets `myP` and `myQ` to 0.

Next, the function tries to acquire the `semReadyOrders` semaphore. If it fails to acquire the semaphore because it is already locked, it continues to the next iteration of the loop. Otherwise, it dequeues an order from the `readyOrders` queue. If the order is not NULL, it decreases the number of orders waiting for delivery, enqueues the order in `myOrders`, increases the number of orders in delivery, and logs that the courier is taking the order.

Finally, after the loop, the function logs that the courier is done. If the courier is not at the shop, it logs that the courier is returning to the shop, simulates the time it takes to return to the shop, logs that the courier has returned to the shop, and resets `myP` and `myQ` to 0. The function then updates the integer pointed to by `arg` with

2.2 Components

2.2.1 Pide Shop

Pide shop is assumed to be at the position (0, 0), although the assignment PDF demanded it to be at the center of the map, it also did not give examples giving the size of the map to the server program.

Even then since position of the pide shop is greatly arbitrary and not too important on the exact values, this decision was made.

With pide shop being at (0, 0) and customers having random values of positions between [0, x] and [0, y], the system is actually equivalent to having customer position values between [-x, x] and [-y, y] with pide shop being at the center.

2.2.2 Courier

The couriers constantly move between customers and shop as long as they have enough order to deliver.

The amount of time it takes for a courier to deliver an order to a customer depends on where was the courier and what was the value given as the `deliverySpeed`.

It is always assumed courier takes a straight path to the next customer. This may not be the most realistic approach but since the main goal is to have a delivery time relative to the distance, this achieves it.

The formula used to calculate `sleep` mount is

$$t = ((p_{\text{customer}} - p_{\text{courier}})^2 + (q_{\text{customer}} - q_{\text{courier}})^2)/(1000 \cdot deliverySpeed)$$

2.2.3 Customers

Customers are randomly positioned on a map and do not have personalized orders, they all order same type of pide.

They do not show individualistic behaviour, and cancel altogether, at least the ones that are yet to receive their order, or not cancel at all.

2.2.4 Cook

Every cook has his own infinite length counters to put prepared orders that are to be put into oven.

They first look at the oven from afar to see if openings are available then check the paddles (apparatus) that are next to the oven if there is any available.

Our cooks are very talented and actions of putting pides in and out of oven is nothing for them therefore they do these instantaneously. Because of this whenever any of the checks mentioned above fail to satisfy availability they just return to prepare next order as they know whoever is occupying the openings won't be taking their time too long and they can check later.

3 Race Conditions Handling

The system has parts that are susceptible to race conditions in ares where cooks need to share same resources, when manager needs to hand out the cooked orders to be delivered, and lastly tracking the statistics.

Race condition in stats tracking is fairly straightforward and is only about having functions to decrease and increase them with locking mutex variables while doing that.

The race condition for manager handing out orders to couriers is solved by having a semaphore variable where the manager posts whenever an order can be picked up to deliver, and is waited whenever a courier does pick up an order.

For the race conditions apparent for cooks, similar methods are used. Since there are two openings and three paddles, both of them are represented as semaphores. Cooks do not like waiting idly, so they `sem_trywait` instead of `sem_wait`. If waiting fails they just return to preparing next order. Cooks also wait each other whenever the number of orders in the oven is changing, either by putting in an order or taking out one, this is achieved by a mutex variable.

4 Signal Handling

The server program prints that user interrupted the program and then calls clean up function.

The `cleanup` function is designed to clean up resources when the server is shutting down. It starts by setting `totalOrders` to -1, which signals to cooks and couriers that the shop is no longer in pursuit of cooking or delivering orders.

The function then logs the start of the cleanup process to both the console and a log file. The `dprintf` function is used to write to the log file, which is identified by the file descriptor `logFd`. The `getTimestamp` function is used to get the current time in a format suitable for logging.

Next, the function checks if the server is connected. If it's not, it logs that the cleanup is complete and returns immediately. This is because that the rest of the cleanup process is only necessary if the server was previously connected.

If the server is connected, the function proceeds to cancel all active threads. It cancels the `managerThread` and then loops over the `cookThreads` and `courierThreads` arrays to cancel each cook and courier thread. The sizes of these arrays are given by `cookPoolSize` and `deliveryPoolSize`, respectively.

After cancelling the threads, the function joins them. This means it waits for each thread to finish executing before continuing. This is important to ensure that all resources used by the threads are released.

The function then frees the memory allocated for various data structures. These include `order`, `cookThreads`, `orderWithTime`, `courierThreads`, `cookThreadsArgs`, and `courierThreadsArgs`. This is necessary to prevent memory leaks.

Next, the function destroys all semaphores and mutexes. Semaphores are used to control access to shared resources, while mutexes are used to prevent race conditions by ensuring that only one thread can access a particular section of code at a time.

The function then clears all circular queues. These queues are presumably used to manage orders in various stages of processing.

Next, the function closes the server socket. This is necessary to free up the port that the server was listening on.

Finally, the function logs that the cleanup is complete and closes the log file. This is the last step in the cleanup process.

And for the client side user interrupt handling, refer to the 2.1.2

5 Testing and Results

```
Number of delivered orders: 48
Order for customer 31 to position (423, 90) is delivered
Courier 0 is returning to the shop
Number of orders waiting to be prepared: 0
Number of orders in preparation: 0
Number of orders waiting for oven: 0
Number of orders in oven: 0
Number of orders waiting couriers: 0
Number of orders on couriers: 1
Number of delivered orders: 49
Order for customer 37 to position (498, 8) is delivered
Courier 2 is returning to the shop
Courier 4 is done
Courier 3 returned to the shop
Courier 3 is done
All orders are cooked and delivered
Courier 1 returned to the shop
Courier 1 is done
Courier 0 returned to the shop
Courier 0 is done
Courier 2 returned to the shop
Courier 2 is done

All threads are done
Stats:
Total orders: 50
Cook 0 prepared 17 orders
Cook 1 prepared 16 orders
Cook 2 prepared 17 orders
Courier 0 delivered 9 orders
Courier 1 delivered 9 orders
Courier 2 delivered 13 orders
Courier 3 delivered 9 orders
Courier 4 delivered 10 orders
Done serving to 127.0.0.1:38482

Waiting new orders...
```

```
Put order from customer 15 at position (314, 383) in queue
Put order from customer 16 at position (96, 31) in queue
Put order from customer 17 at position (127, 484) in queue
Put order from customer 18 at position (431, 496) in queue
Put order from customer 19 at position (356, 49) in queue
Put order from customer 20 at position (273, 375) in queue
Put order from customer 21 at position (268, 474) in queue
Put order from customer 22 at position (147, 135) in queue
Put order from customer 23 at position (145, 254) in queue
Put order from customer 24 at position (345, 321) in queue
Put order from customer 25 at position (103, 322) in queue
Put order from customer 26 at position (331, 160) in queue
Put order from customer 27 at position (463, 428) in queue
Put order from customer 28 at position (166, 136) in queue
Put order from customer 29 at position (224, 188) in queue
Put order from customer 30 at position (141, 390) in queue
Put order from customer 31 at position (423, 90) in queue
Put order from customer 32 at position (421, 50) in queue
Put order from customer 33 at position (426, 353) in queue
Put order from customer 34 at position (46, 282) in queue
Put order from customer 35 at position (254, 172) in queue
Put order from customer 36 at position (9, 22) in queue
Put order from customer 37 at position (498, 8) in queue
Put order from customer 38 at position (158, 144) in queue
Put order from customer 39 at position (114, 355) in queue
Put order from customer 40 at position (465, 217) in queue
Put order from customer 41 at position (29, 148) in queue
Put order from customer 42 at position (378, 493) in queue
Put order from customer 43 at position (76, 396) in queue
Put order from customer 44 at position (129, 301) in queue
Put order from customer 45 at position (84, 122) in queue
Put order from customer 46 at position (43, 359) in queue
Put order from customer 47 at position (64, 465) in queue
Put order from customer 48 at position (410, 343) in queue
Put order from customer 49 at position (318, 308) in queue
Thank you for your order
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/final$
```

Figure 2: Result of 50 orders with 3 cooks and 5 couriers

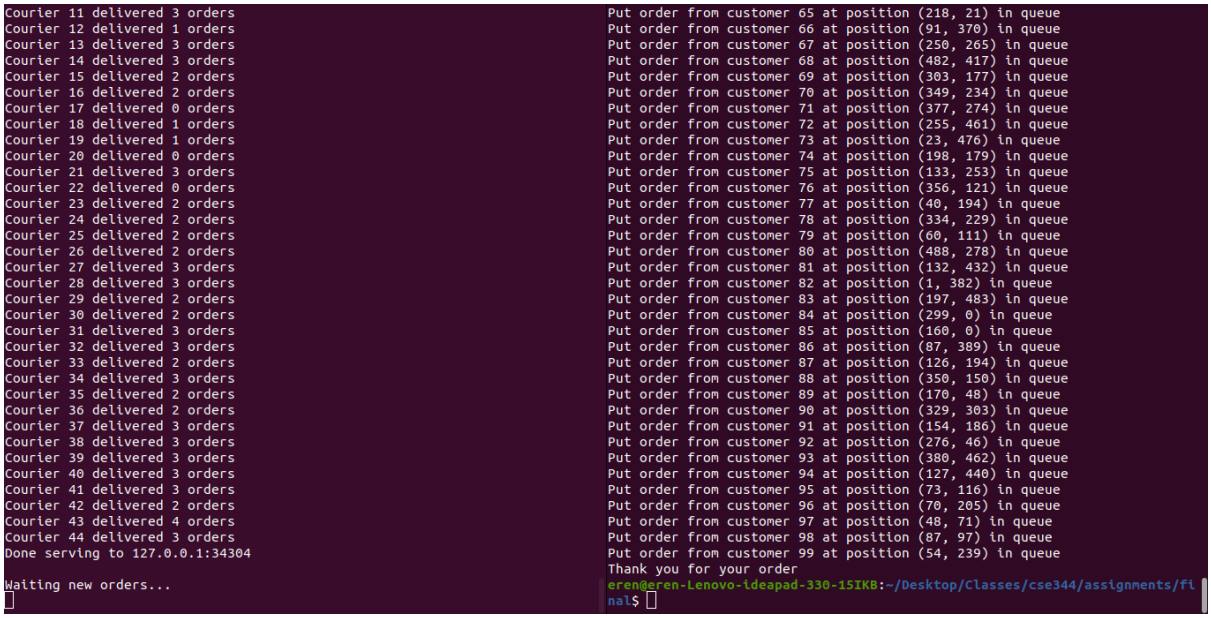

Figure 3: Result of 100 orders with 30 cooks and 45 couriers (1)

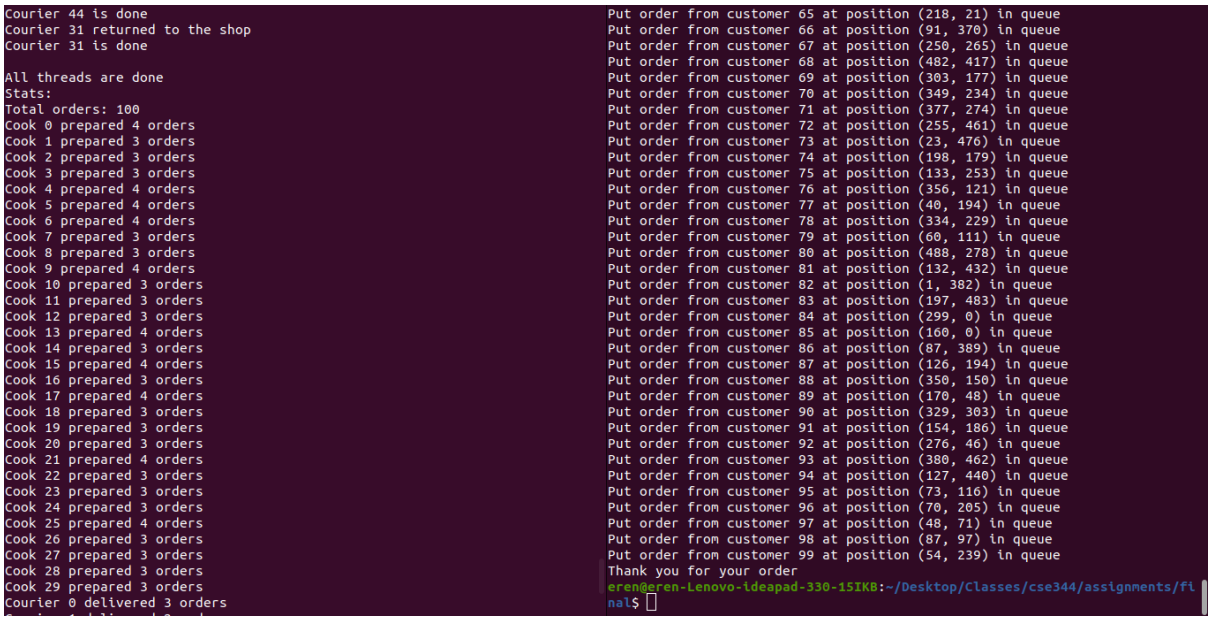

Figure 4: Result of 100 orders with 30 cooks and 45 couriers (2)

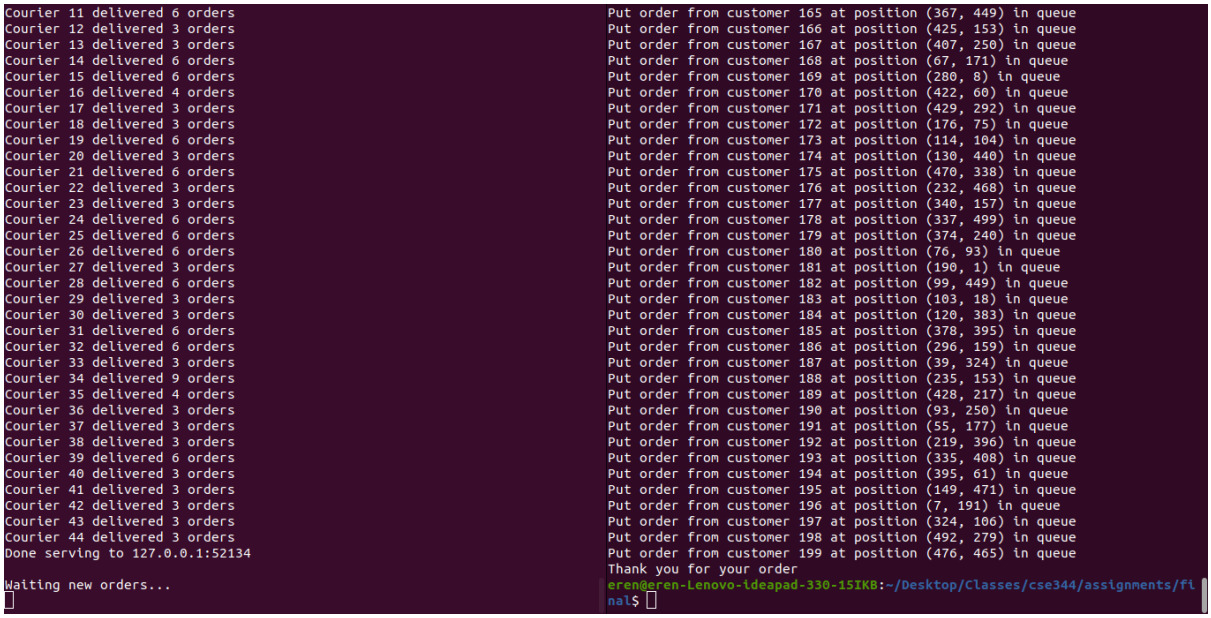

Figure 5: Result of 200 orders with 30 cooks and 45 couriers (1)

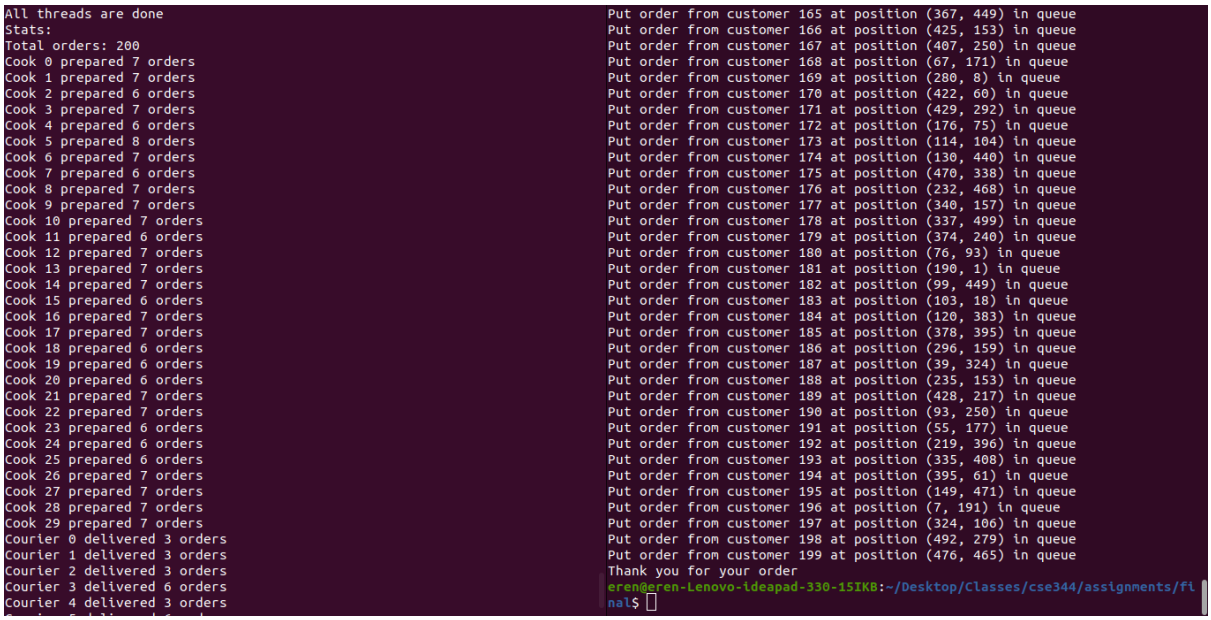

Figure 6: Result of 200 orders with 30 cooks and 45 couriers (2)

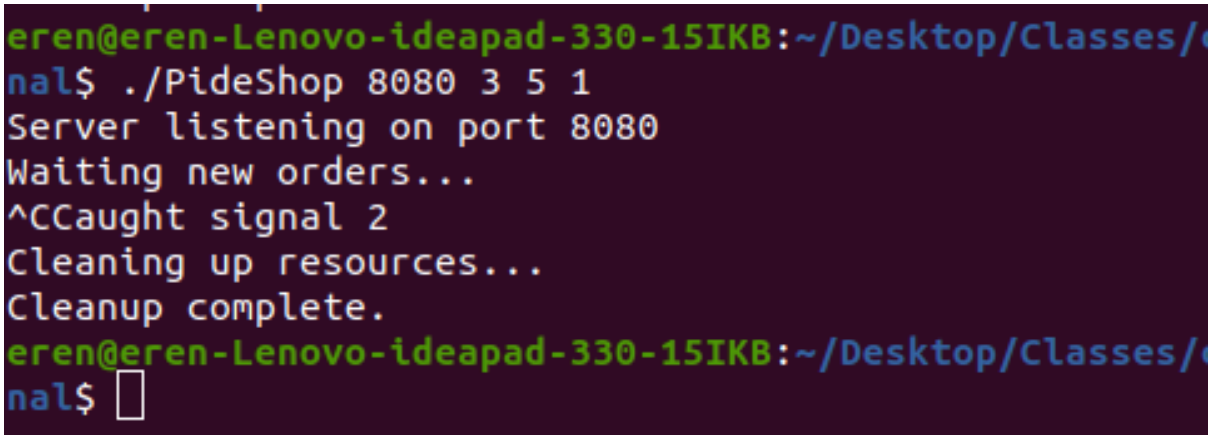

Figure 7: Cancellation on server while not connected

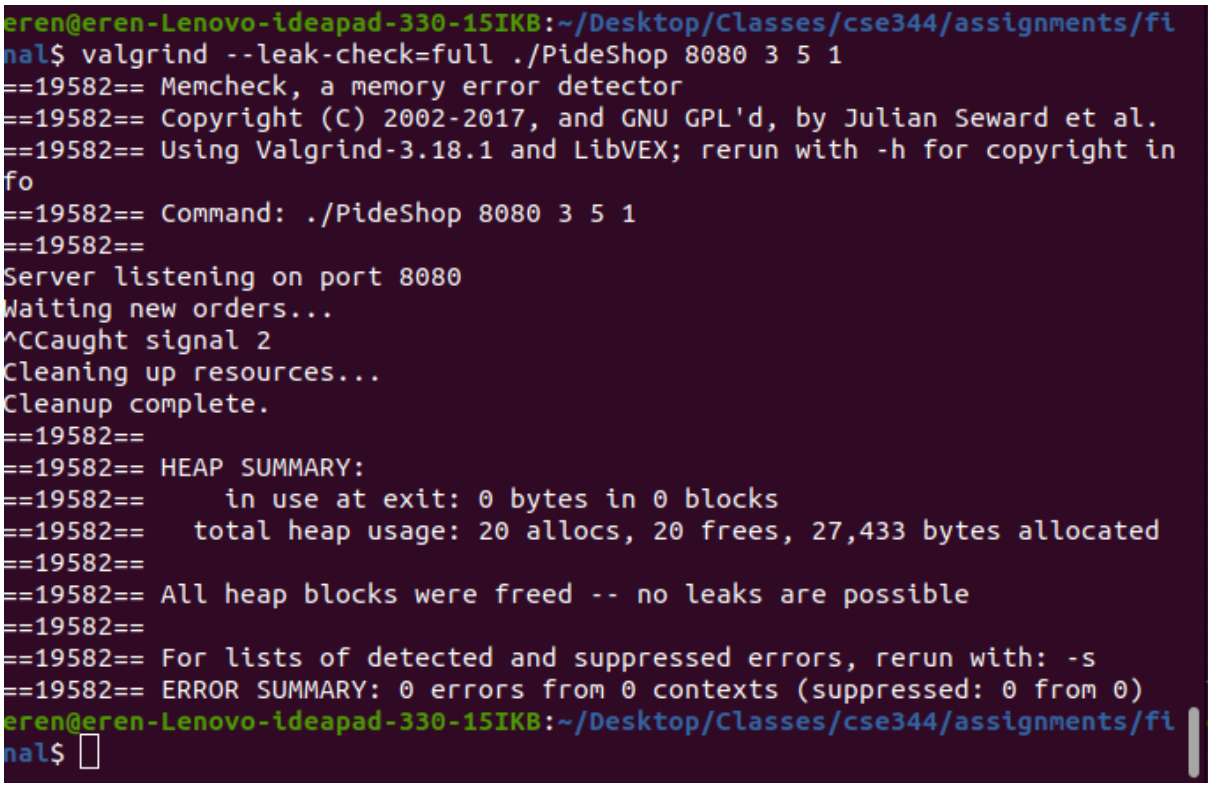

Figure 8: Cancellation on server while not connected with valgrind

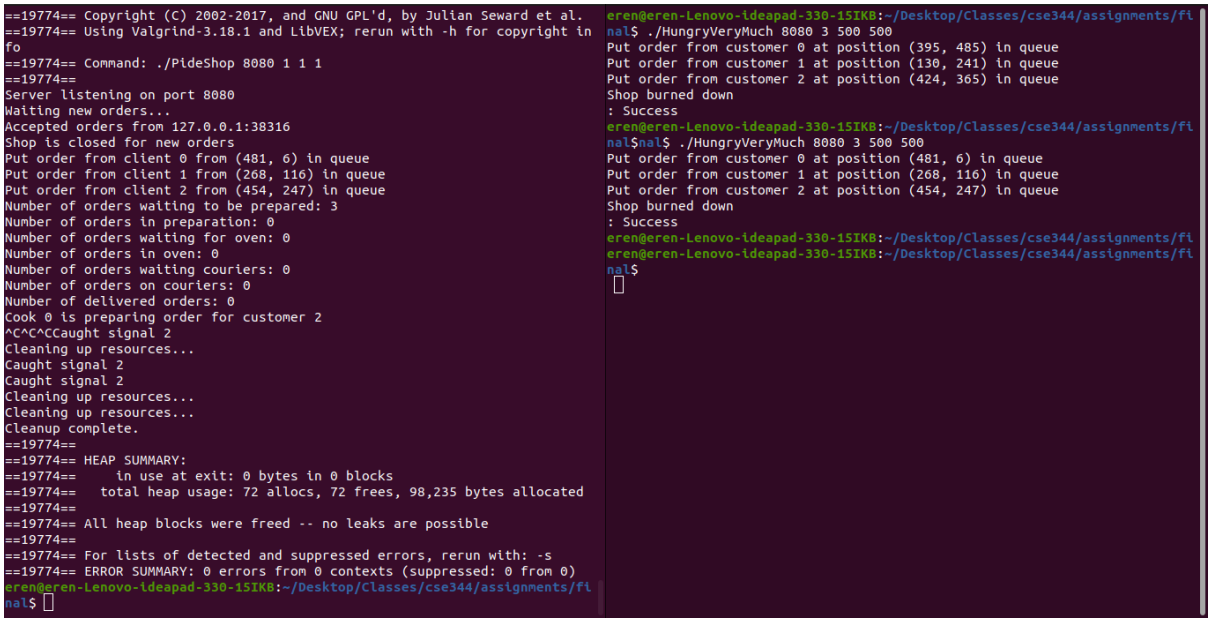

Figure 9: Cancellation on server while connected with valgrind

Note: Probably due to extensive leak checks valgrind slows down the program extremely, because of this the test in Figure 9 is done with small values of cooks, couriers, and customers.

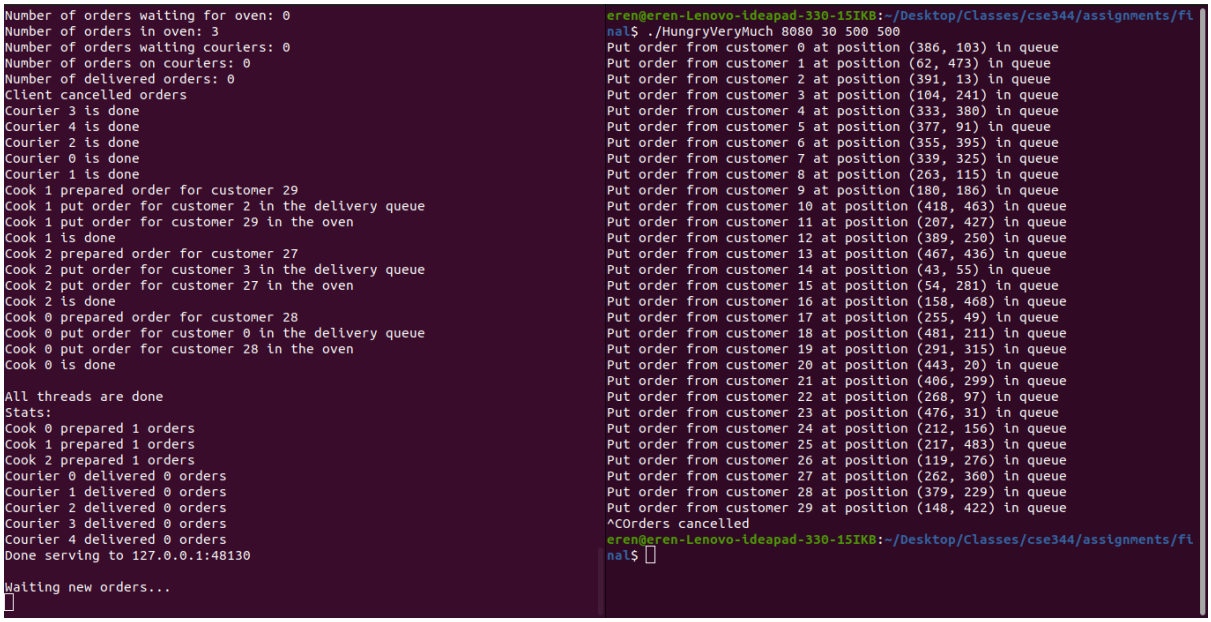

Figure 10: Cancellation on client while connected

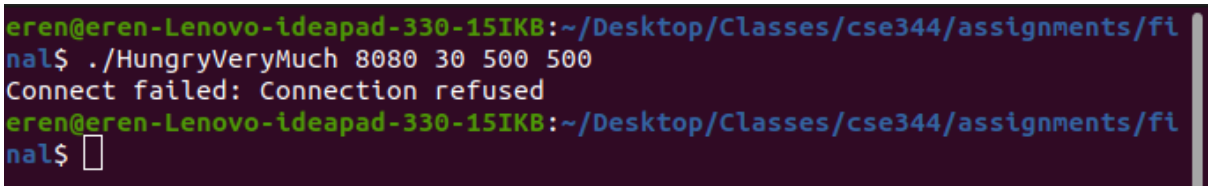

Figure 11: Connection failure on client whilst server is down