Mini Project - OnlineStore

Anuj Arora (anuj.arora@iiitb.ac.in)
Github : https://github.com/axnuzzz/OnlineStore

1 Overview

This project is a command-line application that emulates an OnlineStore. It follows a client-server architecture, where client requests are transmitted via sockets to the server program. The server processes these requests and returns the required results. Users have different privileges based on their role. Regular users, referred to as customers, can view their shopping carts, while Admin users have additional privileges to modify store items.

The communication between the client and server is achieved through Socket programming. Both the client and server sides interact using system calls, while the server side utilizes file locking mechanisms to interact with the data files.

2 Application Architecture

The application has the following files:

- Client.c - this file contains the code to be run at the client side.
- Server.c - this file contains the code to be run at the server side.
- customers.txt - this file contains the customers registered with the store.
- orders.txt - this file contains the carts for each of the users.
- records.txt - this file contains the inventory of the store, i.e. the products currently in the store with their quantities and prices.
- receipt.txt - this file contains the receipt for the user after payment.
- adminReceipt.txt - this file is used to store the history of th e updates, deletes and additions of the products being dine by the admin.
- headers.h - this file contains the structs and macros to be used in the program.

3 How to Run the Code?

- Open 2 terminal windows.
- In the first terminal window, run the commands

```
$ gcc -o server Server.c
$ ./server
```

  This starts up the server.

- In the second terminal window, run the commands

```
$ gcc -o client Client.c
$ ./client
```

  This connects the server to the client.

- We can then run the program as is directed by the client program.

4 headers.h

The following structs and macros are used throughout the program to pass data among the server and client.

- **struct product** - the struct to store the product info (product id, product name, quantity and price).
- **struct cart** - the struct to store the cart info for a customer (customer id, the list of products in the cart).
- **MAX PROD** - the maximum products in a cart. This is set to 20.
- **struct index** - the struct which contains the customer id and his cart offset in the orders.txt file (thus customers.txt serves as an ndx file to orders.txt).

5 Functionalities of the user

Normal User (Customer) The program intends to provide the following functionalities to custo- mers (normal users):

- To register himself as a new customer
- To view the items available at the store with their respective quantities in stock and the prices, so that the customer can know what the store offers
- To add product to his cart, along with the quantity of the product to be ordered
- To edit the quantities of existing products in his cart, this also allows him to delete items from his cart.
- To proceed for payment for all the products in his cart. Following successful payment, a receipt is generated for the user to view his order.

Admin User The admin user serves as the administrator of the OnlineStore. He can manage the products in the online store, and has the following functionalities available to him:

- To add a new product to the store
- To update the quantity available or the price of an existing product in his store.
- To explicitly remove products from the store.
- To see his inventory, i.e. all the items currently present in the store.
- Any update leads to the transaction being recorded in the adminReceipt.txt file.

6 Functionalities of the server

Server:

The server serves the requests of the client (whether a customer or the admin) and has functionality to support the above requests.

7 Implementation of Client.c

➢ Helper functions

The code uses the following helper functions for modularisation:

- **displayMenuUser(), displayMenuAdmin()**
  These functions are utilized within while loops to present the available options to the customer or admin. By examining these menus, the user can input their choices. For customers, the options are represented by letters from 'a' to 'g', while for admins, the options range from 'a' to 'f'.

- **printProduct(), getInventory()**
  These functions are utilized to present the user with a well-structured report following the client's receipt of a response from the server.

- **calculateTotal(), generateReceipt()**
  These functions are employed to calculate the total amount in the user's cart and generate a receipt once the payments are made. The `generateReceipt()` function passes the calculated total and the cart to the server, enabling it to generate the receipt. These functions are invoked when the user selects the option to proceed with the payment for the items in their cart.

- **custIdTaker(), productIdTaker(), quantityTaker(), priceTaker()**
  These functions serve the purpose of receiving user input while ensuring that all the provided values are non-negative. It is important to enforce these conditions because using negative values for these quantities can lead to unpredictable outcomes, which are neither desired nor necessary. Therefore, these functions impose constraints on the inputs to maintain their non-negativity.

➢ Socket Setup

The main function first consists of socket setup, code for which is as follows: in t

```
main ()
{
    printf(" Establishing connection to server n" ); \
    int sockfd = socket(AF INET, SOCK STREAM, 0 );

    if ( sockfd == −1)
    {
        perror (" Error : " ) ; return −1;
    }

    struct sockaddr in serv ;

    serv . sin family = AF INET ;
    serv . sin addr . s addr = INADDR ANY; serv .
    sin port = htons ( 5555 ) ;

    if ( connect ( sockfd , ( struct sockaddr *)& serv , sizeof ( serv )) == −1) perror (" {
        Error : " ) ;
        return −1;
    }
```

➢ Implementation of Customer functions

Next, the program prompts the user to input the type of user they are, with "1" representing a customer and "2" representing an admin. It is important to note that within a single run of the program, the user cannot switch between being a customer and an admin; they must choose one role.

If the user selects the customer option, the customer menu is displayed, and the program proceeds to execute the customer functions. The choice made by the user is sent to the server to inform it of the selected role.

The customer functions mentioned earlier are as described previously, and their specific implementations are provided below. It is worth noting that all user inputs obtained throughout the program are collected using the user input functions defined earlier.

- To exit the menu, we simply break the loop.
- To display products, we use the getInventory() function to get a response from the server, and display it.
- To get the cart of a customer, we take the customer id as input, and return the cart given by the server, or return a message if the customer id is wrong.
- To add a product, we take customer id, product id and quantity as inputs. In case of errors, we

communicate the error to the user, else the cart gets updated at the server side.

- To edit a product, the procedure is the same.
- To pay for the cart, the cart is first displayed to the user, following which the total is calculated and displayed to the user. The user is then asked to enter the amount to pay. If correct amount is entered, we generate the receipt, else we ask the user to enter it again.
- Finally to add himself as a new customer, the customer just enters a confirmation, and the auto generated customer id is given to the user.

➢ Implementation of Admin Functions

If the user selects the admin option, the admin menu is shown, and the program proceeds to execute the admin functions. The admin functions correspond to those described earlier, and their specific implementations are provided below. As mentioned previously, all user inputs are obtained using the helper functions previously defined.

- To add a product, the program prompts the user to enter the product ID, product name, quantity in stock, and price. These inputs are then sent to the server for processing. The server handles the request by either adding the product to the inventory or reporting an error if any issues occur during the process.
- To delete a product from the inventory, the program prompts the user to enter the product ID. If the provided product ID matches an existing product, it is deleted from the inventory. Otherwise, an error message is displayed to indicate that the product ID is incorrect.It's important to note that when an admin deletes a product, the changes are not immediately reflected in the customer's cart. The deleted products will still be visible to the customer until they proceed to the payment page. Once the customer reaches the payment page, the updated values will be displayed, and any products with a price and quantity of zero will indicate that they are no longer available.
- To edit a product's quantity or price, the program prompts the user to provide the product ID of the product they wish to modify, as well as the new quantity or price. If no errors occur during the process, the program updates the product with the new quantity or price, and a corresponding message confirming the update is displayed. However, if any errors are encountered, an appropriate error message is returned.It's important to note that when an admin updates the prices or quantities of products, the changes are not immediately reflected in the customer's cart. The old values will still be visible to the customer when viewing their cart. However, once the customer proceeds to the payment page, the updated values will be displayed, providing an accurate representation of the product's current price and quantity.
- To display the products, the program utilizes the `getInventory()` function to request information from the server. The server responds with the inventory data, and the program then proceeds to display the received information to the user.
- To exit the menu, the program utilizes a loop control mechanism and breaks out of the loop. This allows the program to exit the current menu and proceed with the subsequent actions or terminate the program altogether.

8 Implementation of Client.c

➢ Helper functions

The code uses the following helper functions for modularisation:

- **setLockCust(), unlock(), productReadLock(), productWriteLock(), CartOffsetLock()**
  These functions are responsible for file locking operations. They are invoked whenever there is a need to access the data files. Their purpose is to lock the specific sections of the file that need to be accessed, ensuring exclusive access during that time. This prevents concurrent access or modification by multiple processes or threads, which could lead to data inconsistency or conflicts. The functions are designed to acquire the necessary locks on the file portions required for the specific operation being performed. This ensures that only one process or thread can access that particular portion of the file at any given time.
  - **setLockCust()** : This sets a mandatory read lock on the entire customers.txt. This is used when we want to see the current registered customers with the store.
  - **productReadLock()** : This sets a mandatory read lock on the entire records.txt file. This is used when we want to check particular products or display them.

- **productWriteLock()**:This sets a write lock on the current product, in order to update it. This is used when we want to update or delete a product.
- **CartOffsetLock()**: This takes the cartOffset as an argument (which we get from the getOffset() function) and locks that particular cart for reading/writing.
- **unlock()**: to unlock any given lock.

From this point onward, we assume that proper file access management is in place. This includes the implementation of appropriate locks whenever file access is required and the corresponding unlocking of files wherever necessary. The assumption is that file locking mechanisms are appropriately implemented to ensure exclusive access to files during critical operations, and locks are released after completing the necessary tasks to allow other processes or threads to access the files.

- **getOffset()**

  This function is internally called by many of the functions in the server code. This code iterates over the The program searches the `customers.txt` file to find the cart offset for a given customer ID. If the customer ID is found in the file, the program returns the corresponding offset value. If the customer ID is not found, the program returns -1. This offset value is used as a reference point whenever there is a need to update the cart of a specific user.

- **addProducts()**

  This function is employed by the admin to add a product to the inventory. It accepts the necessary parameters for the product and performs a check to determine if the product ID already exists in the inventory. If a duplicate product ID is detected, the function displays an appropriate message indicating that the product already exists. However, if the product ID is unique, the function proceeds to add the product to the inventory.

  In addition to adding the product, the function also writes a log statement to the `adminReceipt.txt` file, documenting the action taken by the admin for future reference. This log statement serves as a record of the product addition process.

- **listProducts()**

  This function is designed to display the products from the inventory. It can be utilized by both the admin and the user. The function retrieves the product information from the inventory and presents it in a readable format, allowing the admin or user to view the available products along with their relevant details. This function provides a convenient way to showcase the inventory contents to facilitate decision-making and browsing for both administrators and users.

- **deleteProduct()**

  This function is employed by the admin to delete a product from the inventory. It initially checks whether the provided product ID is valid or not. If the product ID is found in the inventory, the function proceeds to delete the product. However, if the product ID is not valid or does not exist in the inventory, the function prints an appropriate error message to indicate that the deletion cannot be performed.

  Additionally, a log statement documenting the product deletion is written to the `adminReceipt.txt` file. This log statement serves as a record of the action taken by the admin, providing a history of product deletions for future reference.

- **updateProduct()**

  This function is utilized to update the price or quantity of a product in the inventory. The behavior of the function depends on the value of the `ch` argument passed to it. If the value of `ch` is 1, it indicates that the price of the product needs to be updated. In this case, the function performs the necessary update on the price.

  On the other hand, if the value of `ch` is not 1 (indicating a different value), it implies that the quantity of the product should be updated. The function then performs the required update on the quantity.

  If the updated quantity is passed as 0, it signifies that the product needs to be deleted from the inventory. In such a case, the function calls the appropriate delete product functionality.

  Furthermore, a log statement documenting the update or deletion action is written to the `adminReceipt.txt` file. This log statement serves as a record of the specific action performed by the admin, providing a history of price or quantity updates and product deletions for future reference.

- **addCustomer()**

  This function is responsible for adding a new customer to the system. It generates a new customer ID for the customer using an auto-generated approach. The customer ID is determined by calculating the maximum value among the previous customer IDs and incrementing it by 1.

  By finding the maximum of the existing customer IDs and adding 1 to it, a unique customer ID is generated for the new customer being added. This ensures that each customer has a distinct identifier within the system, allowing for efficient management and retrieval of customer-related information.

- **viewCart()**

  This function accepts the customer ID as an argument and utilizes the `getOffset()` function to obtain the cart offset for the specified customer. If the cart offset is -1, it signifies that the customer ID is invalid, and no cart exists for that customer. In such cases, appropriate measures are taken to handle the invalid customer ID scenario.

  However, if a valid cart offset is obtained from the `getOffset()` function, it indicates that the customer ID is valid, and the corresponding cart is retrieved. The retrieved cart is then written or displayed to the client, allowing the customer to view the contents of their cart.

  It's important to note that the specific implementation details for writing or displaying the cart to the client may vary depending on the programming context or application requirements.

- **addProductToCart()**

  This function serves the purpose of adding a product to a customer's cart. It involves reading the customer ID, product ID, and quantity from the client. The function then proceeds to validate several conditions before adding the product to the customer's cart.

  Firstly, the function checks if the customer ID is valid. If the customer ID is found to be invalid, an appropriate error message is printed to indicate the issue. Similarly, if the product ID already exists in the cart, a message is displayed to notify that the product is already present in the customer's cart.

  Furthermore, the function verifies if the requested quantity is available in stock. If the requested quantity exceeds the available stock, an error message is displayed to indicate the unavailability of sufficient quantity.

  Additionally, there is a limit, denoted by MAX PROD, on the number of products that can be added to a cart. The function ensures that this limit is not exceeded before adding the product. If the limit is reached, an error message is printed to notify that the cart is already full.

  Only when all the above conditions are met successfully, the product is added to the customer's cart.

  It's worth mentioning that the specific implementation of error messages, validation checks, and limitations may vary depending on the programming context and the desired behavior of the application.

- **editProductInCart()**

  The function you described allows the user to modify the quantity of an item they have already ordered. It reads the customer ID, product ID, and the new quantity from the client. The function follows similar checks for the customer ID and the new quantity as in the `addProductToCart()` function.

  The function also verifies whether the product ID exists in the customer's cart or not. If the product ID is not found in the cart, an error message is displayed to indicate that the product is not currently in the cart.

  The remaining functionality of the function resembles the `addProductToCart()` function. If all the checks are successful and the conditions are met, the product quantity is updated in the user's cart. However, if any of the checks fail, an error message is displayed to notify the user about the issue.

  It's important to note that the specific implementation details, such as error message formats and the logic for updating the product quantity in the cart, may vary depending on the programming context or application requirements.

- **payment()**

  When a customer modifies the quantity of items in their cart, these changes do not directly affect the original inventory. The modifications made by the customer are only reflected in their personal cart and are not considered as actual purchases until they proceed with the payment.

  When the customer decides to pay for their cart, several steps are taken to ensure the accuracy of the transaction. Firstly, the system verifies whether the originally requested quantity of each item is still available in the inventory. This is done to account for the possibility that other customers may have purchased some of the items in the meantime. The total amount to be paid is calculated based on the current stock availability.

  For the payment process, the customer is presented with the amount to be paid on the screen. They simply need to enter the displayed amount to complete the payment. Once the payment is successfully processed, several actions are taken:

  1. All changes made in the customer's cart are recorded in the inventory, updating the quantities of the purchased items accordingly.
  2. The customer's cart is cleared, as the items have now been purchased.
  3. A receipt is generated and stored in the `receipt.txt` file, reflecting the details of the items purchased by the customer.

  These steps ensure that the inventory remains accurate, the customer's cart is appropriately managed, and a record of the purchase is maintained for future reference.

- **generateAdminReceipt()**

  When the admin exits the program, a specific function is called to signify that the admin has completed all their updates or modifications. This function serves as a final step before exiting and is responsible for capturing the current state of the inventory.

  The inventory, which includes all the changes made by the admin during their session, is then written to the `adminReceipt.txt` file. This action ensures that a comprehensive record of all the updates performed by the admin is preserved for reference and future analysis.

  By storing the updated inventory in the `adminReceipt.txt` file, it becomes possible to track and review the changes made by the admin, providing a valuable audit trail of their activities within the system.

  It's important to note that the specific implementation of this functionality, such as the format of the inventory record and the method of writing it to the file, may vary depending on the programming context and the desired behavior of the application.

➢ Socket Setup

The main function first consists of socket setup, code for which is as follows:

```
int main () {
        printf (" Setting up server \n " );
        int fd = open (" records . txt " , O RDWR | O CREAT, 0777 );
        int fdcart = open (" orders . txt " , O RDWR | O CREAT, 0777 );
        int fd custs = open (" customers . txt " , O RDWR | O CREAT, 0777 );

        int sockfd = socket (AF INET, SOCK STREAM, 0 ); if ( sockf
        d == -1)                    {
              perror (" Error : " ); return
              -1;
          }
        struct sockaddr in serv , cli; serv
        . sinfamily = AF INET ;
        serv . sin addr . s addr = INADDR ANY; serv .
        sin port = htons ( 5555 );
        int opt = 1 ;

        if ( setsockopt ( sockfd , SOL SOCKET, SO REUSEADDR, &opt , sizeof ( opt ) ) ) perror (" {
              Error : " );
              return -1;
        }

        if ( bind ( sockfd , ( struct sockaddr *)& serv , sizeof ( serv )) == -1) perror (" {
              Error : " );
              return -1;
        }

        if ( listen ( sockfd , 5 ) == -1) pe {
              rror (" Error : " );
              return -1;
        }

        int size = sizeof ( cli );
        printf (" Server set up successfully \n " );
```

Following this, any client requests are accepted, and a fork() is called to implement a concurrent server mechanism.

➢ Implementation of Server functions

The program reads the input sent by the clients about the type of user and the choice of the user, and calls the appropriate helper functions for the case.

User functions -

- If the user asks to exit the program, we break the loop.
- If the user asks to see the inventory, we then call the listProducts() function to list the products.
- If the user asks to view his cart, the viewCart function is called().
- If the user asks to add a product to his cart, the addProductToCart() function is called.
- If the user asks to edit a product in his cart, the editProductToCart() function is called.
- If the user asks to pay for his cart, the payment() function is called.
- Finally to add the user as a new customer, call the addCustomer() function.

Admin functions

1. If the admin asks to add a product to the store, the addProducts() function is called.
2. If the admin asks to delete a product from the inventory, the deleteProduct() function is called.
3. If the admin asks to edit the price of a product in the inventory, the updateProduct() function is called, with ch set to 1.
4. If the admin asks to edit the quantity of a product in the inventory, the updateProduct() function is called, with ch set to 2.
5. If the admin asks to see the inventory, we then call the listProducts() function to list the products.
6. If the admin asks to exit the program, we break the loop.

9 OS Concepts used in the Implementation

➢ File Locking

The program uses mandatory as well as record locking in the following places in the program:

- When we try to find the customer id in the customers.txt file, we perform a mandatory read lock on the entire file.
- When we have found the offset of the cart for the customer, we perform record locking at that offset in orders.txt to lock that cart for reading or writing.
- When trying to find the product id in the list of products, we perform a mandatory read lock on all products. This is also used when we display the inventory.
- When we update a particular product, we remove the read lock on the products and acquire a write lock on that product, to change it.

➢ Socket programming

The program uses sockets to communicate between the server and the client.

- The program uses the server side socket system calls (socket, bind, listen, accept) to setup the socket at the server side, and client side socket system calls (socket, connect) at the client side.
- The fork() system call is used to setup a concurrent server.
- All reads and writes (from the socket) use read and write system calls.

➢ File handling

- We use basic functions such as open, read, write to perform the basic read and write operations on the files.

10 Screenshots of working