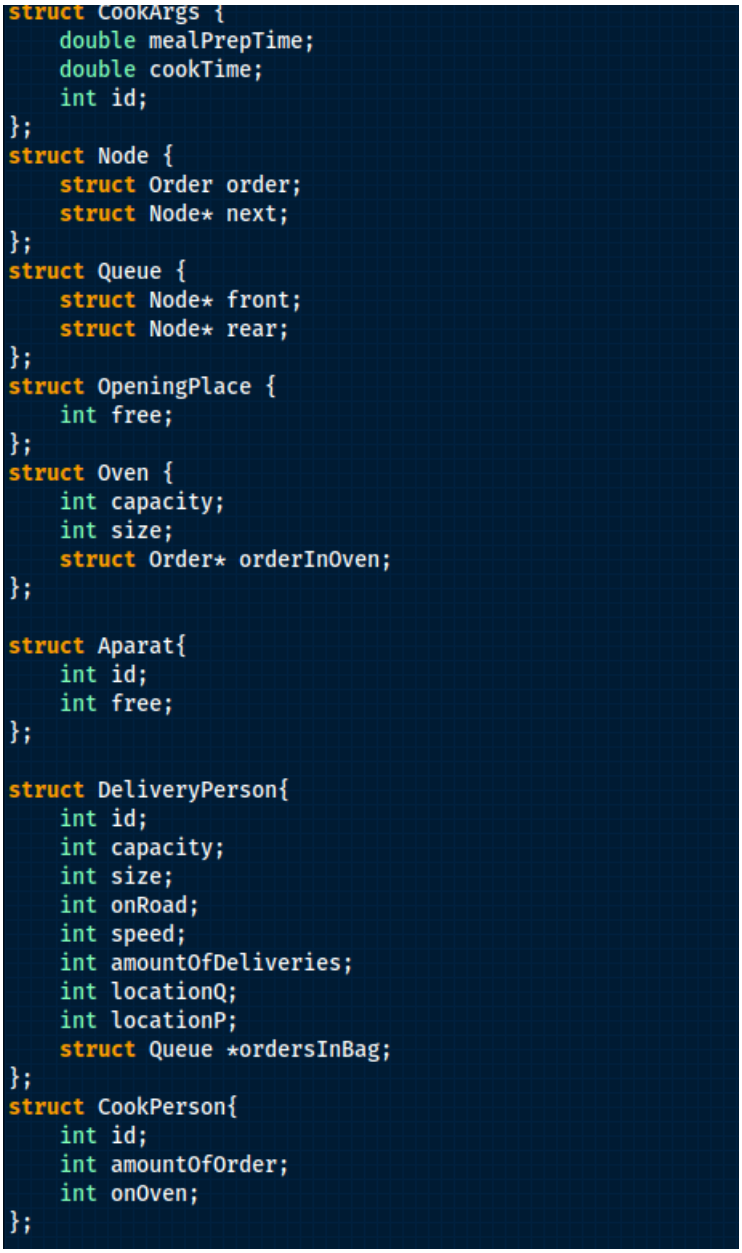
Final report

In the final project, PideShop was requested to be implemented. Connections were established by the server, awaiting a client to connect through a port. Once the client establishes the connection and sends the data, the server begins processing. The client sends all orders in one go through an array. Within the server, this array is structured into multiple queue structures. Queue structures are used for various purposes such as orders for items not yet ready, items ready for pickup, items in the oven, and orders being delivered by the motorcyclist.

To prepare and transport the meals, four different types of threads are used. One represents the chef, another the manager, a third the motorcyclist, and the fourth represents the fire (heat). Here, "fire" refers to the heat inside the stove. When a meal is placed in the oven, each of these threads waits for a certain period to change the variable that indicates the cooking process. Meals ready in the oven are taken by other chefs and placed in the ready queue, allowing the manager to assign orders to available motorcyclists. The variable indicating the need to start delivery is set by the manager. Upon recognizing the need to start delivery, the motorcyclist departs according to the location of the order.

Orders have three states: being prepared, out of the oven, and on the way. These states are communicated to the client. When all notifications are received, the client understands that the process is complete, and the process ends.

Now, let's dive into the code.



```
struct CookArgs {
    double mealPrepTime;
    double cookTime;
    int id;
};
struct Node {
    struct Order order;
    struct Node* next;
};
struct Queue {
    struct Node* front;
    struct Node* rear;
};
struct OpeningPlace {
    int free;
};
struct Oven {
    int capacity;
    int size;
    struct Order* orderInOven;
};

struct Aparat{
    int id;
    int free;
};

struct DeliveryPerson{
    int id;
    int capacity;
    int size;
    int onRoad;
    int speed;
    int amountOfDeliveries;
    int locationQ;
    int locationP;
    struct Queue *ordersInBag;
};
struct CookPerson{
    int id;
    int amountOfOrder;
    int onOven;
};
```

I am using the structs shown above, employing a queue structure and a struct that passes arguments to the thread function. The `DeliveryPerson` struct represents the motorcyclist. The struct holds a value corresponding to the length of the road

```
double calculateMealTime() {

    int M = 30, N = 40;

    lapack_complex_double A[M*N];
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            A[i*N + j] = i + j * I;
        }
    }

    lapack_complex_double U[M*M];
    lapack_complex_double VT[N*N];
    double S[N];
    lapack_complex_double superb[N-1];
    lapack_complex_double work[M*N];
    int lwork = M*N;
    int info;

    clock_t start, end;
    double cpu_time_used;
    start = clock();

    info = LAPACKE_zgesdd(LAPACK_ROW_MAJOR, 'A', M, N, A, N, S, U, M, VT, N);
    if (info > 0) {
        printf("The algorithm computing SVD failed to converge.\n");
        exit(1);
    }

    lapack_complex_double A_pinv[N*M];

    for (int i = 0; i < N*M; i++) {
        A_pinv[i] = 0.0 + 0.0 * I;
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            for (int k = 0; k < M; k++) {
                if (S[k] > 1e-10) {
                    A_pinv[i*M + j] += conj(VT[i*N + k]) * (1.0 / S[k]) * conj(U[j*M + k]);
                }
            }
        }
    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

The function above is used for cooking the meal and preparing it. Delays were implemented where necessary

```
struct Aparat* chosenAparat;
struct OpeningPlace* chosenPlace;
struct CookArgs* cookArgs = (struct CookArgs*) args;
double prepTime = cookArgs->mealPrepTime;
double mealTime = cookArgs->cookTime;
int id = cookArgs->id;
while(!isEmpty(unready) || oven.size != 0){

    if(oven.size < 6 && !isEmpty(unready) ){
        struct Order order;
        pthread_mutex_lock(&mutex_order);
        if(!isEmpty(unready)){
            cooks[id].onOven = 1;
            order = take_order();
            char logMessage[256];
            snprintf(logMessage, 256*sizeof(char), "Cook %d put order %d in oven",id,order.orderId);
            writeToLog(logMessage);
            struct MessageToClient* mesaj= malloc(sizeof(struct MessageToClient));
            mesaj->orderId = order.orderId;
            mesaj->durum = 3;
            if (write(newsockfd, mesaj, sizeof(struct MessageToClient)) < 0) {
                free(mesaj); // Hata durumunda belleği serbest bırak
                error("ERROR writing to socket");
            }
            free(mesaj);
        }
        else{ pthread_mutex_unlock(&mutex_order); break; }
        pthread_mutex_unlock(&mutex_order);
        putToOven(id,order);
    }else if(oven.size == 6 || isEmpty(unready)){
        int mealId = isAnyMealReady();
        takeFromOven(id,mealId);
        if(mealId != -1) {
            char logMessage[256];
            snprintf(logMessage, 256*sizeof(char), "Cook %d take order %d from oven",id,mealId);
            writeToLog(logMessage);

            struct MessageToClient* mesaj= malloc(sizeof(struct MessageToClient));
            mesaj->orderId = mealId;
            mesaj->durum = 3;
            if (write(newsockfd, mesaj, sizeof(struct MessageToClient)) < 0) {
                free(mesaj); // Hata durumunda belleği serbest bırak
                error("ERROR writing to socket");
```

The function above is a chef's function. If the oven is not full, the chef prepares the meal and places it into the queue structure within the oven struct. If the oven is full or there are no unready orders, the chef clears the oven. This way, chefs are utilized to their full capacity

```c
void putToOven(int cookId, struct Order order){

    struct Aparat* chosenAparat;
    struct OpeningPlace* chosenPlace;

    pthread_mutex_lock(&mutex_oven);
    while (oven.size >= oven.capacity) {
        pthread_cond_wait(&cond_oven, &mutex_oven);
    }
    chosenAparat = chooseAparat();
    chosenPlace = choosePlace();
    calculatePrepTime();//sleep(prepTime);                                    // hazırlık yap
    if(oven.size < oven.capacity){
        oven.orderInOven[oven.size] = order;
        oven.size++;
    }else printf("cook %d kapasiteyi aştı HATA \n",cookId);
    pthread_cond_signal(&cond_meal_added);
    pthread_mutex_unlock(&mutex_oven);


    giveAparat(chosenAparat);
    freePlace(chosenPlace);
    cooks[cookId].amountOfOrder++;

}
```

The function above is for placing the meal into the oven. Critical regions are carefully managed to prevent corruption of the oven's queue structure. If the chef receives an order but cannot find empty space, they wait for a signal from the oven.

```c
void takeFromOven(int cookId, int orderId){

    struct Aparat* chosenAparat;
    struct OpeningPlace* chosenPlace;
    struct Order order;
    for(int a = 0 ; a < oven.size ; a++){
        pthread_mutex_lock(&mutex_ready_order);
        if(oven.orderInOven[a].orderId == orderId){

            order = oven.orderInOven[a];
            pthread_mutex_lock(&mutex_oven);
            for(int b = a; b < oven.size - 1 ; b++){
                oven.orderInOven[b] = oven.orderInOven[b+1];
            }

            oven.size--;
            pthread_mutex_unlock(&mutex_oven);

            chosenPlace = choosePlace();
            chosenAparat = chooseAparat();
            enqueue(ready,order);
            numReadyOrders++;
            giveAparat(chosenAparat);
            freePlace(chosenPlace);
            cooks[cookId].onOven = 0;
            pthread_cond_broadcast(&cond_oven);
        }
        pthread_mutex_unlock(&mutex_ready_order); /////// bu mutexe dikkat mutex içinbde mutex var
    }

}
```

The function for removing from the oven follows the same logic as placing into the oven. It waits until it finds an empty OpeningPlace and Aparat, then proceeds with its tasks.

```
void* startPool(void *args){

    while(!ordersFinish()){
            pthread_mutex_lock(&mutex_oven);
            while(oven.size == 0 && !ordersFinish()){
                pthread_cond_wait(&cond_meal_added,&mutex_oven);
            }
            int mealId = findMeal();
            giveFireToMeal(mealId);
            pthread_mutex_unlock(&mutex_oven);
    }
}

int findMeal(){
    int id = -1;
    pthread_mutex_lock(&mutex_oven_find);
    for(int a = 0 ; a < oven.size ; a++){
        if(oven.orderInOven[a].readyToEat == 0)
            id = oven.orderInOven[a].orderId;
    }
    pthread_mutex_unlock(&mutex_oven_find);
    return id;

}
void giveFireToMeal(int mealId){
    calculateMealTime();
    for(int a = 0 ; a < oven.size; a++){
        if(oven.orderInOven[a].orderId == mealId)
            oven.orderInOven[a].readyToEat = 1;

    }
}
```

The function above is responsible for cooking orders inside the stove. The main function for the threads I call "fire threads" is named `startPool`, because these fire threads create a thread pool. It finds an order inside the stove that hasn't been cooked yet, waits for a certain period, and prepares this order to be `readyToEat`.

```
void* functionManager(void* args){

    while( !ordersFinish() || kacEleman(unready)) {
        if(kacEleman(ready) >= 3){
            int id = findFreeMoto();
            giveOrderToMoto(id,3);
        }
        else if(isEmpty(unready) && kacEleman(ready) < 3 && kacEleman(ready) > 0){
            int id = findFreeMoto();
            int orderCount = kacEleman(ready);
            giveOrderToMoto(id,orderCount);
        }
    }
}
```

The Manager function assigns the appropriate number of orders to the suitable motorcyclist. The `giveOrderToMoto` function changes the variable indicating that the motorcyclist should depart, signaling to them that they need to hit the road.

```
void* functionMoto(void* args){
    int* motoId = (int*) (args);
    while( !ordersFinish() ){
        if(motos[*motoId].onRoad){
            carryOrders(*motoId);
            motos[*motoId].onRoad = 0;
        }
    }
}
```

If it finds that it should be on the road based on its own ID, it sets out on the journey.

```
void carryOrders(int id){
    while(!isEmpty(motos[id].ordersInBag )&& motos[id].size != 0){
        pthread_mutex_lock(&mutex_moto);
        struct Order order = dequeue(motos[id].ordersInBag);
        pthread_mutex_unlock(&mutex_moto);
        carryToClient(order,id);
        motos[id].amountOfDeliveries++;
        motos[id].size--;
    }
    backToShop(id);
}
void carryToClient(struct Order order, int id){
    int motoSpeed = motos[id].speed;
    int motoQ = motos[id].locationQ;
    int motoP = motos[id].locationP;
    int orderQ = order.q;
    int orderP = order.p;

    double motoSpeed_km_second = motoSpeed * 1.6 / 100000;

    int distanceKm =  sqrt( ((motoQ - orderQ) * (motoQ - orderQ)) + ((motoP - orderP) * (motoP - orderP))  );
    int carryTime = distanceKm / motoSpeed_km_second;

    struct MessageToClient* mesaj= malloc(sizeof(struct MessageToClient));
    mesaj->orderId = order.orderId;
    mesaj->durum = 3;
    if (write(newsockfd, mesaj, sizeof(struct MessageToClient)) < 0) {
        free(mesaj); // Hata durumunda belleği serbest bırak
        error("ERROR writing to socket");
    }
    free(mesaj);

    sleep(carryTime);                   /// taşıma süresi
    motos[id].locationQ = orderQ;
    motos[id].locationP = orderP;
    pthread_mutex_lock(&mutex_order);
    deliveredOrders++;
    pthread_mutex_unlock(&mutex_order);
    char logMessage[256];
    snprintf(logMessage, 256*sizeof(char), "Moto %d delivered order %d to p → %d , q → %d",id,order.orderId,order.p,order.q);
    writeToLog(logMessage);
```

It delivers the orders considering the distance and speed, ensuring timely delivery, and returns to the starting point (0, 0).

```c
srand(time(NULL));
int pid = getpid();
printf("\n\t<<PID %d\n",pid);
printf("\t<< ... \n");
struct Order *orders = malloc(numOfClients * sizeof(struct Order));
for(int a = 0 ; a < numOfClients; a++){
    orders[a] = generateOrder(p,q);
}

if (write(sockfd, &numOfClients, sizeof(int)) < 0) {
    error("ERROR writing numofclient to socket");
}
if (write(sockfd, orders, numOfClients * sizeof(struct Order)) < 0) {
    error("ERROR writing to socket");
}

if (write(sockfd, &pid, sizeof(int)) < 0) {
    error("ERROR writing to socket");
}

struct MessageToClient* mesaj = malloc(sizeof(struct MessageToClient));
for(int a = 0 ;  a < numOfClients*3; a++){
    if (read(sockfd, mesaj, sizeof(struct MessageToClient)) < 0) {
        error("ERROR reading fron socket");
    }
    char logMessage[100];
    if(mesaj->durum == 1){
        snprintf(logMessage, sizeof(logMessage), "Your order %d is in oven",mesaj->orderId);
        writeToLog(logMessage);
    }
    if(mesaj->durum == 2){
        snprintf(logMessage, sizeof(logMessage), "Your order %d is taken from oven",mesaj->orderId);
        writeToLog(logMessage);
    }
    if(mesaj->durum == 3){
        snprintf(logMessage, sizeof(logMessage), "Moto is carrying your order %d ",mesaj->orderId);
        writeToLog(logMessage);
    }
}
```

The client terminates its operations upon receiving sufficient messages.