FATİH DOĞAÇ
1901042654

CSE344 SYSTEM PROGRAMMING – 2024
FINAL REPORT

1. Introduction

In this project, I implemented a Pide Shop and Customers with socket connection. I used a multithreading approach. Client defines a map ‘p’ by ‘q’ and the pide shop is right in the middle. 0,0 is at left bottom corner.

There are cooks, delivery persons and a manager. When the server receives the orders, manager handles them by first, passing them to cooks to prepare the orders. Then cooks will place the prepared order to oven. When it’s done cooking, cook will pass it back to manager. Then, manager passes it to the delivery person.

2. Mechanism

Socket part:
Socket is created with ipv4 of the server computer. Port and ip will be specified by giving arguments to the server.
Client takes those as arguments as well.

Signal part:
Unfortunatelly, server can’t cancel orders when it receives a interrupt. It frees up everything and then closes.
When client gets interrupted, server can’t know it. So, orders will be prepared and be delivered.

Threads:
There are n cook threads, m delivery person threads, one manager and one oven with 6 places.

Log:
Server saves its logs into server_log.log and client saves its logs into client_log.log

3. Coding

All infos about the order comes initally with a dummy order.

```
init_socket(ip, port);

struct order initial;
read(newSocketFd, &initial, sizeof(struct order));

pide_p = initial.p;
pide_q = initial.q;
orderCount = initial.orderCount;
```

This way server know about the map’s size and number of orders and the client’s PID.

```
for (int i = 0; i < orderCount; i++)
{
    sem_post(&cook_sem); // Here is the order, cook it.
}
```

Manager signals to the cooks orderCount times to prepare the orders. Then enters a loop and waits for the signal of cooked order.

```
void* cook_thread(){
    while (preparedOrderCount < orderCount) {
        sem_wait(&cook_sem);

        if (exitCond != 0)
            break;

        prepare();
```

```
while (1) {
    if (pthread_mutex_trylock(&oven_entry1) == 0) {
        // Entry 1 is mine now.
        char *log = "Cook acquired oven entry 1.\n";
        log_it("server_log.log",log);

        pthread_mutex_lock(&oven_lock);
        if (insidePideCount < OVEN_SIZE) {
            sem_post(&inside_oven);
            pthread_mutex_unlock(&oven_lock);
            pthread_mutex_unlock(&oven_entry1);
            break;
        }
        pthread_mutex_unlock(&oven_lock);
        pthread_mutex_unlock(&oven_entry1);
        log = "Cook released oven entry 1.\n";
        log_it("server_log.log",log);

    } else if (pthread_mutex_trylock(&oven_entry2) == 0) {
        // Entry 2 is mine
        char *log = "Cook acquired oven entry 2.\n";
        log_it("server_log.log",log);

        pthread_mutex_lock(&oven_lock);
        if (insidePideCount < OVEN_SIZE) {
            sem_post(&inside_oven);
            pthread_mutex_unlock(&oven_lock);
            pthread_mutex_unlock(&oven_entry2);
            break;
        }
        pthread_mutex_unlock(&oven_lock);
        pthread_mutex_unlock(&oven_entry2);
        log = "Cook released oven entry 2.\n";
        log_it("server_log.log",log);
    }
```

Cook was already waiting for manager to signal that there is orders to prepare. Then cooks start to prepare orders.

After one preparation, cook goes and checks for the first oven entry, if it is available, goes to oven and checks if there is room for another order in the oven. If not, cook waits for an order to cook and get taken in oven, so there will be space and cook will place the new order in oven. If the first oven entry is not available, cook checks for the second entry and so on.

When all prepared orders are cooked. Cook signals to the manager “that is the last cooked order.” by assigning manager_order_finish = 1

```
void cook_it(){
    sem_wait(&oven_aparatus);
    sem_post(&oven_aparatus);

    usleep(pseudo_inverse / 2);
    char *log = "Order cooked.\n";
    log_it("server_log.log",log);

    pthread_mutex_lock(&finish_mutex);

    finishedOrders[finishedOrderCount] = orders[finishedOrderCount];
    finishedOrderCount++;
    if (finishedOrderCount == orderCount){
        log = "Cooking orders is finished.\n";
        log_it("server_log.log",log);

        manager_order_finish = 1;
    }

    sem_post(&finish);
    pthread_mutex_unlock(&finish_mutex);

}
```

Meanwhile, when the orders continuning cooking, when there is orders that cooked, manager got signaled and manager was passing them to the delivery queue.

```
while(manager_order_finish == 0){
    sem_wait(&finish);
    pthread_mutex_lock(&deliver_mutex);
    deliveryQueue[toBeDelivered] = orders[toBeDelivered];
    toBeDelivered++;
    pthread_mutex_unlock(&deliver_mutex);
    sem_post(&del_sem);
    // send it to delivery
}
```

And delivery persons was already waiting for a signal.

```
int package_x , package_y;

while (exitCond == 0)
{
    sem_wait(&del_sem);

    if (exitCond != 0)
        break;

    pthread_mutex_lock(&deliver_mutex);

    if (toBeDelivered > 0 && delivering < orderCount)
    {
        package_x = deliveryQueue[delivering].x;
        package_y = deliveryQueue[delivering].y;
        delivering++;
    }

    pthread_mutex_unlock(&deliver_mutex);

    deliver(package_x, package_y);
```

When all cooked orders are delivered, last delivery person signals the manager one last time.

```
pthread_mutex_lock(&deliver_mutex);
if (delivering == orderCount)
{
    sem_post(&finish);
    pthread_mutex_unlock(&deliver_mutex);
    break;
}
pthread_mutex_unlock(&deliver_mutex);
```

Manager gets the signal and cleans up everything then manager will leave too.

```
sem_wait(&finish); // Wait for the last delivery
log = "Delivering finished. Exiting gracefully.\n";
log_it("server_log.log",log);

exitCond = 1;
for (int i = 0; i < OVEN_SIZE; i++)
{
    sem_post(&inside_oven); // wake up and exit.
}
for (int i = 0; i < cookNum; i++)
{
    sem_post(&cook_sem); // wake up and exit.
}
for (int i = 0; i < delNum; i++)
{
    sem_post(&del_sem); // wake up and exit.
}

return NULL;
```