Göktuğ Ali Akın
161044018

# CSE 344
# SYSTEM PROGRAMMING

# PROJECT 5 REPORT

## Problem definition

In this project, there are two actors that simulates the flower delivery system. Clients will make requests for flowers, a central thread will collect those requests, and then delegate the work to the closest florist that has the kind of flower requested.

## Parsing the file and storing the actors

Program, first reads the contents of the file to create necessary tools for manipulation of the florists,clients and tools for synchronization.

Types for the manipulation of the file data ;

- **client_t** : represents the client.
- **florist_t** : represents the florist.
- **point_t** : represents the 2D point.
- **request_queue** : represents the requests queue that shared between central thread and related florist.
- **sale_static_t** : represents the sale statics for florists.

**__init__** function reads the file and creates data structures and synchronization tools into this global variables ;

- **int total_florist :** Holds the total florists.
- **int total_client** : Holds the total clients. This information got at the end of the file.
- **florist_t* all_florists** : Holds all florists as florist_t struct array.
- **client_t* all_clients** : Holds all clients as client_t struct array.

- **int* florist_numbers** : Holds florist numbers for all florists to acces them easily.
- **pthread_t* florist_threads** : Holds all florist threads.
- **pthread_mutex_t* all_mutexes** : Holds all mutexes for all florists. They must have different mutexes since they have different request queues.
- **pthread_cond_t* all_condvars :** Holds all condition variables for all florists. They must have different conditional variables since they have different request queues.
- **request_queue** all_queues :** Holds all request queues as arrays for different florists.

Now, we parse the file into useful data structures. We can iterate this data structures instead of iterating the file line by line.

**Note :** Florists do not know the number of clients for synchronization.

**Central thread**

Central thread will push the requests into related florists request queue. This requested queue represented as **requested_queue** struct. This struct contains array implementation of basic queue. Types of the items in the queue is **client_t** type.

Central thread, first gets the client from client array by accessing with all_clients[i], then get the closest valid florist for client. **get_closest_florist()** function does this work, it will return the number (index) of closes florist by looking the Chebyshev distance.

**Florists**

Florists are waiting request from central thread. If they see that there is at least one request, they process them and wait again. This is done by conditional variable and mutex.

**Exit condition of florists** : If central thread says that theres is no incoming client from file, central thread assign true to **delivery_done** global variable. If the related florists request is empty and delivery_done marked true by central thread, this florist thread can stop their execution and return the sale statics.

## Synchronization problems

### 1) Producer - Consumer problem

There is a producer consumer problem between central thread and florist threads. So who is who ? Let's look them one by one ;

**Producer** : Central thread. Central thread produces requests / orders for florists. And push this requests into their request queues.

**Consumer** : Florist thread. Florist thread consumes the requests / orders from their request queue which is fill by producer (central thread)

**Product** : The product that central thread produces is request. A request is actually the all information of client. So, request queues holds clients as **client_t** type.

This producer consumers was solved by conditional variables and mutexes.

### 2) Synchronization barriers

There must be some synchronization barriers in this program to work properly.

**Central thread must wait all florists to finish their jobs**

After central threads read requests and give them to related florists, it must not be exit directly. If it exits directly, there can be delivery loss. Central thread must make sure that all florists done their deliveries. If all deliveries done by the florists, central thread continue its execution.

This is done by one mutex and on conditional variable. Every florist thread, increments the global variable **total_delivered,** and signals the conditional variable **total_delivered_condvar** to wake up the central thread.

Code snippet of florist side ;

```
pthread_mutex_lock(&total_delivered_mutex);
++total_delivered;
pthread_cond_signal(&total_delivered_condvar);
pthread_mutex_unlock(&total_delivered_mutex);
```

Code snippet of central thread side ;

```
central_thread(); // give requests to the florists
/* now, we know the # of clients / requests */
pthread_mutex_lock(&total_delivered_mutex);
/* wait until all flowers are delivered */
while(total_delivered < total_client)
        pthread_cond_wait(&total_delivered_condvar,&total_delivered_mutex);
pthread_mutex_unlock(&total_delivered_mutex);
```

**Central thread must wait until all florists are closing their shop (exiting from thread) to print the sale statics**

In order to print the sale statics, in the same time with nicely formatted, central thread must wait until all florists are closing their shop. This is done by conditional variable and mutex.

Code snippet ;

```
pthread_mutex_lock(&florist_done_mutex);
while(florist_done < total_florist)
        pthread_cond_wait(&florist_done_condvar,&florist_done_mutex);
```

**Stopping the execution of florist threads**

If all deliveries are done by the florists, it is time to closing the shops (terminate the florist threads).

Central thread, assign true to delivery_done global variable that represents the deliveries are done by the all florists. This means that, there will be no other requests will be given to florists. If request queue is empty and delivery_done flag is 1, florist thread can exit.

Code snippet of central thread ;

```
for (int i = 0; i < total_florist; ++i) // lock all mutexes
        pthread_mutex_lock(&all_mutexes[i]);
delivery_done = 1; // mark as done
for (int i = 0; i < total_florist; ++i) // signal the florist threads
        pthread_cond_signal(&all_condvars[i]);
for (int i = 0; i < total_florist; ++i)
        pthread_mutex_unlock(&all_mutexes[i]);
```

Code snippet of florist thread ;

```
pthread_mutex_lock(&all_mutexes[florist_no]);
while(all_queues[florist_no]->size == 0 && !delivery_done)
        pthread_cond_wait(&all_condvars[florist_no],&all_mutexes[florist_no]);
if(all_queues[florist_no]->size == 0 && delivery_done) // exit condition
                break;
```

By this way, when central thread marks the **delivery_done** flag as true and signals the florist threads, they can break out of the loop and return their sale statics.

## Other implementation notes

### Sleeping in milliseconds

Since, usleep() has been deprecated on some POSIX versions, I used manual implementation for usleep() using nanosleep().

For more details, see : https://stackoverflow.com/questions/1157209/

### Utility functions

“utils.h” header file and “utils.c” file contains the utility functions for manipulating the strings, sleeping in ms, generate random numbers and parsing & reading the file etc.

Other implementation details, explanations can be found in the source code, comments just before the function definition.