# CSE344 – System Programming - Signal Handling, HW5 Report

Ahmed Semih Özmekik, 171044039
June 7, 2020

**Abstract**

Demonstration of solutions for problems encountered in the homework 5.

## 1 Problem

We can say that there are 2 different problems that we need to deal in the homework. First, it is just a problem for the necessity of a model to represent the components and data of the problem, which we actually encounter in every problem. The second is to choose a florist among the waiting florists and assign a suitable task for him. It is a synchronization problem to arrange the waiting and activation stages of the florist that is desired to act here. We are expected to solve this problem only with the allowed tools.

Now let's open up the sync problem a little more because we consider it more important and think it needs more explanation. Let's get to know the actors. First of all, we can say that there is a supervisor actor that will be the main thread. Supervisor's job is to send a florist to that client by looking at the client list so that each client's delivery can be handled in order. For clients, our criterion for assigning a florist is that, according to a distance unit calculated by the Chebyshev distance method, it must be the closest florist and at the same time the flower that the client desires must be in sale in that florist.

The task of Supervisor is to find the appropriate florist for the client and activate the appropriate florist thread. One of the main points in our problem is this: In these appointments made by Supervisor, we assume that the florist in question is still working. (We realize this assumption with the sleep () function in implementation.) In this case, the supervisor will not have to wait for a florist to finish his job. That is, the supervisor will constantly assign jobs to florists (often the same florist) and will move on to the next appointment as soon as his assignment is over. These are the elements in our problem. Now let's move on to the solutions.

## 2 Solution

### 2.1 Design Solution

In the design, the model objects created for the actors we consider are as follows. All of the complete objects are located in the model model.h header file.

```c
#ifndef MODEL_H
#define MODEL_H

#define NAME_LENGTH 32

/***
 * MODELS HEADER FILE REPRESENTING DATA
 * represents the model objects to represents the actors in the homework.
 * @see program.c
 ***/


struct Point // represents a point in a cartesian coordinat system
{
    float x, y;
};

struct Client
{
    char name[NAME_LENGTH];
    struct Point location;
    char flower[NAME_LENGTH]; // client demands 1 flower.
};

struct Florist
{
    char name[NAME_LENGTH];
    struct Point location;
    float speed;
    char **flower; // florist delivers n kind of flowers.
    int total_sale, total_time;
    unsigned int flower_count;
};

#endif
```

## 2.2 Sync Solution

First of all, let's underline that we need a data structure that will assist us in solving our aforementioned problem without focusing on synchronization. As it is presented to us as a hint in the homework, a better structure cannot be considered for this data structure than the queue structure. We assign a task to the queue for a florist in each assignment. And a florist will continue to do business as long as he has an item in it's queue. The required queue structure is defined by representing an item as a client with a simple implementation.

Let us examine the actors' sync approaches step by step by showing the implementation.

```c
supervisor()
{
    ...
    /* processing requests */
    for (size_t i = 0; i < cm.count; i++)
    {
        size_t florist = find_florist(&fm, *cm.client[i]);
        lock(fm.mutex);
        enqueue(fm.queue_list[florist], *cm.client[i]); // add the flower to queue of the florist
    .
        broadcast(fm.cond);                              // wake up florist.
        unlock(fm.mutex);
    }
    /* end of processing requests */

    // signal finished.
    lock(fm.mutex);
    *fm.finished = TRUE;
    broadcast(fm.cond);
    unlock(fm.mutex);
    ...
}
```

As can be seen, the supervisor firstly determines which client will assign a florist in a separate section and in order, in advance. Then he takes the lock of mutex and adds a client to the queue of the florist in the appropriate index. And then it broadcasts the condition variable, waking up all the florists. With this awakening, only the florist, who is a client in his queue, that it's queue is not empty, will be put to work. All others will fall asleep again. Then, after the process of the requests is finished, it will flash a finish signal and indicate that the client will not come to the florists again.

Let's examine the approach of florists.

```c
florist()
{
    for (;;)
    {
        lock(sync->mutex);
        while (is_empty(sync->queue) && !(*sync->finished))
            cwait(sync->cond, sync->mutex);
        if (*sync->finished && is_empty(sync->queue))
        {
            unlock(sync->mutex);
            break;
        }
        struct Client client = dequeue(sync->queue);
        unlock(sync->mutex);
        int prep_time = random_time() + sync->florist->speed * distance(sync->florist, &client);
        usleep(prep_time);
        sync->florist->total_time += prep_time;
        sync->florist->total_sale++;
        printf("\nFlorist %s has delivered a %s to %s in %dms", sync->florist->name,
               flower_to_str(client.flower), client.name, prep_time);
    }
}
```

A florist is going to work by first getting the lock of mutex. The condition for Florist to wait is that his own queue to be empty and the job assignment is not finished yet, i.e. the finished signal isn't given. As we have just seen, two situations can cause him to wake up. Either the finish signal was given or a task was assigned to him, that is, an item came to his queue.

Let's say a task has been assigned to him. In this case, he checks if the finished signal has been given to find out the reason for the wake-up. If the finish signal has not been given yet, it immediately takes the client from the queue and starts the business by giving mutex.

If it was awakened because of the finish signal, this time he checks: Do I still have a task to do? In other words, do I still have a client in my queue? If he still has tasks to do, he continues to do it as expected, even if the supervisor's assignments are over. If he has no work to do and the finish signal is given, he immediately exits by releasing the mutex.

This is our solution. Our job is remarkably eased thanks to the use of mutexes and condition variables.