CSE344 – System Programming - Signal Handling, HW4 Report

Ahmed Semih Özmekik, 171044039
May 17, 2020

Abstract

Demonstration of solutions for problems encountered in the homework.

1 Problem

The main problem in this assignment was the synchronization problem. Many other things besides this were a simple design choice. For example, how we would represent ingredients and chefs as data structure was a design problem. So I can say that it was taking shape on two problems. The first is a set of design choices, and the other is a synchronization problem. Now, in the solution section, I will both detail these two problems and explain the solutions I have brought.

2 Solution

2.1 Design Solution

By design, we have to observe this rule in our design here, since there must be a genericity (ie, chefs must use the same function). I simply used an integer mask to represent the ingredients. And I assigned a mask to an array for each binary group. In this way, we determine from the beginning what chef will wait and what. Even if we did not determine, we could randomly choose among these indices. The main part of the design can be seen as follows:

```
/ we will use bitwise mask to represent ingredients.
enum Ingredient
{
    MILK = 2,
    FLOUR = 4,
    WALNUTS = 8,
    SUGAR = 16,
    EMPTY,
};
const int DISCRETE_MASKS[CHEF_NUMBER] = {
    MILK | FLOUR,
    MILK | WALNUTS,
    MILK | SUGAR,
    FLOUR | WALNUTS,
    FLOUR | SUGAR,
    WALNUTS | SUGAR,
};
```

In this way, actually; ”The ingredients will be delivered to a data structure of your choosing (e.g. array, stack, etc), located at the heap and thus shared among all involved threads. In this way, The approach I brought to the rule ”The ingredients will be delivered to a data structure of your choosing (e.g. array, stack, etc),located at the heap and thus shared among all involved threads.” is as follows. With this design, we can directly store what the wholesaler brings in a single integer variable. For example it was simply done as follows:

```
/** THREAD & SYNC STRUCTURE **/
struct Resources
{
    ...
    int *store; // a dummy data structure to simulate storing ingredients delivered by wholesaler
    .
    sem_t *store_mutex;
```

```
};

// called by wholesaler
void deliver(struct Resources *ws, int mask)
{
    xsem_wait(ws->store_mutex);
    *ws->store = mask;
    xsem_post(ws->store_mutex);
}
```

2.2 Sync Solution

Our problem is very similar to the problem of cigarette smokers we saw in the lesson. The only difference is that every consumer (that is, the chef for this problem) lacks 2 items.

So, this problem could be difficult to approach with POSIX semaphore. So we could use **sem_trywait**. However, since every consumer lacks 2 items, we would not have realized a fully efficient solution here either. Perhaps a little busy waiting would have occurred.

Instead, using System V semaphores, we can solve this using 2 semaphores directly by doing post and wait. There are 4 semaphore as there are 4 ingredients in total. According to the 2 ingredients that every chef expects to come, only the semaphores that he wants is waiting. The wholesaler posts the semaphore of the ingredients found in the file. Thanks to System V, semaphores, we were able to solve the problem easily. That's our all sync resources shared between wholesaler and chef threads.

In order to maintain simplicity in other semaphores, I preferred POSIX semaphore. Below, our solution can be examined better in the parts of the code related to sync.

```
/** THREAD & SYNC STRUCTURE **/
struct Resources
{
    int index;
    int *finished; // to indicate supplier is finished delivery.
    int sem_ingred;
    sem_t *dessert; // chefs will be posting desert.
    sem_t *finish_mutex;
    int *store; // a dummy data structure to simulate storing ingredients delivered by wholesaler
    .
    sem_t *store_mutex;
};

wholesaler()
{
    ...
    while(TRUE)
    {
        printf("\nthe wholesaler delivers %s", print_mask(mask, str));
        sem_of_ingredients(mask, sem_index);
        deliver(ws, mask);
        systemV_sempost(ws->sem_ingred, sem_index); // post two ingredients.
         // wait for dessert.
        printf("\nthe wholesaler is waiting for dessert");
        xsem_wait(ws->dessert);
        printf("\nthe wholesaler has obtained the dessert and left to sell it");
    }
    ...
}

void chef()
{
 ...
   while (!finished(ct))
    {
       ...
```

```
36          systemV_semwait(ct->sem_ingred, sem_index); // grab 2 ingredients.
37          if (finished(ct))
38              break;
39          take_delivery(ct, i);
40          prepare_dessert(i);
41          sem_post(ct->dessert); // dessert is ready.
42      }
43      ...
44  }
```