Project 2 Report

王智烨 521021911051

Project Files

```
Prj2+521021911051
    report.pdf
    report.md
    Typescript.pdf
    faneuil.c
    LarryCurlyMoe.c
    makefile
```

Experiments

The contents of 2 experiments of Project 2 and the analysis are presented below, arranged in the order of the problems.

Part 1: Stooge Farmers Problem

Introduction

Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Curly then fills the hole up. There are several synchronization constraints:

- Moe cannot plant a seed unless at least one empty hole exists, but Moe does not care how far Larry gets ahead of Moe.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed, but the hole has not yet been filled. Curly does not care how far Moe gets ahead of Curly.
- Curly does care that Larry does not get more than MAX holes ahead of Curly. Thus, if there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.
- Command line

  ```
  ./LCM Maxnum    //Maxnum: Max number of unfilled holes
  ```

Analysis

- Semaphore

```
sem_t Shovel;//1
sem_t filled;//Max
sem_t unfilled;//0
sem_t unfilled_with_seed;//0
//Initialize
sem_init(&Shovel,0,1);
sem_init(&filled,0,Maxnum);
sem_init(&unfilled,0,0);
sem_init(&unfilled_with_seed,0,0);
```

In this part, I use 3 semaphore to arrange the work of Larry, Moe and Curly.

As there is only a shovel, so the semaphore `Shovel` will be set to 1. For the similar reason, semaphore `filled`, which indicates how many holes need to be dug, seeded and filled, is set to `Maxnum`.

- Function

  The work of Moe, Curly and Larry are declared respectively in functions `func_Larry`, `func_Moe` and `func_Curly`.

```
void *func_Larry(void *args)
{
    int Holes_dug = 0;
    while (Holes_dug < Maxnum)//whether Larry dig enough holes
    {
        sem_wait(&filled);//to check whether there are holes unfilled
        sem_wait(&Shovel);// is Shovel available
        //Larry's work...
        sem_post(&Shovel);// Shovel is available now
        sem_post(&unfilled);// a hole unfilled
    }
    pthread_exit(NULL);
}

void *func_Moe(void *args)
{
    int Holes_seeded = 0;
    while (Holes_seeded < Maxnum)//whether Moe seed enough holes
    {
        sem_wait(&unfilled);//to check whether there are holes unseeded
        //Moe's work...
        sem_post(&unfilled_with_seed);//an unfilled hole but with seed
    }
    pthread_exit(NULL);
}

void *func_Curly(void *args)
{
    int Holes_filled = 0;
    while (Holes_filled < Maxnum)//whether Curly fill enough holes
    {
        sem_wait(&unfilled_with_seed);// whether there is an unfilled hole
with seeds
        sem_wait(&Shovel);//is Shovel availbale
        //Curly's work
        sem_post(&Shovel);// Shovel is available now
        sem_post(&filled);//In fact, this is not necessory
    }
    pthread_exit(NULL);
}
```

Part 2: The Faneuil Hall Problem

Introduction

There are three kinds of threads: *immigrants*, *spectators*, and one *judge*. Immigrants must wait in line, check in, and then sit down. At some point, the judge enters the building. When the judge is in the building, no one may enter, and the immigrants may not leave. Spectators may leave. Once all immigrants check in, the judge can confirm the naturalization and immigrants who sitted down can be confirmed. After the confirmation, the immigrants who are confirmed by the judge swear and pick up their certificates of U.S. Citizenship, while the others wait for the next judge. The judge leaves at some point after the confirmation. Spectators may now enter as before. After immigrants get their certificates, they may leave.

To make these requirements more specific, let's give the threads some functions to execute and put constraints on those functions.

- Immigrants must invoke *enter*, *checkIn*, *sitDown*, *swear*, *getCertificate* and *leave*.
- The judge invokes *enter*, *confirm* and *leave*.
- Spectators invoke *enter*, spectate and *leave*.
- While the judge is in the building, no one may *enter* and immigrants may not *leave*.
- The judge can not confirm until all immigrants, who have invoked enter, have also invoked *checkIn*.
- Command Line

  ```
  ./faneuil
  ```

Analysis

- Semaphore

  ```
  sem_t judIn, check, sitDown, immConfirmed[MAX_THREAD];
  sem_t immEnter;//to prevent judege enter while there is no immigrant in the hall
  // initialize
  sem_init(&judIn, 0, 1);
  sem_init(&immEnter, 0, 0);
  sem_init(&check, 0, 0);
  sem_init(&sitDown, 0, 0);
  for (i = 0; i<MAX_THREAD; ++i)
          sem_init(&immConfirmed[i], 0, 0);
  ```

  `judIn` is used to show there is a judge in the hall, so no immigrants can enter or leave, no spectators can enter and no the other judge can enter in.

`check` is used to help judge check whether all the immigrants who entered the hall has finished check.

`sitDown` is similar to the `check`, help judge to check whether all the immigrants sat down.

`immConfirmed[]` is an array to record whether the immigrant has been confirmed, so the immigrant can *swear* and *getCertificate*.

`immEnter` is a special semaphore to prevent starvation. The judge will not enter the hall until there is an immigrant in it. Also, I didn't pay attention to whether the immigrant had been confirmed.

- Function

```
void *func_immigrant()
{
    sem_wait(&judIn);
    sem_post(&judIn);
    int imm_no = immNum;
    ++ immNum;
    ++ uncheck;
    ++ unconfirmNum;
    //enter
    shortdelay();
    printf("Immigrant #%d enter\n", imm_no);
    sem_post(&immEnter);
    //checkIn
    shortdelay();
    printf("Immigrant #%d checkIn\n", imm_no);
    sem_post(&check);
    --uncheck;

    shortdelay();
    printf("Immigrant #%d sitDown\n", imm_no);
    push(&Sitdown, imm_no);//save the no of the immigrant
    sem_post(&sitDown);

    sem_wait(&immConfirmed[imm_no]);
    shortdelay();
    printf("Immigrant#%d swear\n", imm_no);
    shortdelay();
    printf("Immigrant#%d getCertificate\n", imm_no);

    sem_wait(&judIn);
    shortdelay();
    printf("Immigrant #%d leave\n", imm_no);
    sem_post(&judIn);
    sem_wait(&immEnter);

    pthread_exit(NULL);
}

void *func_spectator()
{
    sem_wait(&judIn);
    sem_post(&judIn);
    int spec_no = specNum;
    ++specNum;

    shortdelay();
    printf("Spectator #%d enter\n", spec_no);

    shortdelay();
    printf("Spectator #%d spectate\n", spec_no);

    shortdelay();
    printf("Spectator #%d leave\n", spec_no);

    pthread_exit(NULL);
}

void *func_judge()
{
    sem_wait(&immEnter);
    sem_post(&immEnter);

    sem_wait(&judIn);

    int jud_no = judNum;
    ++judNum;
    shortdelay();
    printf("Judge #%d enter\n", jud_no);

    //wait for unchecked immigrants
    for (int i = 0; i < uncheck; ++i)
        sem_wait(&check);

    if (unconfirmNum != 0)
    {
        do{
            sem_wait(&sitDown);
            int imm_no = pop(&Sitdown);
            shortdelay();
            printf("Judege #%d confrm the immigrant #%d \n", jud_no,
imm_no);
            --unconfirmNum;

            sem_post(&immConfirmed[imm_no]);
        }while(unconfirmNum > 0);
    }
    shortdelay();
    printf("Judge #%d leave\n", jud_no);
    sem_post(&judIn);
    pthread_exit(NULL);
}
```

- Multithread

  The way I implement multithread here uses a function `new_thread`, which will randomly generate an immigrant or a spectator or a judge. This is based on the method shown by the teacher Lin in class.

```
void new_thread(int tid)
{
    switch(rand()%3)
    {
        case 0:pthread_create(&threads[tid], NULL, func_immigrant,
NULL);break;
        case 1:pthread_create(&threads[tid], NULL, func_spectator,
NULL);break;
        case 2:pthread_create(&threads[tid], NULL, func_judge, NULL);break;
    }
}
//main
while(1)
    {
        new_thread(Nthread);
        ++Nthread;
        delay();
    }
```

- SitDown Array

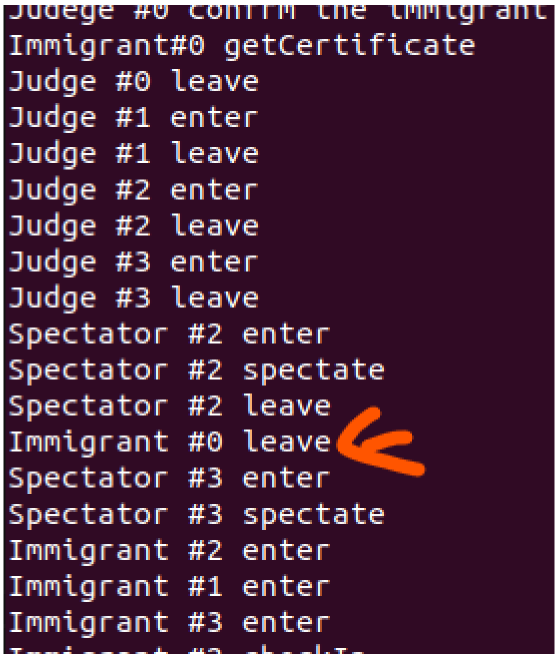
  To record the immigrant who have sat down but not been confirmed, I build a queue in my program, and maintain it when an immigrant sit down or is confirmed.

```
shortdelay();
printf("Immigrant #%d sitDown\n", imm_no);
push(&Sitdown, imm_no);//save the no of the immigrant
```

```
int imm_no = pop(&Sitdown);
shortdelay();
printf("Judege #%d confrm the immigrant #%d \n", jud_no, imm_no);
--unconfirmNum;
```

Some Fun facts

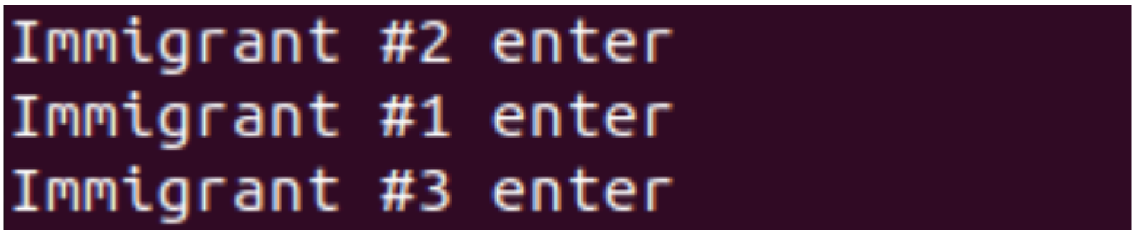
- Many judges!



```
Judege #0 confrm the immigrant
Immigrant#0 getCertificate
Judge #0 leave
Judge #1 enter
Judge #1 leave
Judge #2 enter
Judge #2 leave
Judge #3 enter
Judge #3 leave
Spectator #2 enter
Spectator #2 spectate
Spectator #2 leave
Immigrant #0 leave
Spectator #3 enter
Spectator #3 spectate
Immigrant #2 enter
Immigrant #1 enter
Immigrant #3 enter
```

As stated above, I use semaphore `immEnter` to prevent many judges enter the hall and there is no immigrants in it. However, after I ran the program, I still found such multiple judges entering the hall in succession.

After careful study and observation, it was discovered that the reason for this was that one immigrant had completed the process but did not leave in time, leading the judge to believe that there were still immigrants inside who had not completed the process.

- Come FIRST leave LAST
- 

```
Immigrant #2 enter
Immigrant #1 enter
Immigrant #3 enter
```

  In the test, I also found that sometimes some immigrants with more advanced serial numbers completed some actions faster.

  The reason is that the `print()` is after `shortdelay()`.So in some cases, some of the first immigrants to appear but enter the hall at the end. Corresponding to the real situation, we can assume that these people walk more slowly.