Project 2 Report

陈潇雨 523031910681

Experiments

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

Part 1: 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 judge,mutex,check_judge,check_in,swear_sem;
// initialize
sem_init(&judge, 0, 1);
sem_init(&mutex, 0, 1);
sem_init(&check_in, 0, 1);
sem_init(&check_judge, 0, 1);
sem_init(&swear_sem, 0, 0);
```

- `judge` 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.
- `mutex` is used to protect shared resources and critical sections, ensuring operations on shared variables like `waiting_count` and `judge_present` are atomic.
- `check_in` is used to make sure immgrants check in order.
- `check_judge` is used to help judge check whether all the immigrants who entered the hall has finished check.
- `swear_sem` is used to control the swearing process, allowing immigrants to swear only after being confirmed by the judge, ensuring the correct sequence of naturalization steps.

Fuction

enter and leave

```
void enter(int specifiers,int id){
    if(specifiers == judges){
        sem_wait(&judge);
        judge_present = 1;
        printf("%s #%d enter\n", role_names[specifiers], id);
```

```
    }
    else if(specifiers == immgrants){
        sem_wait(&judge);
        to_check++;
        if(to_check == 1){
            sem_post(&check_judge);
        }
        printf("%s #%d enter\n", role_names[specifiers], id);
        sem_post(&judge);
    }
    else{
        sem_wait(&judge);
        printf("%s #%d enter\n", role_names[specifiers], id);
        sem_post(&judge);
    }
}
```

```
void leave(int specifiers,int id){
    if(specifiers == judges){
        printf("%s #%d leave\n", role_names[specifiers], id);
        sem_post(&judge);
    }
    else if(specifiers == immgrants){
        sem_wait(&mutex);
        if(judge_present) {
            sem_post(&mutex);
            sem_wait(&judge);
            sem_post(&judge);
            sem_wait(&mutex);
        }
        printf("%s #%d leave\n", role_names[specifiers], id);
        sem_post(&mutex);
    }
    else {
        printf("%s #%d leave\n", role_names[specifiers], id);
    }

}
```

The functions above can be invoked by all kinds of threads. But their behaviors differ significantly for each role:

- Judge：
  When entering,it dosen't release the `judge` semaphore, thus making others unable to enter.When leaving, judge simply releases the `judge` semaphore and prints the departure message.
- Immigrants：
  For immigrants, both entering and leaving require careful synchronization. When entering, they must check for judge's presence using `judge` semaphore and increment the `to_check` counter. If they are the first immigrant `(to_check == 1)`, they signal `check_judge` to notify the judge. When leaving, they face stricter control - they must verify if a judge is present using `judge`.
- Spectators：
  Spectators follow the simplest protocols - when entering, they briefly acquire the judge semaphore to ensure no judge is present, then release it immediately after entry. For leaving, they have complete freedom and can exit without any semaphore operations, as their movements don't affect the core naturalization process.

func_immigrant

```
void *func_immigrant() {
    int id = immgrantnum++;
    enter(immgrants, id);
    sleep(rand() % NUM);
    checkIn(id);
    sleep(rand() % NUM);
    sitdown(id);
    sem_wait(&swear_sem);
    printf("Immigrant #%d swear\n", id) ;
    sleep(rand() % NUM);
    printf("Immigrant #%d getCertificate\n", id);
    leave(immgrants, id);
    return NULL;
}
```

```
void checkIn(int id){
    sem_wait(&check_in);
    to_check--;
    sleep(rand()%NUM); // Simulate time taken to check in
    printf("Immigrant #%d checkIn\n", id);
    if(to_check == 0){
        sem_post(&check_judge);//all check in then judge can confirm
    }
    sem_post(&check_in);
```

```
void sitdown(int id){
    sem_wait(&mutex);
    printf("Immigrant #%d sitDown\n", id);
    waiting_immigrants[waiting_count++]=id;
    sem_post(&mutex);
}
```

The `func_immigrant` function orchestrates the naturalization process through several key steps. Besides `enter` and `leave`, it invokes `checkIn()` where immigrants acquire the `check_in` semaphore for orderly registration, decrements `to_check`, and signals the judge via `check_judge` semaphore when all immigrants have checked in.

After checking in, immigrants call `sitdown()` which records their ID in `waiting_immigrants` array and increments `waiting_count` under `mutex` protection. They then wait on the `swear_sem` semaphore to be released by the judge's confirmation before proceeding with their oath and receiving their certificates.

func_judge

```
void *func_judge(){
    int id = judgenum++;
    enter(judges, id);
    sleep(rand()%NUM);
    confirm_immigrant(id, waiting_immigrants);
    sleep(rand()%NUM);
    leave(judges, id);
    return NULL;
}
```

```
void confirm_immigrant(int judge_id, int *waiting_immigrants){
    sem_wait(&check_judge);
    sem_wait(&mutex);

    for(int i=0;i<waiting_count;i++){
    printf("Judge #%d confirm the immigrant #%d\n", judge_id, waiting_immigrants[i] );
    sem_post(&swear_sem);
}
    waiting_count = 0;
    sem_post(&mutex);
    sem_post(&check_judge);
}
```

The `func_judge` function manages the judge's confirmation process for naturalization. Beyond `enter` and `leave` operations, it calls `confirm_immigrant()` where the judge processes waiting immigrants based on `waiting_count` .

During confirmation, protected by `mutex` semaphore, the judge confirms each immigrant in the `waiting_immigrants` array and releases the `swear_sem` semaphore to allow them to proceed with their oath. After confirming all waiting immigrants, the judge resets `waiting_count` to 0. The `mutex` protection ensures this process is atomic and won't be interrupted by immigrants calling `sitdown()` .

func_spectator

```
void *func_spectator(){
  int id=specatornum++;
    enter(specators, id);
    sleep(rand()%NUM);
    printf("Spectator #%d spectate\n", id);
    sleep(rand()%NUM);
    leave(specators, id);
    return NULL;
}
```

The `func_spectator` function implements the simplest role in the naturalization process. After obtaining a unique ID, spectators enter the hall, spend random intervals (simulated by `sleep()`) observing the proceedings and then leave. Unlike judges and immigrants, spectators have minimal interaction with the synchronization mechanism - they only

need to ensure no judge is present when entering, and can leave freely at any time without additional checks or synchronization requirements.

Test runs

To demonstrate the program execution, here is a sample output:

```
(base) root@LAPTOP-G6UNDHO9:/mnt/d/Code# ./faneuil
Spectator #0 enter
Spectator #1 enter
Spectator #0 spectate
Immigrant #0 enter
Spectator #1 spectate
Immigrant #0 checkIn
Immigrant #0 sitDown
Spectator #0 leave
Spectator #1 leave
Spectator #2 enter
Immigrant #1 enter
Spectator #2 spectate
Spectator #2 leave
Judge #0 enter
Immigrant #1 checkIn
Immigrant #1 sitDown
Judge #0 confirm the immigrant #0
Judge #0 confirm the immigrant #1
Immigrant #0 swear
Immigrant #1 swear
Immigrant #1 getCertificate
Judge #0 leave
Judge #1 enter
Immigrant #0 getCertificate
Judge #1 leave
Spectator #3 enter
Immigrant #1 leave
Immigrant #2 enter
Immigrant #0 leave
Spectator #4 enter
Spectator #3 spectate
Immigrant #3 enter
Spectator #3 leave
```

Since the Faneuil Hall Problem runs in an infinite loop by design, we only demonstrate a partial execution output to illustrate the synchronization between judges, immigrants, and spectators. But the output demonstrates the key aspects of our synchronization solution, including proper ordering of naturalization steps and coordination between different roles.

Part 2: The Santa Claus problem

Introduction

This problem demonstrates the use of semaphores to coordinate three types of processes: Santa Claus,reindeer, and elves.Santa Claus sleeps in his shop at the North Pole and can only be wakened by either (1) all nine reindeer being back from their vacation in the South Paciffc, or (2) by some of the elves having difffculties making toys. Suppose we have the following scenario:

- To allow Santa to get some sleep, the elves can only wake him when three of them have problems.While three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return.
- If Santa wakes up to ffnd three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.)
- The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.
- After the ninth reindeer arrives, Santa must invoke `prepareSleigh`, and then all nine reindeermust invoke `getHitched`.
- After the third elf arrives, Santa must invoke `helpElves`. Concurrently, all three elves should invoke `getHelp`.
- All three elves must invoke `getHelp` before any additional elves enter (increment the elf counter).
- Command Line

```
./santa
```

Analysis

Semaphore

```
sem_t mutex;
sem_t santa_sleep;
sem_t reindeer_wait;
sem_t elves_wait;
```

```
sem_t elf_queue;
sem_t reindeer_done;
sem_t elves_done;
sem_t reindeer_barrier;
// initialize
sem_init(&santa_sleep, 0, 0);//wake up Santa
sem_init(&reindeer_wait, 0, 0);//getHitchedv in queue
sem_init(&elves_wait, 0, 0);//get help in queue
sem_init(&mutex, 0, 1);//proction
sem_init(&elf_queue, 0, 3);//others wait when Santa helps the three before
sem_init(&reindeer_done, 0, 0);//concurrently
sem_init(&elves_done, 0, 0);//concurrently
sem_init(&reindeer_barrier, 0, REINDEER_COUNT); //prevent reindeer waiting while some haven't been getHitched
```

- `santa_sleep` is used to wake up Santa when either 3 elves need help or all 9 reindeer have returned from vacation.
- `mutex` is used to protect shared resources and critical sections, ensuring operations on shared variables like `reindeer_num` and `elf_num` are atomic.
- `reindeer_wait` is used to control reindeer waiting for Santa to prepare the sleigh, ensuring they get hitched in order.
- `elves_wait` is used to control elves waiting for Santa's help, making sure they receive help in groups of three.
- `elf_queue` is used to limit the number of waiting elves to 3, preventing more elves from joining while Santa is helping.
- `reindeer_done` is used to signal Santa that a reindeer has finished getting hitched, ensuring synchronization during sleigh preparation.
- `elves_done` is used to signal Santa that an elf has received help, ensuring synchronization during help sessions.
- `reindeer_barrier` is used to prevent new reindeer from entering the waiting queue until all current reindeer have been hitched, ensuring orderly sleigh preparation cycles.

Fuction

func_santa()

```
void *func_santa() {
```

```
while(1) {
    if(count == 30) {
        sem_wait(&mutex);
        program_finished = 1;

        for(int i = 0; i < ELF_COUNT; i++) {
            sem_post(&elves_wait);
            sem_post(&elf_queue);
        }

        for(int i = 0; i < REINDEER_COUNT; i++) {
            sem_post(&reindeer_wait);
            sem_post(&reindeer_barrier);
        }

        sem_post(&mutex);
        break;
    }
    sem_wait(&santa_sleep);
    sem_wait(&mutex);

    if(reindeer_num == 9) {
        santa_state = 1;
        printf("Santa: preparing sleigh\n");


        for(int i = 0; i < 9; i++) {
            sem_post(&reindeer_wait);
        }
        sem_post(&mutex);
        for(int i = 0; i < 9; i++) {
            sem_wait(&reindeer_done);
        }

        sem_wait(&mutex);
        reindeer_num = 0;
        santa_state = 0;
        count++;
        for(int i = 0; i < 9; i++) {
            sem_post(&reindeer_barrier);
        }
        sem_post(&mutex);
```

```
        }
        else if(elf_num == 3 && elves_waiting) {
            santa_state = 2;
            printf("Santa: help elves\n");

            for(int i = 0; i < 3; i++) {
                sem_post(&elves_wait);
            }
            sem_post(&mutex);
            for(int i = 0; i < 3; i++) {
                sem_wait(&elves_done);
            }

            sem_wait(&mutex);

            santa_state = 0;
            sem_post(&mutex);
        }
    }
    return NULL;
}
```

1.Program Termination Logic

The first part handles program termination after 30 sleigh preparations. When count reaches 30, Santa:

- Sets `program_finished` to 1 under `mutex` protection
- Releases all semaphores (`elves_wait` , `elf_queue` , `reindeer_wait` , `reindeer_barrier`).This ensures all waiting threads (both reindeer and elves) can receive the termination signal and exit properly

2.Main Task Management

Santa remains dormant until awakened by santa_sleep semaphore, then checks two conditions:

Reindeer Handling (Priority Task)

```
if(reindeer_num == 9) {
    // Handle reindeer preparation
}
```

- Takes priority over elf requests
- Prepares sleigh when all 9 reindeer have returned
- Uses `reindeer_wait` to coordinate reindeer getting hitched

Implements two crucial synchronization mechanisms:

- `reindeer_done` semaphore ensures all reindeer complete "getting hitched" before Santa can help elves
- `reindeer_barrier` prevents new reindeer from entering the waiting state until all current reindeer have been hitched

Elf Helping (Secondary Task)

```
else if(elf_num == 3 && elves_waiting) {
    // Handle elf assistance
}
```

- Activates when exactly 3 elves are waiting
- Uses `elves_wait` to signal elves they can receive help
- Implements similar synchronization with `elves_done` :

  1.Ensures all three elves receive help before Santa returns to sleep
  2.Maintains proper order of help sessions
  3.Prevents overlapping help sessions with reindeer preparation
- `elf_queue` limits the number of waiting elves to 3

func_elf()

```
void *func_elf() {
    while(!program_finished) {
        sleep(rand() % WORK_TIME);

        sem_wait(&elf_queue);
        sem_wait(&mutex);
```

```
        if(program_finished) {
            sem_post(&mutex);
            sem_post(&elf_queue);
            break;
        }

        if(santa_state == 0 && reindeer_num < 9) {
            elf_num++;
            printf("Elf: waiting (%d)\n", elf_num);

            if(elf_num == 3) {
                elves_waiting = 1;
                printf("Elves call Santa\n");
                sem_post(&santa_sleep);
            }
            sem_post(&mutex);

            sem_wait(&elves_wait);
            if(program_finished) {
                sem_post(&elves_wait);
                break;
            }

            printf("Elf: get help\n");
            sleep(rand() % num);
            sem_post(&elves_done);

            sem_wait(&mutex);
            elf_num--;
            if(elf_num == 0) {
                elves_waiting = 0;
            }
            sem_post(&mutex);
            sem_post(&elf_queue);
        } else {
            sem_post(&mutex);
            sem_post(&elf_queue);
            sleep(rand() % WORK_TIME);
        }
    }
    return NULL;
```

```
}
```

The `func_elf` function uses two key semaphores for synchronization: `elf_queue` controls elves' waiting line to ensure no more than 3 elves can wait simultaneously for Santa's help, while `elves_wait` is used by Santa to signal which elves can receive help. After simulating work time, elves try to join the queue through `elf_queue`. Under `mutex` protection, they increment `elf_num` and when the third elf arrives, they wake Santa. Upon receiving the `elves_wait` signal from Santa, they get help and signal completion through `elves_done`, ensuring Santa knows when the help session is finished.

func_reindeer()

```
void *func_reindeer() {
    while(!program_finished) {
        sem_wait(&reindeer_barrier);

        if(program_finished) {
            sem_post(&reindeer_barrier);
            break;
        }

        sleep(rand() % VACATION_TIME);

        sem_wait(&mutex);
        reindeer_num++;
        printf("Reindeer: waiting (%d)\n", reindeer_num);
        if(reindeer_num == 9) {
            printf("Reindeer 9 calls Santa.\n");
            sem_post(&santa_sleep);
        }
        sem_post(&mutex);

        sem_wait(&reindeer_wait);
        printf("Reindeer: getting hitched\n");
        sem_post(&reindeer_done);
    }
    return NULL;
}
```

The `func_reindeer` function uses the `reindeer_barrier` semaphore to ensure synchronized group behavior,which is metioned in the analysis of `func_santa()`. After simulating vacation time, reindeer enter a waiting state protected by `mutex`, incrementing `reindeer_num` and waking Santa when all nine have returned. Upon receiving the `reindeer_wait` signal from Santa, they proceed with getting hitched and signal completion through `reindeer_done`, ensuring all reindeer complete their hitching process before santa does other things.

Test runs:

To demonstrate the program execution, here is a sample output:

```
(base) root@LAPTOP-G6UNDHO9:/mnt/d/Code# ./santa
Reindeer: waiting (1)
Reindeer: waiting (2)
Reindeer: waiting (3)
Reindeer: waiting (4)
Elf: waiting (1)
Elf: waiting (2)
Elf: waiting (3)
Elves call Santa
Santa: help elves
Elf: get help
Elf: get help
Elf: get help
Reindeer: waiting (5)
Reindeer: waiting (6)
Elf: waiting (1)
Reindeer: waiting (7)
Elf: waiting (2)
Elf: waiting (3)
Elves call Santa
Santa: help elves
Elf: get help
Elf: get help
Elf: get help
Reindeer: waiting (8)
Reindeer: waiting (9)
Reindeer 9 calls Santa.
Santa: preparing sleigh
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Reindeer: getting hitched
Elf: waiting (1)
Elf: waiting (2)
Elf: waiting (3)
Elves call Santa
Santa: help elves
```

As the Santa Claus Problem runs for 30 rounds, we only display a portion of the output here. From these sample outputs, we can observe the proper synchronization between Santa, reindeer, and elves, demonstrating that the program successfully maintains the required coordination and ordering of events throughout its execution.