Project 2 Report

w1049

Contents

1 Stooge Farmers Problem

1.1 Semaphores

I use 4 semaphores to solve this problem, here are the names and the initial value of them:

- `shovel = 1`, one shovel is available
- `filled = MAX`, Larry can dig at most `MAX` holes ahead of Curly
- `unfilled = 0`, the number of unfilled holes, waiting for Moe to plant
- `planted = 0`, the number of planted holes, waiting for Curly to fill

1.2 Design

Larry, Moe and Curly are designed as 3 functions, running in 3 threads. Every function has a infinite loop. When the holes exceed the limit (`endnum = 123`), the thread will exit.

The id of each hole is a local variable `id` in every thread, which increases when an action is done. This means that I limit each thread to take action on the hole with lowest `id`. This simplifies the implementation and the result is still legal.

I make each thread sleep twice, once during the action and once after the action. The first sleep is to simulate the time of digging, planting and filling. The second sleep is to simulate the time of walking to the next hole. If there is only once sleep, the thread will `wait` the semaphore immediately after it `post` one, which makes the output not so like the real world.

2 The Faneuil Hall problem

2.1 Semaphores

I use 5 semaphores to here, 3 of them are more likely to be signals:

- `all_checkin = 0`, a binary semaphore sent by the last immigrant to check in
- `confirm = 0`, sent from judge to immigrant, telling him that he can be confirmed
- `confirmed = 0`, sent from immigrant to judge, telling him that the immigrant has been confirmed

The other 2 are more like mutexes:

- `mutex = 1`, protecting the global variable `checked`
- `judge = 1`, protecting other global variables, and making sure there's only one judge in the hall

2.2 Design

Three kinds of threads are created in an infinite loop according to probability. There's a long interval between thread creation, simulating people coming in sequence.

2.2.1 Check in

First I notice one thing in the constraints: when the judge is in the hall, all 3 kinds of people are not allowed to enter, and only the spectator can leave. I use a `judge` semaphore (as a mutex) to implement this.

When a immigrant trys to check in, he acquire the `mutex`, and increase the `checked`. Then he checks if a judge is in the hall: if so, he will check if he is the last one to check in.

```
sem_wait(&mutex); // acquire the mutex to modify checked
// checkin
++checked;
SLEEP;
printf("Immigrant #%d checkIn\n", id);
if (judge_id >= 0 && checked == entered)
    sem_post(&all_checkin); // wake up judge, and give mutex to him
else sem_post(&mutex); // give mutex to anybody
```

The judge acquires the mutex at the beginning, but if he finds that not all immigrants have checked in, he will give up the mutex. When the code below is executed, the judge will have the mutex again (the last immigrant acquired it but didn't release it).

```
if (entered > checked) { // if not all imms checked in
    sem_post(&mutex); // give them a chance to check in
    sem_wait(&all_checkin); // wait for all imms to check in
}
```

2.2.2 Confirmation

For the confirmation, the general idea is to use a queue to store the immigrant numbers that have sat down, and the judge will read the queue. But it is more complicated to implement. I put the confirmation in the immigrant function instead of the judge function, and record the current judge id in the global variable `judge_id` (there can only be one judge or no judge at the same time, so it works).

After the immigrant sits down, he will waits the `confirm` signal from the judge. On success, he print the confirmation and send a `confirmed` signal to the judge.

```
sem_wait(&confirm); // I need to be confirmed
// confirm
SLEEP;
printf("Judge #%d confirm the immigrant #%d\n", judge_id, id);
sem_post(&confirmed); // I have been confirmed
```

When the judge starts confirmation, it sends some `confirm` signals, then waits for the same number of `confirmed` signals.

```
// confirm
for (int i = 0; i < entered; i++)
    sem_post(&confirm); // tell all imms, you can be confirmed
for (int i = 0; i < entered; i++)
    sem_wait(&confirmed); // after all imms are confirmed
entered = checked = 0;
```

This has the disadvantage that the judge waits for every immigrant to finish confirming before leaving, whether he sits down or not, but the result is legal. In the case of a program with only short delay, there are usually no immigrants who doesn't sit down for a long time, so the result should be consistent with the queue implementation.