Operating System Project 2 Report

Jinru Han 521021910982

Contents

1 Specific approach and implementation:

1.1 Stooge Farmers Problem

Overall implementation approach: In `main`, three threads named Larry, Moe, and Curly are created separately, each of which executes a corresponding program in a loop. The condition for exiting the loop is when the processed hole reaches the maximum value set[1]. The implementation details of different functions are as follows:

Larry To ensure that Larry does not get more than `MAX` holes ahead of Curly, wait for the `filled` semaphore before performing the digging operation. Since the shovel cannot be shared with Curly, wait further for the `shovel` semaphore. After completing the digging operation, release the `shovel` semaphore and the `unfilled` semaphore (see details in 1.1).

The global variable `holesDug` is used to record the ID of the empty hole. The specific code implementation is as follows:

–Larry–

```c
void *Larry(void *arg) {
    do {
        sem_wait(&filled);
        sem_wait(&shovel);
        // Digging
        holesDug++;
        printf("Larry digs another hole #%d\n",holesDug);
        sleep(rand()%3);
        sem_post(&shovel);
        sem_post(&unfilled);
    } while(holesDug<MAX_HOLES);
    pthread_exit(NULL);
}
```

Moe First, check if there are empty holes, which is done using the `unfilled` semaphore. Wait for the `unfilled` semaphore, perform the "plant seed" operation, and release the `unfilled_with_seed` semaphore after completion (see details in 1.1).

The global variable `empty_holes` is used to record the IDs of holes with seeds. The specific code implementation is as follows:

[1] Set as 100 in the submitted program

—Moe—

```c
void *Moe(void *arg) {
    do {
        sem_wait(&unfilled);
        // Seeding
        empty_holes++;
        printf("Moe plants a seed in a hole #%d\n",empty_holes);
        sleep(rand()%3);
        sem_post(&unfilled_with_seed);
    } while(empty_holes<MAX_HOLES);
    pthread_exit(NULL);
}
```

Curly First, check if there are holes with seeds, which is done using the `unfilled_with_seed` semaphore. Wait for the `unfilled_with_seed` semaphore first, and since the shovel cannot be shared with Larry, wait for the `shovel` semaphore. After performing the "fill hole" operation, release the `shovel` and `filled` semaphores so that Larry can continue to dig holes.

The global variable `holes_unfilled` is used to record the IDs of unfilled holes with seeds. The specific code implementation is as follows:

—Curly—

```c
void *Curly(void *arg) {
    do {
        sem_wait(&unfilled_with_seed);
        sem_wait(&shovel);
        // Filling
        holes_unfilled++;
        printf("Curly fills a planted hole #%d\n",holes_unfilled);
        sleep(rand()%3);
        sem_post(&shovel);
        sem_post(&filled);
    } while(holes_unfilled<MAX_HOLES);
    pthread_exit(NULL);
}
```

1.2 Faneuil Hall Problem

Overall implementation approach: There is one judge and multiple immigrants and specters, so the judge process is created to execute the `judge` function in a loop. In the `main` function, new processes are created in a loop to execute the `immigrant` and `spector` functions, respectively. The creation of immigrant and specter threads ends when the total number of immigrants reaches the maximum value set[2]. In each process, individuals of different identities need to wait for the `mutex` semaphore to avoid starvation. The implementation details of each function are as follows:

immigrant The process ID is used as the ID for immigrants. When entering the hall, they need to wait for the judge to leave and the `mutex` semaphore. The number of immigrants entering and checked in are recorded in two `int` variables, `immigrant_enter_cnt` and `immigrant_ckn_cnt`, respectively, which serve as the condition for the judge to confirm (1.2). The `confirmed` semaphore is used to wait for the judge's confirmation, and after the judge leaves, immigrants can leave. The specific implementation code is as follows:

–immigrant–

```c
void *immigrants(void *args){
    pthread_detach(pthread_self());
    int id = *(int *)args;
    // immigrant wait to enter if judge is in the building
    while (judge_in);   //wait for judge to leave
    sem_wait(&mutex);   //avoid starvation
    // enter
    sem_wait(&immigrant_enter); //protect "immigrant_enter_cnt"
    immigrant_enter_cnt++;
    while (judge_in);
    printf("Immigrant #%d enters.\n", id);
    sleep(rand()%3);
    sem_post(&immigrant_enter);
    sem_post(&mutex);
    // checkin and sitdown
    sem_wait(&immigrant_ckn);   //protect "immigrant_ckn_cnt"
    immigrant_ckn_cnt++;
    printf("Immigrant #%d checkIn.\n", id);
    printf("Immigrant #%d sitDown.\n", id);
```

[2] In the submitted program, it is set to `MAX_IMMIGRANT=10`

```
    sleep(rand()%3);
    sem_post(&immigrant_ckn);
    while (!judge_in);  //wait for judge to enter
    sem_wait(&confirmed);   //wait for judge to confirm
    sem_post(&confirmed);
    //Immigrants wait to leave if judge is in the building
    while (judge_in);
    // get certificate and leave
    printf("Immigrant #%d getCertificate.\n", id);
    printf("Immigrant #%d leaves.\n", id);
    sleep(rand()%3);
    pthread_exit(NULL);
}
```

judge The judge's enter operation only needs to wait for the `mutex` semaphore. The judge's enter and leave states both need to modify the global `bool` variable `judge_in` to provide judgment conditions for immigrant and spector operations.

The condition for judge to confirm is `immigrant_enter_cnt == immigrant_ckn_cnt`, as mentioned in section 1.2. After confirmation, the judge can leave.

Use the global variable `judge_enter_times` to record the number of times the judge has entered.

The end loop condition for the judge process is that the number of confirmed immigrants is less than the total number of checked-in immigrants. The specific implementation code is as follows:

–Judge–

```
void *judge(){
    do {
        sleep(rand()%3+3);
        // judge enter
        sem_wait(&mutex);   // avoid starvation
        judge_enter_times++;
        printf("Judge #%d enters.\n", judge_enter_times);
        judge_in = true;
        sleep(rand()%3);
        sem_post(&mutex);
        // judge wait to confirm if not all entering immigrants have checked in
        while (immigrant_enter_cnt != immigrant_ckn_cnt);
```

```
        sem_wait(&confirmed);
        // judge confirm
        for (int i = last; i <= immigrant_ckn_cnt ; ++i) {
            printf("Judge confrim the immigrant #%d.\n", i);
            sleep(rand()%3);
        }
        last = immigrant_ckn_cnt + 1;
        sem_post(&confirmed);
        // judge leave
        printf("Judge #%d leaves.\n", judge_enter_times);
        sleep(rand()%3);
        judge_in = false;
    } while ((last-1)<MAX_IMMIGRANT);// while there are still immigrants to be confirmed
    pthread_exit(NULL);
}
```

spector Similarly, the process id is used as the id for each spector. The entry condition for a spector is the same as that for an immigrant, that is, it needs to wait for the judge to leave and the mutex semaphore. After entering, the spector can spectate and leave unconditionally. The specific implementation code is as follows:

–Spector–

```
void *spector(void* args){
    pthread_detach(pthread_self());
    int id = *(int *)args;
    // spector wait to enter if judge is in the building
    while (judge_in);
    sem_wait(&mutex);// avoid starvation
    while (judge_in);
    // spector enter
    printf("Spector #%d enters.\n",id);
    sem_post(&mutex);
    // spector spectate
    printf("Spector #%d spectates.\n", id);
    sleep(rand()%3);
    // spector leaves
    printf("Spector #%d leaves.\n", id);
    sleep(rand()%3);
```

```
    pthread_exit(NULL);
}
```

2 Problems encountered

1. Simulating multiple immigrants in the Faneuil Problem. Since it is necessary to simulate multiple immigrants operating simultaneously, multiple threads need to be created instead of executing in a loop as in the first problem. Initially, I thought that the pointer to the first `pthread_t` variable created by the `pthread_create` function had to point to the same process until it ended. After consulting with the teacher, I understood that it was possible to create multiple threads using a while loop without overwriting the previously created threads, but the thread pointed to by the pointer of the first `pthread_t` variable changes.
   However, this method was unable to save the pointers of the child threads, so `pthread_join` cannot be used in the parent process to wait for each thread to end, resulting in resource leaks. Therefore, `pthread_detach (pthread_self())` is used to set the child thread as a detached state at the beginning of each thread, allowing the system to automatically recycle the resources of the child thread. Finally, `pthread_exit` is used to exit the thread and release resources, without the need for the parent thread to explicitly call `pthread_join()`.

2. Avoid running multiple identical threads simultaneously The original program did not use the `sleep` function when creating threads, resulting in a large number of `immigrant` functions being executed at once, while the `judge` and `inspector` functions were waiting. After being reminded by the teacher in class, the `sleep` function was added when creating threads, allowing each thread to execute in an interleaved manner, simulating people of different identities entering the hall in an interleaved manner.

3 Acknowledgments

Thank you to Assistant Researcher Qinya Li for her patient guidance and detailed answers to specific questions. Thank you to the teaching assistants for their help and assistance.