Operating System Project 2

周旭东 521021910829

2023 April

摘要

在操作系统课程设计 Project2 中，通过两个任务学习解决进程并发执行过程中的进程间通信问题，使用 Pthread 和 semaphore，并尝试阻止 starvation 的发生。

目录

1 Task1: Stooge Farmers Problem

1.1 问题描述

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.

1.2 算法

1.2.1 Larry 部分算法

Larry 需要等待 Curly 释放的 unfilled（避免饥饿）和唯一的 shovel 后开始工作，并在工作后释放了一个空的洞穴 empty

```
1 void *larry(){
2     while(dignum <= process_need){
3         sem_wait(&unfilled);
4         sem_wait(&shovel);
5         //dig
6         sem_post(&shovel);
7         sem_post(&empty);
8     }
9 }
```

1.2.2 Moe 部分算法

Moe 需要等待 Larry 释放的 empty 后开始 seed，并在 seed 后释放了一个 seeded.

```
1 void *moe(){
2     while(seednum <= process_need){
3         sem_wait(&empty);
4         printf("Moe plants a seed in a hole #%i.\n",seednum++);
5         sem_post(&seeded);
6     }
```

```
}
```

1.2.3 Curly 部分算法

Curly 需要等待 Moe 释放的 seeded 后开始 fill，并在 fill 后释放了一个 unfilled.

```
void *curly(){
    while(fillnum <= process_need){
        sem_wait(&seeded);
        sem_wait(&shovel);
        printf("Curly fills a planted hole #%i.\n",fillnum++);
        sem_post(&shovel);
        sem_post(&unfilled);
    }
}
```

unfilled 有上限 MAX，在初始阶段将 Max 将其赋值

```
        sem_init(&unfilled, 0, MAX);
```

1.2.4 starvation 应对策略

采用循环 semaphore 策略保证公平性，避免饥饿: Larry wait(unfilled) and signal(empty); Moe wait(empty) and signal(seeded); Curly wait(seeded) and signal(unfilled)

```
        void *larry(){
                sem_wait(&unfilled);
                ...
                sem_post(&empty);
        }
        void *moe(){
                sem_wait(&empty);
                ...
                sem_post(&seeded);
        }
        void *curly(){
                sem_wait(&seeded);
                ...
                sem_post(&unfilled);
        }
```

1.3 结果展示

执行 LCM ，指定 MAX=20，运行 500 次规模得到如图 1 输出

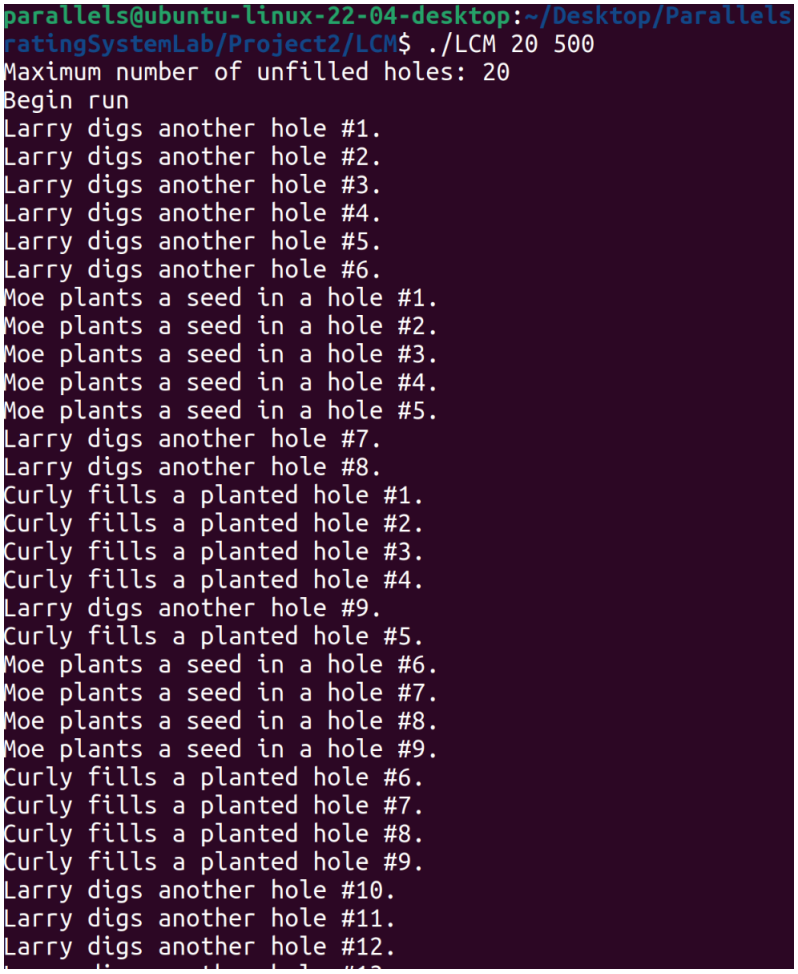

图 1: An example of LCM solution

2 Task2: The Faneuil Hall problem

2.1 问题描述

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.

2.2 算法

2.2.1 immigrant 部分算法

移民进入时要看法官是否在房间里。进入后，移民 chek in 并 sit down，这些操作需要 mutex 进行保护。如果法官在等待，最后一个移民释放 allsigned，否则释放 mutex 供其余使用。

此后需要等待法官的 confirm，倘若获得 certification，且法官离开，则移民可离开。

```
1  void* immigrant(){
2      while (1)
3      {
4          sem_wait(&nojudgeIN);// before judge to get in
5          immigrantcount++;
6          sleep(rand()%3+1);
7          printf("Immigrant #%d enter\n", immigrantcount);
8          entered++;
9          sem_post(&nojudgeIN);
10
11         sem_wait(&mutex);// wait for mutex to check
12         sleep(rand()%3+1);
13         printf("Immigrant #%d checkIn\n", immigrantcount);
14         // sitdown
15         sleep(rand()%3+1);
16         printf("Immigrant #%d sitDown\n", immigrantcount);
17         checked++;
18         // if all ready, signal allsigned, else release mutex
19         if(judgein == 1 && entered == checked){
20             sem_post(&allsigned);
21         }
22         else{
```

```
23              sem_post(&mutex);
24          }
25
26          sem_wait(&confirmed);// wait to be confirmed
27          sleep(rand()%3+1);
28          printf("Immigrant #%d getCertificate\n", immigrantcount);
29
30          sem_wait(&nojudgeIN);// wait for judge to get out
31          sleep(rand()%3+1);
32          printf("Immigrant #%d leave\n", immigrantcount);
33          sem_post(&nojudgeIN);
34      }
35  }
```

2.2.2 judge 部分算法

法官需要等待 nojudgeIN，以防止两位法官同时开庭。进入后持有 nojudgeIN 以禁止移民和旁观者进入，并持有 mutex 以保证他的进入。

如果法官到的时候，所有进入的人都 check in 了，她就可以立即进行。否则，释放 mutex 以允许 check in。当最后一个移民 check in 并发出 allsigned 信号时，法官持回 mutex。

confirm 后，法官为签到的移民发出 confirmed 信号。然后法官 leave 并释放 mutex 和 nojudgeIN。此后移民和旁观者可自由进出。

```
1   void* judge(){
2       while (1)
3       {
4           sem_wait(&nojudgeIN);// prevent starvation
5           sem_wait(&mutex);
6           judgein = 1;
7           judgecount++;
8           sleep(rand()%3+1);
9           printf("Judge #%d enter\n", judgecount);
10          if(entered > checked){
11              sem_post(&mutex);
12              sem_wait(&allsigned);// all checked, began confirmation
13          }
14
15          if(checked != 0){
16              sleep(rand()%3+1);
17              printf("Judge #%d confirm the immigrant #%d\n",
                    judgecount, immigrantcount);
18              checked--;
```

```
19              entered--;
20              sem_post(&confirmed);
21          }
22          sleep(rand()%3+1);
23          printf("Judge #%d leave\n", judgecount);
24          judgein = 0;
25
26          sem_post(&mutex);
27          sem_post(&nojudgeIN);
28      }
29  }
```

2.2.3 spectator 部分算法

旁观者较为自由，只受 nojudgeIN 控制，当没有法官在场，可进出。

```
1   void* spectator(){
2       while (1)
3       {
4           sem_wait(&nojudgeIN);
5           spectatorcount++;
6           sleep(rand()%3+1);
7           printf("Spectator #%d enter\n", spectatorcount);
8           sem_post(&nojudgeIN);
9
10          sleep(rand()%3+1);
11          printf("Spectator #%d spectate\n", spectatorcount);
12          sleep(rand()%3+1);
13          printf("Spectator #%d leave\n", spectatorcount);
14      }
15
16  }
```

2.2.4 starvation 应对策略

在每部分开始前都需争夺 nojudgeIN，从而保证公平性，避免饥饿。

```
1   void* immigrant(){
2           sem_wait(&nojudgeIN);
3           ...
4   }
5   void* judge(){
6           sem_wait(&nojudgeIN);
```

```
7           ...
8   }
9   void* spectator(){
10          sem_wait(&nojudgeIN);
11          ...
12  }
```

2.2.5 时间仿真

采用 sleep(rand()); 在 1-4s 时间内进行随机等待

```
1           sleep(rand()%3+1);
```

2.3 结果展示

执行 faneuil ， 经过一段时间后得到如图 2 输出

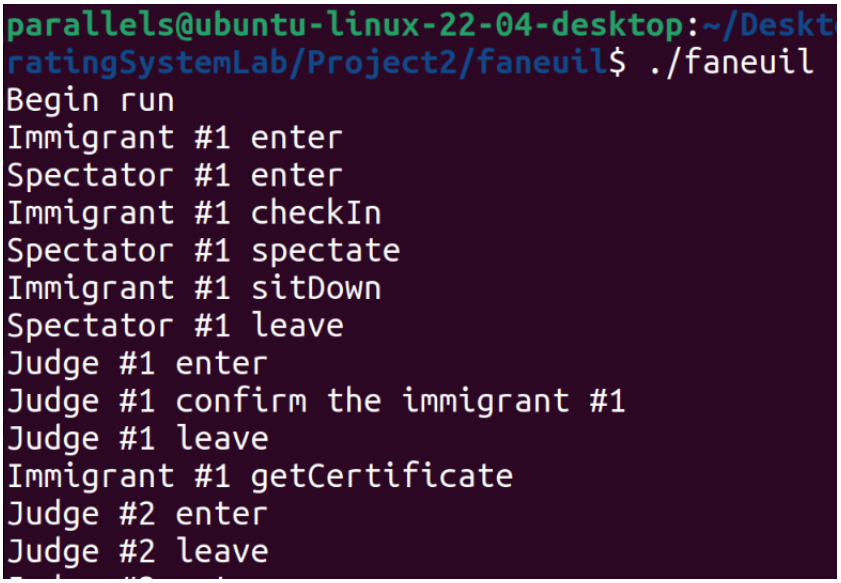

图 2: An example of faneuil solution

3 Code

3.1 LCM

```
1           #include <stdio.h>
2   #include <unistd.h>
```

```
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <semaphore.h>
6
7  int process_need = 100;//the num of seeds we have
8
9  //semaphore
10 sem_t shovel;
11 sem_t empty;
12 sem_t seeded;
13 sem_t unfilled;
14 sem_t mutex;//prevent starvation
15
16 //counter
17 int MAX;
18 int dignum = 1;
19 int seednum = 1;
20 int fillnum = 1;
21
22 void *larry(){
23     while(dignum <= process_need){
24         sem_wait(&unfilled);
25         sem_wait(&mutex);
26         sem_wait(&shovel);
27         printf("Larry digs another hole #%i.\n",dignum++);
28         sem_post(&shovel);
29         sem_post(&mutex);
30         sem_post(&empty);
31     }
32 }
33 void *moe(){
34     while(seednum <= process_need){
35         sem_wait(&empty);
36         sem_wait(&mutex);
37         printf("Moe plants a seed in a hole #%i.\n",seednum++);
38         sem_post(&mutex);
39         sem_post(&seeded);
40     }
41 }
42 void *curly(){
43     while(fillnum <= process_need){
44         sem_wait(&seeded);
```

```
45          sem_wait(&mutex);
46          sem_wait(&shovel);
47          printf("Curly fills a planted hole #%i.\n",fillnum++);
48          sem_post(&shovel);
49          sem_post(&mutex);
50          sem_post(&unfilled);
51      }
52  }
53
54  int main(int argc, char *argv[]){
55      if(argc < 2){//parameters detection
56          printf("Command Error\nCommand:./LCM <num_of_unfilled_holes
                > <(num_of_max_seeds)>\n");
57          return -1;
58      }
59      MAX = atoi(argv[1]);
60      process_need = atoi(argv[2]);
61      printf("Maximum number of unfilled holes: %d\n",MAX);
62
63      pthread_t ltid;//Larry
64      pthread_t mtid;//Moe
65      pthread_t ctid;//Curly
66
67      //initializing the semaphores
68      sem_init(&shovel, 0, 1);
69      sem_init(&empty, 0, 0);
70      sem_init(&seeded, 0, 0);
71      sem_init(&unfilled, 0, MAX);
72      sem_init(&mutex, 0, 1);
73
74      printf("Begin run\n");
75      pthread_create(&ltid, NULL, larry, NULL);  //create the larry
            thread
76      pthread_create(&mtid, NULL, moe, NULL);  //create the moe
            thread
77      pthread_create(&ctid, NULL, curly, NULL);  //create the curly
            thread
78
79      pthread_join(ltid,NULL);
80      pthread_join(mtid,NULL);
81      pthread_join(ctid,NULL);
82
```

```
83      sem_destroy(&shovel);
84      sem_destroy(&empty);
85      sem_destroy(&seeded);
86      sem_destroy(&unfilled);
87      sem_destroy(&mutex);
88
89      printf("End run\n");
90      return 0;
91  }
```

3.2 faneuil

```
1           #include <stdio.h>
2   #include <unistd.h>
3   #include <stdlib.h>
4   #include <pthread.h>
5   #include <semaphore.h>
6
7   // semaphore
8   sem_t noJudge;
9   sem_t confirmed;
10  sem_t allSignedIn;
11  sem_t mutex;
12
13  // counter
14  int judgein = 0;
15  int immigrantcount = 0;
16  int judgecount = 0;
17  int spectatorcount = 0;
18
19  int entered = 0;
20  int checked = 0;
21
22  void* immigrant(){
23      while (1)
24      {
25          sem_wait(&noJudge);// before judge to get in
26          immigrantcount++;
27          sleep(rand()%3+1);
28          printf("Immigrant #%d enter\n", immigrantcount);
29          entered++;
```

```
30          sem_post(&noJudge);
31
32          sem_wait(&mutex);// wait for mutex to check
33          sleep(rand()%3+1);
34          printf("Immigrant #%d checkIn\n", immigrantcount);
35          // sitdown
36          sleep(rand()%3+1);
37          printf("Immigrant #%d sitDown\n", immigrantcount);
38          checked++;
39          // if all ready, signal allSignedIn, else release mutex
40          if(judgein == 1 && entered == checked){
41              sem_post(&allSignedIn);
42          }
43          else{
44              sem_post(&mutex);
45          }
46
47          sem_wait(&confirmed);// wait to be confirmed
48          sleep(rand()%3+1);
49          printf("Immigrant #%d getCertificate\n", immigrantcount);
50
51          sem_wait(&noJudge);// wait for judge to get out
52          sleep(rand()%3+1);
53          printf("Immigrant #%d leave\n", immigrantcount);
54          sem_post(&noJudge);
55      }
56  }
57
58  void* judge(){
59      while (1)
60      {
61          sem_wait(&noJudge);// prevent starvation
62          sem_wait(&mutex);
63          judgein = 1;
64          judgecount++;
65          sleep(rand()%3+1);
66          printf("Judge #%d enter\n", judgecount);
67          if(entered > checked){
68              sem_post(&mutex);
69              sem_wait(&allSignedIn);// all checked, began
                    confirmation
70          }
```

```
71
72          if(checked != 0){
73              sleep(rand()%3+1);
74              printf("Judge #%d confirm the immigrant #%d\n",
                    judgecount, immigrantcount);
75              checked--;
76              entered--;
77              sem_post(&confirmed);
78          }
79          sleep(rand()%3+1);
80          printf("Judge #%d leave\n", judgecount);
81          judgein = 0;
82
83          sem_post(&mutex);
84          sem_post(&noJudge);
85      }
86  }
87
88  void* spectator(){
89      while (1)
90      {
91          sem_wait(&noJudge);
92          spectatorcount++;
93          sleep(rand()%3+1);
94          printf("Spectator #%d enter\n", spectatorcount);
95          sem_post(&noJudge);
96
97          sleep(rand()%3+1);
98          printf("Spectator #%d spectate\n", spectatorcount);
99          sleep(rand()%3+1);
100         printf("Spectator #%d leave\n", spectatorcount);
101     }
102
103 }
104
105 int main(int argc, char *argv[]){
106     pthread_t itid;// immigrant
107     pthread_t jtid;// judge
108     pthread_t stid;// spectator
109
110     // nitializing the semaphores
111     sem_init(&noJudge, 0, 1);
```

```
112     sem_init(&confirmed, 0, 0);
113     sem_init(&allSignedIn, 0, 0);
114     sem_init(&mutex, 0, 1);
115
116     printf("Begin run\n");
117     pthread_create(&itid, NULL, immigrant, NULL);
118     pthread_create(&jtid, NULL, judge, NULL);
119     pthread_create(&stid, NULL, spectator, NULL);
120
121     pthread_join(itid,NULL);
122     pthread_join(jtid,NULL);
123     pthread_join(stid,NULL);
124
125     sem_destroy(&noJudge);
126     sem_destroy(&confirmed);
127     sem_destroy(&allSignedIn);
128     sem_destroy(&mutex);
129
130     printf("End run\n");
131     return 0;
132 }
```