

华北水利水电大学

North China University of Water Resources and Electric Power

《操作系统》实验报告

实验 四 进程的同步与互斥

院 系 信息工程学院

专 业 人工智能

学 号 202018526

姓 名 高树林

指 导 教 师 杨学颖

完成时间: 2022-12-23

实验目的

了解和掌握进程的同步与互斥关系。

实验要求

1. 程序 4-1.c 模拟了 1 个生产者和 1 个消费者，请改写该程序，模拟 5 个生产者和 5 个消费者，它们共享一个包含 8 个缓冲区的缓冲池。产品以 4 位编号，最高位表示生产者编号、其他表示该生产者的产品号，
2. 补充完成程序 4-2.c，实现如下功能：假设有三个并发进程（P、Q、R），其中 P 负责从输入设备上读取信息并传给 Q，Q 将信息加工后传给 R，R 负责将信息打印输出。写出符合下列条件下的并发程序：进程 P、Q 共享一个由 5 个缓冲区组成的缓冲池，进程 Q、R 共享另一个由 8 个缓冲区组成的缓冲池。
3. 实现如下功能：假设理发店由等待间（2 个座位）和理发间（只有一个座位）构成，5 位顾客先后进入等待间，再进入理发间。试写出模拟理发师和顾客的程序，要求为每一个顾客创建一个线程。

实验分析

分析 1：生产者消费者问题之间对于缓冲区的访问是一个互斥关系。如果生产者对缓冲区的写入操作不互斥，则可能造成进程 p1 和进程 p2 同时获得缓冲区的某个位置为空，进程 p1 要往缓冲区的该位置写入数据，同时 p2 也要往该位置写入数据，导致两个进程往其中写入数据的时候出现了数据覆盖的现象。（可能是进程 p1 先写入数据之后进程 p2 又立即往相应的位置写入了数据）如果消费者读取数据的时候没有互斥，则可能造成两个消费者进程同时读取某一段数据，导致各自只读取了数据的一部分，造成读取错误。生产者与消费者又是互相协作的一个关系，必须有生产者的生产之后才有消费者的消费，因此生产者与消费者之间又是一个同步的关系。

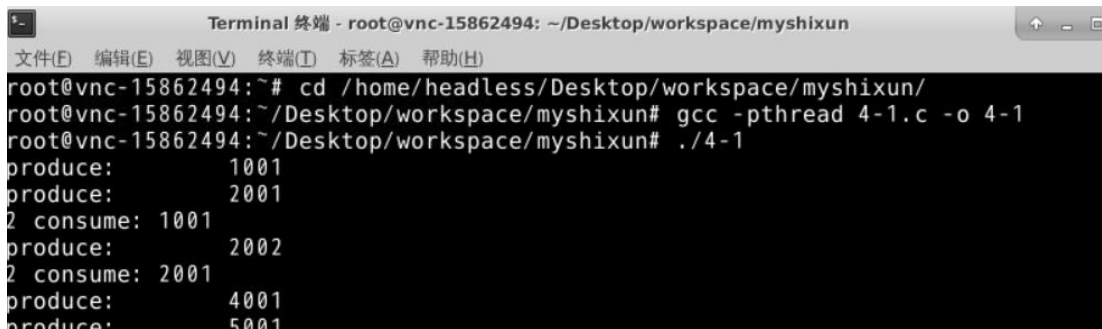
分析 2：三个并发问题，其实还是生产者消费者问题的一类表现。其中一个进程是另一个进程的生产者，另外一个进程又是该进程的生产者。逐步传递的关系实现了 r 进程消费由 q 进程传递来的资源，q 进程消费由 p 进程传递来的资源。因此 q 进程需要等待 p 进程的资源，r 进程需要等待 q 的资源，是两组单生成者

单消费者的组合。

分析 3：一个给顾客信号量，一个理发师信号量（看他自己是不是闲着），第三个是互斥信号量（Mutual exclusion，缩写成 mutex）。一位顾客来了，他想拿到互斥信号量，他就等着直到拿到为止。顾客拿到互斥信号量后，会去查看是否有空着的椅子（可能是等候的椅子，也可能是理发时坐的那张椅子）。如果没有一张是空着的，他就走了。如果他找到了一张椅子，就会让空椅子的数量减少一张，这位顾客接下来就使用自己的信号量叫醒理发师。这样，互斥信号标就释放出来供其他顾客或理发师使用。如果理发师在忙，这位顾客就会等。理发师就会进入了一个永久的等候循环，等着被在等候的顾客唤醒。一旦他醒过来，他会给所有在等候的顾客发信号，让他们依次理发。

实验结果

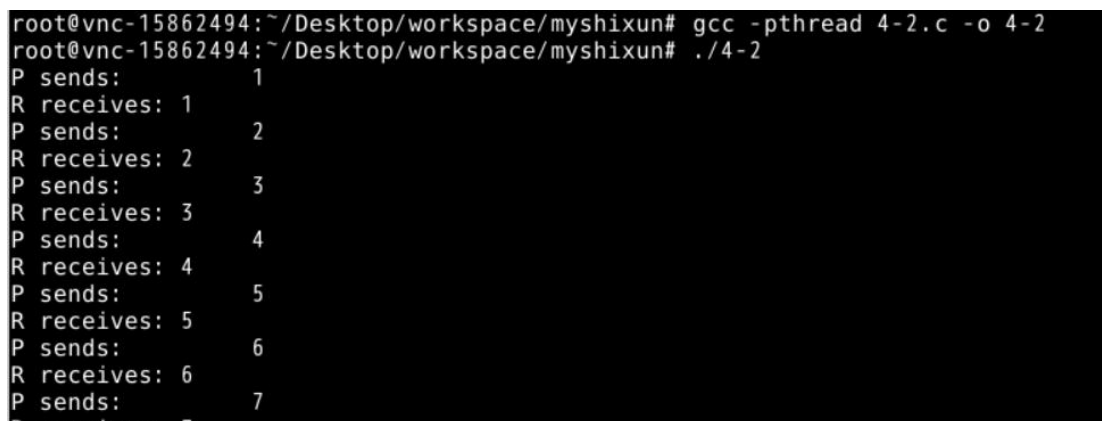
结果 1：生产者消费者问题的结果如下图 1 所示。

A terminal window titled 'Terminal 终端 - root@vnc-15862494: ~/Desktop/workspace/myshixun' showing the execution of a producer-consumer program. The user runs 'cd /home/headless/Desktop/workspace/myshixun/', 'gcc -pthread 4-1.c -o 4-1', and './4-1'. The output shows alternating 'produce' and 'consume' actions with increasing counts: produce: 1001, consume: 2001, produce: 2001, consume: 1001, produce: 2002, consume: 2001, produce: 4001, consume: 5001.

```
root@vnc-15862494:~# cd /home/headless/Desktop/workspace/myshixun/
root@vnc-15862494:~/Desktop/workspace/myshixun# gcc -pthread 4-1.c -o 4-1
root@vnc-15862494:~/Desktop/workspace/myshixun# ./4-1
produce:      1001
consume:    2001
produce:      2001
consume:    1001
produce:      2002
consume:    2001
produce:      4001
consume:    5001
```

图 1 生产接消费者结果

结果 2：三个并发进程的结果如下图 2 所示。

A terminal window showing the execution of a program with three concurrent processes. The user runs 'gcc -pthread 4-2.c -o 4-2' and './4-2'. The output shows alternating 'P sends' and 'R receives' actions with counts from 1 to 7.

```
root@vnc-15862494:~/Desktop/workspace/myshixun# gcc -pthread 4-2.c -o 4-2
root@vnc-15862494:~/Desktop/workspace/myshixun# ./4-2
P sends:      1
R receives:   1
P sends:      2
R receives:   2
P sends:      3
R receives:   3
P sends:      4
R receives:   4
P sends:      5
R receives:   5
P sends:      6
R receives:   6
P sends:      7
```

图 2 三个并发进程的结果

结果 3：理发师问题的结果如下图 3 所示。

```
P sends:      20
R receives: 20
root@vnc-15862494:~/Desktop/workspace/myshixun# gcc -pthread 4-3.c -o 4-3
root@vnc-15862494:~/Desktop/workspace/myshixun# ./4-3
customer 3: enter waiting-room
customer 3: sit down
customer 3: enter cutting-room and sit down
barber: start cutting
customer 2: enter waiting-room
customer 2: sit down
customer 1: enter waiting-room
customer 1: sit down
customer 4: enter waiting-room
customer 5: enter waiting-room
barber: finish cutting
customer 3: bye
customer 2: enter cutting-room and sit down
barber: start cutting
customer 4: sit down
barber: finish cutting
```

图 3 理发师问题结果

实验总结

通过本次实验，我复习了一遍生产者与消费者的问题，正值期末复习，这个实验让我弄懂了进程的同步与互斥的关系，省去了我单独复习的大把时间，可谓是幸运至极。说到底，所谓进程同步，指的就是异步环境下的一组并发进程因直接制约而互相发送消息、进行互相合作、互相等待，使得各进程按一定的速度执行的过程称为进程间的同步；而所谓的互斥，则是两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。也就是说，发生进程互斥是，一个进程正在访问临界资源，另一个要访问该资源的进程必须等待。其实这些问题再之前的大作业中都做过，这次只是加深了记忆，通过多次的反复练习，我发现对于这类问题我能更加快速解决，实实在在地体验到了“熟能生巧”的魅力所在。

代码:

生产者消费者问题:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #include <unistd.h>
5. #include <pthread.h>
6. #include <semaphore.h>
7. #define N 8
8. #define PRODUCT_NUM 10
9. int buffer[N], readpos = 0, writepos = 0;
10. sem_t full, empty, mutex_p, mutex_c;
11. void sleep_random(int t) {
12.     sleep((int)(t * (rand() / (RAND_MAX * 1.0))));
13. }
14. void *produce(void *id){
15.     int i;
16.     for (i = 0; i < PRODUCT_NUM; i++){
17.         sleep_random(2);
18.         sem_wait(&empty);
19.         sem_wait(&mutex_p);
20.         printf("produce: %d\n", 1000 * (*(int*)id) + i + 1);
21.         buffer[writepos] = 1000 * (*(int*)id) + i + 1;
22.         writepos++;
23.         if (writepos >= N)
24.             writepos = 0;
25.         sem_post(&mutex_p);
26.         sem_post(&full);
27.     }
28. }
29. void *consume(void *id){
30.     int i;
31.     for (i = 0; i < PRODUCT_NUM; i++){
32.         sleep_random(2);
33.         sem_wait(&full);
34.         sem_wait(&mutex_c);
35.         printf("%d consume: %d\n", *(int*)id, buffer[readpos]);
36.         buffer[readpos++] = - 1;
37.         if (readpos >= N)
38.             readpos = 0;
39.         sem_post(&mutex_c);
40.         sem_post(&empty);
41.     }
42. }
```

```
43. #define M 5
44. int main(){
45.     int res, i, id[M];
46.     pthread_t tp[M], tc[M];
47.     void *thread_result;
48.     for (i = 0; i < M; i++)
49.         id[i] = i+1;
50.     for (i = 0; i < N; i++)
51.         buffer[i] = - 1;
52.     srand((int)time(0));
53.     sem_init(&full, 0, 0);
54.     sem_init(&empty, 0, N);
55.     sem_init(&mutex_p, 0, 1);
56.     sem_init(&mutex_c, 0, 1);
57.     for (i = 0; i < M; i++) {
58.         res = pthread_create(&tp[i], NULL, produce, &id[i]);
59.         if (res != 0){
60.             perror("failed to create thread");
61.             exit(1);
62.         }
63.         res = pthread_create(&tc[i], NULL, consume, &id[i]);
64.         if (res != 0){
65.             perror("failed to create thread");
66.             exit(1);
67.         }
68.     }
69.     for (i = 0; i < M; i++) {
70.         res = pthread_join(tp[i], &thread_result);
71.         if (res != 0){
72.             perror("failed to join thread");
73.             exit(2);
74.         }
75.         res = pthread_join(tc[i], &thread_result);
76.         if (res != 0){
77.             perror("failed to join thread");
78.             exit(2);
79.         }
80.     }
```

三个进程同步问题:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #include <unistd.h>
5. #include <pthread.h>
6. #include <semaphore.h>
7. #define LIMIT 20
8. #define M 5
9. #define N 8
10. int buffer1[M], readpos1 = 0, writepos1 = 0;
11. int buffer2[N], readpos2 = 0, writepos2 = 0;
12. sem_t empty1, empty2, full1, full2;
13. void sleep_random(int t) {
14.     sleep((int)(t * (rand() / (RAND_MAX * 1.0))));
15. }
16. void *P(){
17.     int i;
18.     for (i = 0; i < LIMIT; i++){
19.         sleep_random(2);
20.         sem_wait(&empty1);
21.         printf("P sends:      %d\n", i+1);
22.         buffer1[writepos1++] = i + 1; /* put to buffer1 */
23.         if (writepos1 >= M)
24.             writepos1 = 0;
25.         sem_post(&full1);
26.     }
27. }
28. void *Q(){
29.     int i, data;
30.     for (i = 0; i < LIMIT; i++){
31.         sleep_random(2);
32.         sem_wait(&full1);
33.         data = buffer1[readpos1]; /* get from buffer1 */
34.         buffer1[readpos1++] = - 1;
35.         if (readpos1 >= M)
36.             readpos1 = 0;
37.         sem_post(&empty1);
38.         sem_wait(&empty2);
39.         buffer2[writepos2++] = data; /* put to buffer2 */
40.         if (writepos2 >= N)
41.             writepos2 = 0;
42.         sem_post(&full2);
43.     }
```

```
44. }
45. void *R(){
46.     int i;
47.     for (i = 0; i < LIMIT; i++){
48.         sleep_random(2);
49.         sem_wait(&full12);
50.         printf("R receives: %d\n", buffer2[readpos2]); /* get from buffer2 *
           /
51.         buffer2[readpos2++] = - 1;
52.         if (readpos2 >= N)
53.             readpos2 = 0;
54.         sem_post(&empty2);
55.     }
56. }
57. int main(){
58.     int i;
59.     pthread_t t1, t2;
60.     for (i = 0; i < M; i++)
61.         buffer1[i] = - 1;
62.     for (i = 0; i < N; i++)
63.         buffer2[i] = - 1;
64.     srand((int)time(0));
65.     sem_init(&empty1, 0, 1);
66.     sem_init(&empty2, 0, 1);
67.     sem_init(&full11, 0, 0);
68.     sem_init(&full12, 0, 0);
69.     pthread_create(&t1, NULL, P, NULL);
70.     pthread_create(&t2, NULL, Q, NULL);
71.     R();
72.     return 0;
73. }
```


理发师问题:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #include <unistd.h>
5. #include <pthread.h>
6. #include <semaphore.h>
7. #define SEAT_NUM 2
8. #define CUSTOMER_NUM 5
9. sem_t start, finish, cutting_seat_empty, empty_waiting_seats;
10. void sleep_random(int t) {
11.
12.     sleep((int)(t * (rand() / (RAND_MAX * 1.0))));
13. }
14. void *barber()
15. {
16.
17.     while(1)
18.     {
19.
20.         sem_wait(&start);
21.         printf("barber: start cutting\n");
22.         sleep_random(3);
23.         printf("barber: finish cutting\n");
24.         sem_post(&finish);
25.         sleep_random(2);
26.     }
27. }
28. void *customer(void *id)
29. {
30.
31.     const int myid = *(int*)id;
32.     sleep_random(2);
33.     printf("customer %d: enter waiting-room\n", myid);
34.     sem_wait(&empty_waiting_seats);
35.     printf("customer %d: sit down\n", myid);
36.     sem_wait(&cutting_seat_empty);
37.     printf("customer %d: enter cutting-room and sit down\n", myid);
38.     sem_post(&empty_waiting_seats);
39.     sem_post(&start);
40.     sem_wait(&finish);
41.     printf("customer %d: bye\n", myid);
42.     sem_post(&cutting_seat_empty);
43. }
```

```
44. int main()
45. {
46.
47.     int i, id[CUSTOMER_NUM];
48.     pthread_t t[CUSTOMER_NUM];
49.     srand((int)time(0));
50.     sem_init(&cutting_seat_empty, 0, 1);
51.     sem_init(&empty_waiting_seats, 0, SEAT_NUM);
52.     sem_init(&start, 0, 0);
53.     sem_init(&finish, 0, 0);
54.     for (i = 0; i < CUSTOMER_NUM; i++)
55.     {
56.
57.         id[i] = i + 1;
58.         pthread_create(&t[i], NULL, customer, &id[i]);
59.     }
60.     barber();
61.     return 0;
62. }
```