



SHANDONG UNIVERSITY

操作系统第 3 次实验报告

网络空间安全学院 (研究院)

2021 网安三班

谢子洋 202100460116

2023 年 5 月 11 日

目录

1	实验内容	2
1.1	任务 1: 信号量实现生产者-消费者问题解决算法	2
1.1.1	算法	2
1.1.2	实现	3
1.1.3	结果	4
1.2	任务 2: 信号量实现哲学家就餐问题解决算法	5
1.2.1	算法	5
1.2.2	实现	6
1.2.3	结果	8
1.3	任务 3: 信号量实现读者-写者问题解决算法	9
1.3.1	算法	9
1.3.2	实现	10
1.3.3	结果	11
2	实验心得	12
2.1	信号量操作	12
2.2	信号量作用	12
	参考文献	13

1 实验内容

1.1 任务 1: 信号量实现生产者-消费者问题解决算法

1.1.1 算法

将信号量操作 `up`、`down` 作为系统调用使用, 并在进行测试信号量、更新信号量、使某个进程休眠三种操作时屏蔽全部中断.

(1) 信号量

本算法使用三个信号量:

`full`: 记录充满的缓冲槽数目, 初值为 0.

`empty`: 记录空的缓冲槽数目, 初值为缓冲区槽数目 `N`.

`mutex`: 二元信号量, 确保生产者和消费者不会同时访问缓冲区, 初值为 0.

信号量 `full` 和 `empty` 用于同步. `full` 保证缓冲区满时生产者停止生产, `empty` 保证缓冲区为空时消费者停止购买. 信号量 `mutex` 为二元信号量, 用于互斥, 保证了任意时刻只有一个进程读写缓冲区.

(2) 进程

生产者:

```
1 Producer{
2   while (True) {
3     down(empty)
4     down(mutex)
5     insert_item()
6     up(mutex)
7     up(full)
8   }
9 }
```

消费者:

```
1 Consumer{
2   while (True) {
3     down(full)
4     down(mutex)
5     remove_item()
6     up(mutex)
7     up(empty)
8   }
9 }
```

1.1.2 实现

所有实现仅修改了 main.c 文件.

```
1  // question1
2  #define N 2
3  OS_EVENT *full;
4  OS_EVENT *empty;
5  OS_EVENT *mutex;
6  int comNum=0;
7  static OS_STK producer_stk[TASK_STK_SIZE]; //unsigned int type
8  static OS_STK consumer_stk[TASK_STK_SIZE];
9  void producer(void *p_arg);
10 void consumer(void *p_arg);
11 static void producer(void *p_arg)
12 {
13     int i;
14     INT16U timeout=0;
15     INT8U *perr;
16     for (;;)
17     {
18         print_str("Producer!\n");
19         OSSemPend(empty,timeout,perr);OSTimeDly(1);
20         OSSemPend(mutex,timeout,perr);OSTimeDly(1);
21
22         comNum+=1;//produce
23         print_str("produce 1,comNum=");
24         printNum(comNum);
25         print_str("\n");
26
27         OSSemPost(mutex);OSTimeDly(1);
28         OSSemPost(full);OSTimeDly(1);
29         OSTimeDly(100);
30         for(i=0;i<=10000;i++);
31     }
32 }
33 static void consumer(void *p_arg)
34 {
35     int i;
36     INT16U timeout=0;
37     INT8U *perr;
38     for (;;)
39     {
40         print_str("Consumer!\n");
41         OSSemPend(full,timeout,perr);OSTimeDly(1);
42         OSSemPend(mutex,timeout,perr);OSTimeDly(1);
43
44         comNum-=1;//consume
45         print_str("consume 1,comNum=");
46         printNum(comNum);
47         print_str("\n");
48
49         OSSemPost(mutex);OSTimeDly(1);
50         OSSemPost(empty);OSTimeDly(1);
51         OSTimeDly(200);
52         for(i=0;i<=10000;i++);
53     }
54 }
```

```

55  int main(void)
56  {
57      OSInit();
58      systick_init();
59
60      OSTaskCreate(producer, (void *)1, &producer_stk[TASK_STK_SIZE-1], TASK1_PRIO);
61      OSTaskCreate(consumer, (void *)0, &consumer_stk[TASK_STK_SIZE-1], TASK2_PRIO);
62
63      full = OSSemCreate (0);
64      empty = OSSemCreate (N);
65      mutex = OSSemCreate (1);
66      OSTStart();
67
68      return 0;
69  }

```

1.1.3 结果

通过更改生产者和消费者离开互斥区后的 `OSTimeDly (time)` 可令消费者在不同的时机进行消费. 比如当设置两者 `Dly` 都为 100 时, 生产者生产一个消费者就消费一个; 设置消费者 `Dly` 略大于生产者 `Dly` 的 N 倍, 则可令消费者在缓冲栈填满时才进行消费.

两种设置方式所得结果如下:

```

produce, comNum=1
consume, comNum=0
produce, comNum=1
consume, comNum=0
produce, comNum=1
consume, comNum=0
produce, comNum=1
consume, comNum=0
produce, comNum=1
consume, comNum=0

```

图 1: 延迟大致相同

```

produce, comNum=1
consume, comNum=0
produce, comNum=1
produce, comNum=2
consume, comNum=1
produce, comNum=2
consume, comNum=1
produce, comNum=2
consume, comNum=1
produce, comNum=2

```

图 2: 消费者延迟约为生产者两倍

可观察到, 当消费者延迟约为生产者两倍时, 除开始部分, 消费者都是在缓冲区填满时才进行消费.

1.2 任务 2: 信号量实现哲学家就餐问题解决算法

1.2.1 算法

(1) 定义互斥量:

mutex2: 二元互斥信号量, 确保不会有多个哲学家同时访问临界区, 设初值为 1.

s[N]: 信号量数组, 初值都设为 1. 每个信号量对应一个哲学家, 在其所需叉子被占用时阻塞. 确保一把叉子在同一时刻只有一个哲学家拿起.

(2) 设置哲学家的三种行为:

test: 哲学家 *i* 检查两个邻居状态, 当两个邻居都未处于进餐状态时进餐, 并对信号量 **s[i]** 进行 **up** 操作.

```
1  test(i){
2      if(State[i]==Hungry&& State[Left]!=Eating && State[Right]!=Eating){
3          State[i]=Eating
4          up(s[i])
5      }
6  }
```

take_fork: 对 **mutex2** 进行 **down** 操作, 互斥地进入临界区. 设置哲学家状态为 **Hungry**. 检查是否可以拿到相邻的叉子, 若可以则拿取. **up** 操作 **mutex2**, 离开临界区. 对信号量 **s[i]** 进行 **down** 操作, 如果未能拿到需要的叉子则此时会阻塞, 该哲学家将会一直等待直到叉子可用.

```
1  take_fork(i){
2      down(mutex2)
3      State[i]=Hungry
4      test(i)
5      up(mutex2)
6      down(s[i])
7  }
```

put_fork: 对 **mutex2** 进行 **down** 操作, 互斥地进入临界区. 设置哲学家状态为 **Thinking**, 此时该哲学家使用的两个叉子空闲出来. 先后检查相邻两个哲学家是否可以拿到所需叉子, 若可以则令其停止阻塞并进食. **up** 操作 **mutex2**, 离开临界区.

```
1  put_fork(i){
2      down(mutex2)
3      State[i]=Thinking
4      test(Left)
5      test(Right)
6      up(mutex2)
7  }
```

(3) 进程

哲学家进行如下行为:

```
1 philosopher(i) {
2     while (True)
3     {
4         Thinking()
5         take_fork(i)
6         Eating()
7         put_fork(i)
8     }
9 }
```

1.2.2 实现

```
1  #define N2      5
2  #define Hungry  1
3  #define Eating  2
4  #define Thinking 3
5  #define Right (i+1)%N2
6  #define Left (i+N2-1)%N2
7  OS_EVENT *s[N2];
8  OS_EVENT *mutex2;
9  int State[N2];
10 static OS_STK philo_stk[5][TASK_STK_SIZE];
11 int randNum[100]={2, 2, 3, 3, 2, 2, 1, 5, 1, 4, 3, 3, 4, 3, 5, 2, 2, 1, 2, 1, 3,
12 1, 5, 3, 2, 4, 5, 3, 3, 1, 2, 1, 3, 4, 2, 5, 2, 2, 4, 4, 2, 5, 2, 4, 4, 1, 1, 5,
13 3, 2, 1, 3, 2, 4, 5, 5, 4, 4, 3, 2, 3, 1, 1, 4, 4, 5, 2, 2, 5, 1, 5, 5, 3, 1, 5,
14 1, 4, 5, 3, 3, 4, 5, 3, 5, 4, 4, 4, 3, 4, 4, 2, 5, 3, 1, 1, 3, 4, 5, 4, };
15 int random_choice=0;
16 static void test(int i)
17 {
18     if(State[i]==Hungry&& State[Left]!=Eating && State[Right]!=Eating)
19     {
20         State[i]=Eating;
21         OSSemPost((s[i]));
22     }
23 }
24 static void take_fork(i)
25 {
26     INT16U timeout=0;
27     INT8U *perr;
28     OSSemPend(mutex2,timeout,perr);
29     State[i]=Hungry;
30     test(i);
31     OSSemPost(mutex2);
32     OSSemPend(s[i],timeout,perr);
33 }
34 static void put_fork(i)
35 {
36     INT16U timeout=0;
37     INT8U *perr;
38     OSSemPend(mutex2,timeout,perr);
39     State[i]=Thinking;
```

```

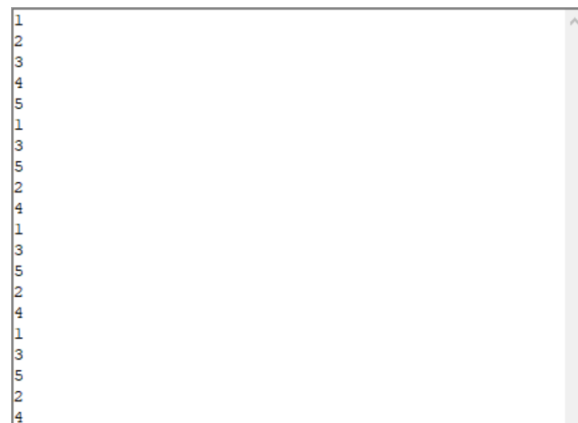
40     test(Left);
41     test(Right);
42     OSSemPost(mutex2);
43 }
44 static void philosopher(void *p_arg)
45 {
46     int waitTime=1;
47     int i=((int*)p_arg);
48     for(;;)
49     {
50         //thinking()
51         take_fork(i);OSTimeDly(1);
52         printNum(i+1);print_str("\n");//eating
53         OSTimeDly(1);
54         put_fork(i);OSTimeDly(1);
55     }
56 }
57 int main(void)
58 {
59     OSInit();
60     systick_init();
61     int nameList[5]={0,1,2,3,4};
62
63     OSTaskCreate(philosopher, (void *)&nameList[0],
64                 &philo_stk[0][TASK_STK_SIZE-1], 1);
65     OSTaskCreate(philosopher, (void *)&nameList[1],
66                 &philo_stk[1][TASK_STK_SIZE-1], 2);
67     OSTaskCreate(philosopher, (void *)&nameList[2],
68                 &philo_stk[2][TASK_STK_SIZE-1], 3);
69     OSTaskCreate(philosopher, (void *)&nameList[3],
70                 &philo_stk[3][TASK_STK_SIZE-1], 4);
71     OSTaskCreate(philosopher, (void *)&nameList[4],
72                 &philo_stk[4][TASK_STK_SIZE-1], 5);
73     for(int j=0;j<5;j++)
74     {
75         s[j]=OSSemCreate(1);
76     }
77     mutex2 = OSSemCreate(1);
78     OSStart();
79     return 0;
80 }

```


1.2.3 结果

由于 uC/OS 并非真正的并行, 只能通过 OS_Time_Dly() 实现伪并行, 无法实现真正的并行. 当所有延迟 Dly 都设置为固定值时, 哲学家进餐的顺序一定是固定的, 并且只与哲学家的优先级有关.

在所有 Dly 值设定为固定值时, 得到结果如下. 除了最开始输出一次"12345", 此后一直循环输出"13524".



```
1
2
3
4
5
1
3
5
2
4
1
3
5
2
4
1
3
5
2
4
1
3
5
2
4
```

图 3: 设置固定 Dly

可通过设置 Dly 值为随机数以实现哲学家问题随机化 (进餐顺序随机). 实现中使用数组保存了一组伪随机数, 令哲学家进餐后等待随机时间, 得到结果如下, 可观察到哲学家进餐顺序不是固定的.



```
1
3
5
2
4
3
1
5
2
4
3
1
5
2
3
1
4
5
1
2
```

图 4: 设置伪随机 Dly 值

1.3 任务 3: 信号量实现读者-写者问题解决算法

1.3.1 算法

(1) 设置信号量:

mutex3: 互斥信号量, 设初值为 1. 控制对读者数 **rc** 的访问.

db: 互斥信号量, 设初值为 1. 控制对数据库 **db** 的访问.

(2) 进程:

reader:

互斥访问 **rc**, 对其进行加 1 操作.

如果是第一个读者, 则对信号量 **db** 进行 down 操作, 要么阻塞等待释放, 要么继续执行.

释放对 **rc** 的互斥访问并读取数据.

互斥访问 **rc**, 对其进行自减 1.

如果是最后一个读者, 则对信号量 **db** 进行 up 操作, 释放数据库访问权.

释放对 **rc** 的互斥访问权.

```
1 reader{
2     while (True) {
3         down(mutex3)
4         rc+=1
5         if (rc==1) down(db)
6         up(mutex3)
7         read()
8         down(mutex3)
9         rc-=1
10        if (rc==0) up(db)
11        up(mutex)
12    }
13 }
```

writer: 获得 **db** 互斥访问权, 更新数据, 释放 **db** 的互斥访问权.

```
1 writer{
2     while (True) {
3         down(db)
4         write()
5         up(db)
6     }
7 }
```

1.3.2 实现

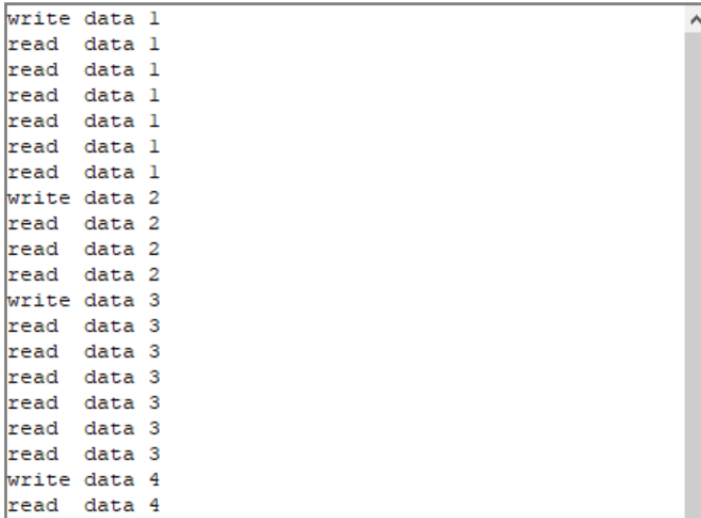
本文实现了读者优先版本的算法, 写者等待所有读者完成读取后才进行写入. 设置三个读者和一个写者, 写者每次将数据修改为原值加 1.

```
1  #define N3 2
2  OS_EVENT *db;
3  OS_EVENT *mutex3;
4  int rc=0;
5  static OS_STK muti_stk[10][TASK_STK_SIZE]; //unsigned int type
6  int data=0;
7  void reader(void *p_arg)
8  {
9      INT16U timeout=10;
10     INT8U *perr;
11     for(;;)
12     {
13         OSSemPend(mutex3,timeout,perr);OSTimeDly(1);
14         rc+=1;OSTimeDly(1);
15         if(rc==1){OSSemPend(db,timeout,perr);}
16         OSSemPost(mutex3);OSTimeDly(1);
17
18         syscall_print_str("read data ");printNum(data);
19         syscall_print_str(" \n");OSTimeDly(50);
20
21         OSSemPend(mutex3,timeout,perr);OSTimeDly(1);
22         rc-=1;OSTimeDly(1);
23         if(rc==0){OSSemPost(db);}
24         OSSemPost(mutex3);OSTimeDly(1);
25         OSTimeDly(20);
26     }
27 }
28 void writer(void *p_arg){
29     INT16U timeout=10;
30     INT8U *perr;
31     for(;;)
32     {
33         OSSemPend(db,timeout,perr);OSTimeDly(1);
34         data+=1;syscall_print_str("write data ");
35         printNum(data);syscall_print_str(" \n");OSTimeDly(1);
36         OSSemPost(db);OSTimeDly(1);
37         OSTimeDly(100);
38     }
39 }
40 int main(void)
41 {
42     OSInit();
43     systick_init();
44
45     OSTaskCreate(reader, (void *)0,
46                  &muti_stk[0][TASK_STK_SIZE-1], 1);
47     OSTaskCreate(reader, (void *)0,
48                  &muti_stk[1][TASK_STK_SIZE-1], 2);
49     OSTaskCreate(reader, (void *)0,
50                  &muti_stk[2][TASK_STK_SIZE-1], 3);
51     OSTaskCreate(writer, (void *)0,
52                  &muti_stk[3][TASK_STK_SIZE-1], 4);
```

```
53
54     mutex3 = OSemCreate (1);
55     db=OSemCreate(1);
56     OSStart();
57     return 0;
58 }
```

1.3.3 结果

实现所得结果如下:



```
write data 1
read data 1
read data 1
read data 1
read data 1
read data 1
read data 1
read data 1
write data 2
read data 2
read data 2
read data 2
read data 2
write data 3
read data 3
read data 3
read data 3
read data 3
read data 3
read data 3
write data 4
read data 4
```

图 5: 读者-写者问题

可观察到所有读者读取到的都是写者最新写入的数据, 不存在对着读取到过去数据的情况.

2 实验心得

2.1 信号量操作

对信号量有两种操作 `up` 和 `down`, 且都为原子操作:

`up(s)`:

如果 `s` 的值大于零, 就给它减 1;

如果 `s` 值为零, 就挂起该进程的执行.

`down(s)`:

如果有其他进程因等待 `s` 而被挂起, 就让它恢复运行;

如果没有进程因等待 `s` 而挂起, 就给 `s` 加 1.

2.2 信号量作用

信号量主要用于进程之间的互斥和同步.

互斥: 保证一个不可共享的资源同时只有一个进程访问.

同步: 保证进程的先后执行顺序.

实现互斥:

设置互斥信号量 `mutex`, 初值为 1.

在临界区之前执行 `down(mutex)`.

在临界区之后执行 `up(mutex)`.

实现同步:

设置同步信号量 `s`, 初始为 0.

在“前操作”之后执行 `up(s)`.

在“后操作”之前执行 `down(s)`.

参考文献

- [1] Andrew S.Tanenbaum,Herbert Bos. 现代操作系统. 北京: 机械工业出版社,2017.