项目三　　LiteOS 同步实验

22920212204396　　黄子安

一、实验目的

1、利用 Pthread 库，实现生产者-消费者问题。

二、实验环境

1.物理机：windows 操作系统
2.VMware 虚拟机：ubuntu 18.04.6
3.开发板：imx6ull Mini

三、实验内容

1、问题描述

生产者和消费者共享一个初始化为空、大小 n 的缓冲区，只有缓冲区没满时，生产者才能把消息写入缓冲区，否则需要等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息或者一个消费者从中取出消息。

2、实验思路

为了实现互斥与同步，这里需要用到三个信号量
1、信号量 mutex 作为互斥信号量，用于控制互斥访问缓冲池，初值为 1
2、信号量 full 用于记录当前缓冲区是否已满，初值为 0
3、信号量 empty 用于记录缓冲区是否空，初值为 n
这里信号量的实现可以使用头文件<semaphore.h>提供的 sem_t 类型和相关函数

3、函数介绍

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_r
outine) (void *), void *arg)
```

用于创建一个线程

- thread：指向线程标识符的指针，用于存储新创建的线程标识符
- attr：线程属性的指针，用于设置线程的属性，NULL 表示使用默认属性
- start_routine：线程函数的地址，新线程将从该函数开始执行
- arg：传递给线程函数的参数

```
int pthread_join(pthread_t thread, void **retval)
```

用于等待一个线程的结束

- thread：要等待结束的线程标识符
- retval：用于存储线程返回值的指针，可以为 NULL

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

用于初始化一个信号量

- sem：指向信号量对象的指针。
- pshared：指定信号量的共享方式，0 表示信号量只能在当前进程内的线程之间共享，非零表示信号量可在多个进程之间共享。
- value：指定信号量的初始值

```
int sem_wait(sem_t *sem)
```

用于对信号量进行等待操作，如果信号量的值大于 0，则将其减 1；如果信号量的值为 0，则阻塞等待，即 **P 操作**

- sem：指向信号量对象的指针

```
int sem_post(sem_t *sem)
```

用于对信号量的值进行自增 1 操作，即 **S 操作**

- sem：指向信号量对象的指针。

4、实验程序

4.1、定义缓冲区

先实现一个**队列**用作缓冲区，只需要定义一个 Front 指针和 Rear 指针即可，之后实现入队和出队的函数，这里为了让实验现象更加明显，定义了 BUFFER_SIZE 是 16，使得队列更容易满从而体现互斥与同步。这里还有一点就是因为使用了两个信号量来控制缓冲区是不是满的或者空的，所以不需要对 Front 和 Rear 进行额外判断两者相等时队列是空的还是满的

```
int buffer[BUFFER_SIZE];
int front=0,rear=0;
void enqueue(int x)
{
    buffer[front++]=x;
    if(front>=BUFFER_SIZE) front%=BUFFER_SIZE;
}

int dequeue( )
{
    int x=buffer[rear++];
    if(rear>=BUFFER_SIZE) rear%=BUFFER_SIZE;
    return x;
}
```

4.2、实现生产者

生产者会不断往缓冲区中写入数据，在开始之前需要先对信号量 empty 执行 P 操作，来确保当前缓冲区有空位可以输入数据并获得这些空位，之后再对信号量 mutex 执行 P 操作，来请求进入临界区再往里面写入数据；完成后对信号量 mutex 执行 S 操作，表示离开并释放临界区，再对信号量 full 执行 S 操作，表示缓冲区中数据+1

```
void * producer()
{
    printf("producer start!\n");
    for (int i = 0; i <= OVER; i++) {
        printf(" put-->%d\n", i);
        sem_wait(&empty);
        sem_wait(&mutex);

        enqueue(i);

        sem_post(&mutex);
        sem_post(&full);
    }
    printf("producer stopped!\n");
    return NULL;
}
```

4.3、实现消费者

消费者会不断从缓冲区读取数据，在开始之前需要先对信号量 full 执行 P 操作，来确保当前缓冲区存在可以读取的数，之后再对信号量 mutex 执行 P 操作，来请求进入临界区，之后从中读取数据；完成后对信号量 mutex 执行 S 操作，表示离开并释放临界区，再对信号量 empty 执行 S 操作，表示缓冲区中空数量+1

```
void * consumer()
{
    printf("consumer start!\n");
    while (1)
    {
        sem_wait(&full);
        sem_wait(&mutex);

        int d=dequeue();

        sem_post(&mutex);
        sem_post(&empty);
        printf("              %d-->get\n", d);
        if (d == OVER ) break;
    }

    printf("consumer stopped!\n");
    return NULL;
}
```

4.4、初始化函数

主线程中先对三个信号量进行初始化，之后创建对应的线程，为了避免主线程退出，需要用到 join 函数阻塞主线程直到子线程运行结束

```c
int main()
{
    pthread_t pid1,pid2;
    sem_init(&mutex,0,1);
    sem_init(&empty,0,BUFFER_SIZE);
    sem_init(&full,0,0);

    pthread_create(&pid1,NULL,producer,NULL);
    pthread_create(&pid2,NULL,consumer,NULL);

    pthread_join(pid1,NULL);
    pthread_join(pid2,NULL);

    return 0;
}
```

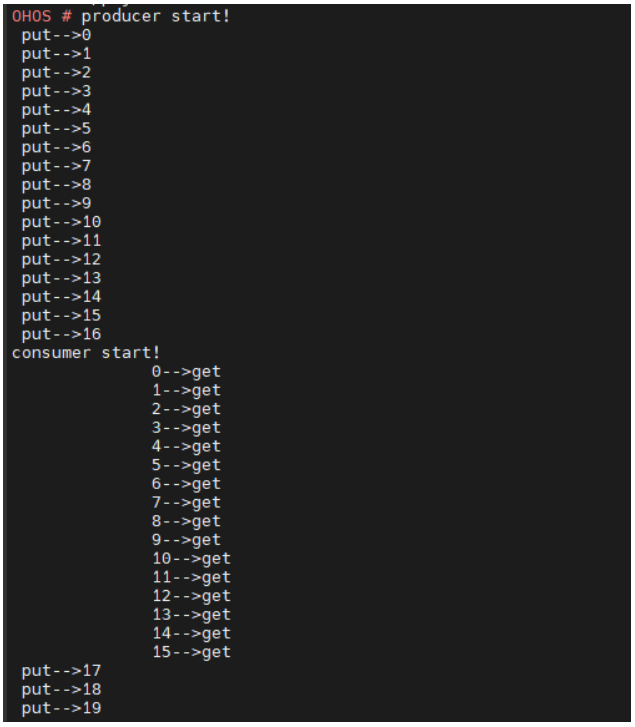
4.5、编译运行代码

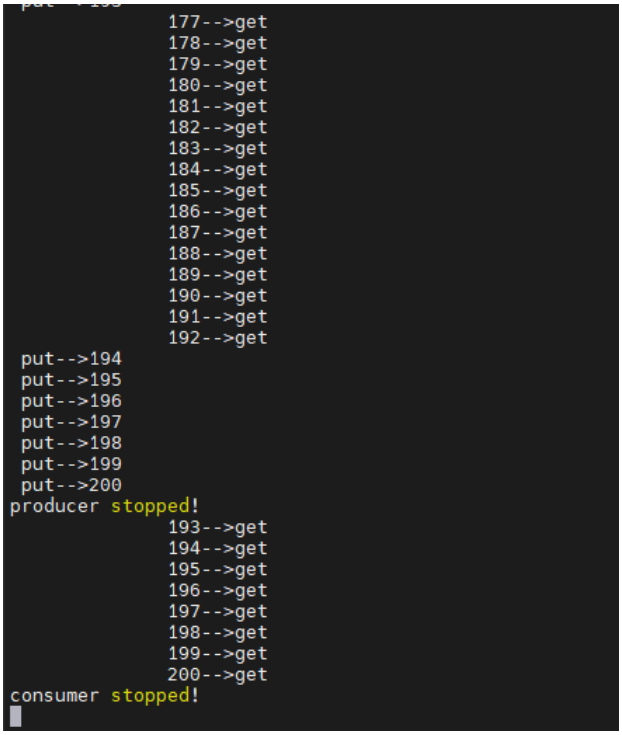
同之前实验都一样，需要使用交叉编译并烧写到开发板中，命令如下：

```
//编译内核
cd  /home/book/openharmony/kernel/liteos_a
make clean
make -j 16
make rootfs
//交叉编译 prj3
cd ~
clang -target arm-liteos --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/ -o prj3 prj3.c
cp  prj3  /home/book/openharmony/kernel/liteos_a/out/imx6ull/rootfs/bin
cd  /home/book/openharmony/kernel/liteos_a/out/imx6ull/
mkfs.jffs2  -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
```

四、实验结果

运行结果如下图所示





五、实验分析

通过实验截图可以发现两个线程之间实现了以下功能点：

- 只有缓冲区没满时，生产者才能把消息写入缓冲区，否则需要等待，图中可以看到生产者**最多放入了 16 个数据后缓冲区就满了，所以生产者就被迫进行等待**
- 只有缓冲区不空时，消费者才能从中取出消息，否则必须等待，从结果图中可以看出消费者**最多读取 16 个数据后缓冲区就空了，所以消费者就被迫进行等待**

到此看似达到目标了，但是我个人认为其实实验过程并没有体现缓冲区是临界资源，即只允许一个生产者放入消息或者一个消费者从中取出消息，因为缓冲区太小了，生产者在时间片轮转前就早已完成了写满缓冲区，这个时候生产者会因为 empty=0 而被阻塞，没有体现 mutex 对缓冲区起到了互斥作用，所以需要加一个验证实验证明 mutex 发挥了作用

验证实验思路只要让生产者每次执行动作的时间大于轮转周期即可，这个时候 mutex 锁的作用就会被体现，稍微修改下生产者代码，这里信号量要改成+2 或-2，因为一次会写入两个数据

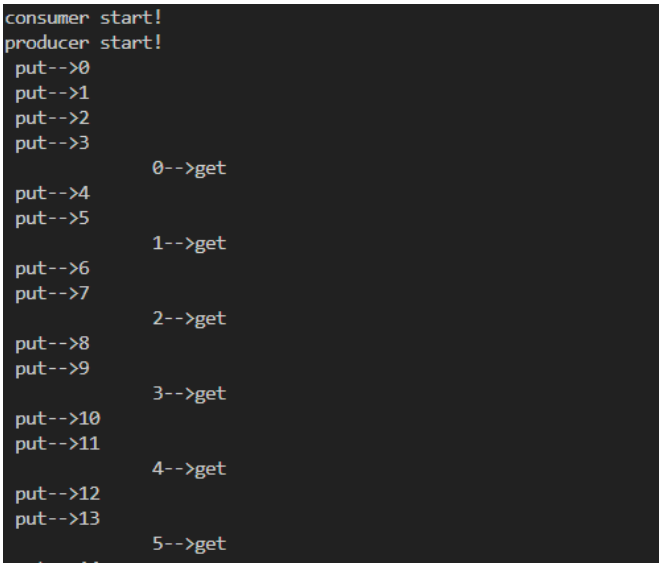
```
void * producer()
{
    printf("producer start!\n");
    for (int i = 0; i <= 2*OVER; i++) {

        sem_wait(&empty);
        sem_wait(&empty);
        sem_wait(&mutex);

        enqueue(i);
        printf(" put-->%d\n", i);
        sleep(1);
        enqueue(++i);
        printf(" put-->%d\n", i);

        sem_post(&mutex);
        sem_post(&full);
        sem_post(&full);
    }
    printf("producer stopped!\n");
    return NULL;
}
```

运行结果如下图：



通过分析可以发现生产者放入数字 0 后生产者会睡眠 1s，这个时候必然会轮到消费者了，但是消费者没有读取 0，原因就是生产者通过 mutex 锁住了缓冲区，从而使得消费者没法访问缓冲区，证明 mutex 起到了互斥作用

至此证明了这组信号设置在 Liteos 中实现了消费者-生产者之间的同步与互斥关系

这里还有一个问题就是为什么是先对 **empty、full 进行 P 操作再对 mutex 进行 P 操作**，而不能反过来，理由如下：

考虑如果反过来写，假设当生产者在某时刻填满了缓冲区，但是此时消费者线程并没有进行读取操作（虽然实验中不会出现这个情况但是现实调度肯定会出现），这个时候会导致 empty 依旧为 0，再时间片轮转回生产者后就会先将 mutex 减为 0 获取缓冲区，但是因为 empty=0，所以生产者被 P 操作阻塞了；这个时候消费者也没法去访问缓冲区了（因为 mutex 被生产者置为 0 了，生产者被阻塞了没法对 mutex 执行 S 操作），这就导致消费者也被阻塞了，造成死锁现象；同理考虑消费者也会出现这个情况，因此对 empty、full 和 mutex 进行 P 操作有先后顺序。

六、实验总结

本次实验通过 pthread 库实现了生产者和消费者模型，从实际编码上感受到了信号量的使用方式，比在书上纯理论看到的要深刻很多，也发现了很多只通过理论学习不会发现的细节问题

七、参考资料

[1] https://blog.csdn.net/m0_58367586/article/details/127710002

[2]https://blog.csdn.net/RuanFun/article/details/134550629?spm=1001.2014.3001.5502

八、附录

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define BUFFER_SIZE 16
#define OVER 200
sem_t mutex,full,empty;

int buffer[BUFFER_SIZE];
int front=0,rear=0;
void enqueue(int x)
{
    buffer[front++]=x;
    if(front>=BUFFER_SIZE) front%=BUFFER_SIZE;
}

int dequeue( )
{
    int x=buffer[rear++];
    if(rear>=BUFFER_SIZE) rear%=BUFFER_SIZE;
    return x;
}

void * producer()
{
    printf("producer start!\n");
   for (int i = 0; i <= OVER; i++) {
        printf(" put-->%d\n", i);
        sem_wait(&empty);
        sem_wait(&mutex);

        enqueue(i);

     sem_post(&mutex);
        sem_post(&full);
 }
   printf("producer stopped!\n");
```

```
37.    return NULL;
38.}
39.void * consumer()
40.{
41.     printf("consumer start!\n");
42.    while (1)
43.     {
44.         sem_wait(&full);
45.         sem_wait(&mutex);
46.
47.         int d=dequeue();
48.
49.         sem_post(&mutex);
50.         sem_post(&empty);
51.         printf("               %d-->get\n", d);
52.         if (d == OVER ) break;
53.    }
54.
55.    printf("consumer stopped!\n");
56.    return NULL;
57.}
58.int main()
59.{
60.     pthread_t pid1,pid2;
61.     sem_init(&mutex,0,1);
62.     sem_init(&empty,0,BUFFER_SIZE);
63.     sem_init(&full,0,0);
64.
65.     pthread_create(&pid1,NULL,producer,NULL);
66.     pthread_create(&pid2,NULL,consumer,NULL);
67.
68.     pthread_join(pid1,NULL);
69.     pthread_join(pid2,NULL);
70.
71.     return 0;
72.}
```