01 实验概述

- 了解和熟悉 linux 系统下的信号量集和共享内存。
- 使用 linux 系统提供的信号量集（`semget`、`semop`、`semctl` 等系统调用）和共享内存（`shmget`、`shmat`、`shmdt`、`shmctl` 等系统调用）实现生产者和消费者问题。

02 实验过程

2.1 缓冲区定义

缓冲区要求:

- 缓冲区共 `BUFFER_CELL_NUM` 块，每块 `BUFFER_CELL_SIZE` 字节，每块缓可存入文本；
- 指针 `in` 指向第一个空缓存块，指针 `out` 指向第一个满缓存块；
- 可向 `in` 位置存入产品，从 `out` 位置取走产品。造成效果：先进先出。

缓冲区抽象:

- 从数据结构角度，此缓冲区为循环队列，`in` 为队尾、`out` 为队头；

缓冲区实现:

首先用结构体封装单个缓存块。

```
#define BUFFER_CELL_SIZE 1024
#define BUFFER_CELL_NUM 5

// 一个缓存块
struct BufferCell
{
    char text[BUFFER_CELL_SIZE];
};
```

利用多个缓存块组成缓存区结构。

```
// 缓存区，包含若干缓存块
// 从数据结构角度，实际即循环队列
struct Buffer
{
    int in, out, n;  // 队列尾，队列头，队列空间大小
    BufferCell data[BUFFER_CELL_NUM];

    // 将tmp指向的文本装入Buffer队尾
    void push_back(char *tmp)
    {
        // 1. 装入数据
        strncpy(data[in].text, tmp, (BUFFER_CELL_SIZE-1) * sizeof(char));
        data[in].text[BUFFER_CELL_SIZE - 1] = '\0';
        // 2. 更新指针
        in++;
        in %= BUFFER_CELL_NUM;
        n++;
    }

    // 将Buffer队头的元素移出，并将其内容拷贝到tmp
    void pop_front(char *tmp)
    {
        // 1. 拷贝出数据到data
        strncpy(tmp, data[out].text, (BUFFER_CELL_SIZE-1) * sizeof(char));
        tmp[BUFFER_CELL_SIZE - 1] = '\0';
        // 2. 更新指针
        out++;
        out %= BUFFER_CELL_NUM;
        n--;
    }

    // 打印当前Buffer状态（尾指针位置in，头指针位置out，n个位置的空满状态）
    void print()
```

```
    {
        // 代码略，详见实验源代码
        // 可实现如下打印效果：('#'为满缓冲区，`-`为空缓冲区)
        // 缓冲区状态：[buffer.in] = 1,  [buffer.out] = 0,  [data] = [ #---- ]
    }
};
```

2.2 信号量控制

本实验共需3个信号量，分别用于：

1. 缓冲区互斥访问；
2. 维护空缓冲区个数；
3. 维护满缓冲区个数；

对上述3个信号量，需要进行如下3种操作：

1. 赋初值；
2. P（wait）操作；
3. V（signal）操作；

对上述3个操作做如下封装：

```
#include <iostream>
#include <unistd.h>
#include <sys/sem.h>
// 初始化信号量集semid中第index个信号量，赋值为initial_value
void initialize_semaphores(int semid,int index,int initial_value)
{
    // 使用 semctl 进行初始化
    if (semctl(semid, index, SETVAL, initial_value) == -1)
    {
        perror("semctl");
        exit(EXIT_FAILURE);
    }
}

// 执行 P 操作
// 对semid的信号量数组的第semaphore_num信号量操作
void semaphore_P(int semid, int semaphore_num)
{
```

```
    struct sembuf operation;
    operation.sem_num = semaphore_num;
    operation.sem_op = -1; // P 操作
    operation.sem_flg = 0;
    if (semop(semid, &operation, 1) == -1)
    {
        perror("semop P");
        exit(EXIT_FAILURE);
    }
}

// 执行 V 操作
// 对semid的信号量数组的第semaphore_num信号量操作
void semaphore_V(int semid, int semaphore_num)
{
    struct sembuf operation;
    operation.sem_num = semaphore_num;
    operation.sem_op = 1; // V 操作
    operation.sem_flg = 0;
    if (semop(semid, &operation, 1) == -1)
    {
        perror("semop V");
        exit(EXIT_FAILURE);
    }
}
```

2.3 信号量获取与初始化

在获取信号量时，消费者与生产者的行为有所不同：

- 生产者：
  - 若对应信号量数组已经存在，则直接获取；
  - 若不存在，则创建，且需对信号量赋初值；
- 消费者：
  - 仅可获取已经存在的信号量，即信号量应由生产者创建；
  - 若不存在，则报错后结束进程；

本实验所需信号量的初值如下：

- 第0个信号量， 初值设为1 （用于缓冲区互斥访问）；

- 第1个信号量， 初值设为 BUFFER_CELL_NUM （维护 EMPTY 缓冲区个数）；
- 第2个信号量， 初值设为0 （用于维护 FULL 缓冲区个数）；

```
// 获取信号量（3个），赋初值，返回其id
// key为信号量数组创建键值； create为1表示不存在则创建，为0不存在则报错
int get_semaphore(int key, int create = 1)
{
    // 本实验共需3个信号量
    int num_semaphores = 3;
    // 尝试获取，如果不存在则返回-1
    int t_semid = semget(key, num_semaphores, 0777);
    // 如果返回-1，则说明需要创建此信号量
    int isInitial = t_semid == -1;
    // 消费者调用此函数时，create=0，不能由消费者创建信号量
    if (!create && isInitial)
    {
        printf("消费者获取 KEY=%d 的信号量失败，需要通过生产者创建！\n", key);
        exit(1);
    }
    // 尝试创建信号量
    if ((t_semid = semget(key, num_semaphores, 0777 | IPC_CREAT)) == -1)
    {
        printf("Semaphore Array Creat Failed!\n");
        exit(1);
    }
    // 如果是首次创建信号量，则进行初始化
    if (isInitial)
    {
        printf("初次创建 KEY=%d 信号量数组，信号量初始化……\n", key);
        //  a. 将第0个信号量， 初值设为1 （用于缓冲区互斥访问）
        initialize_semaphores(t_semid, 0, 1);
        //  b. 将第1个信号量， 初值设为 BUFFER_CELL_NUM （EMPTY缓冲区个数）
        initialize_semaphores(t_semid, 1, BUFFER_CELL_NUM);
        //  c. 将第2个信号量， 初值设为0 （FULL 缓冲区个数）
        initialize_semaphores(t_semid, 2, 0);
    }
    else
        printf("已存在 KEY=%d 的信号量，直接获取……\n", key);

    return t_semid;
}
```

2.4 共享内存获取与初始化

共享内存获取要求：

- 对于生产者：
    - 若对应共享内存已存在，则直接获取；
    - 若不存在，则创建；且需要对此空间进行初始化（清零）；
- 对于消费者：
    - 仅可获取已经存在的的共享内存，即应由生产者创建；
    - 若不存在，则报错后结束进程；
- 如何使用此共享空间创建 `Buffer`？
    1. 获取空间大小与Buffer相同的共享内存空间；
    2. 将 `Buffer *buffer` 指针指向此共享内存，即可通过该指针操作；

共享内存获取与初始化实现：

```
// 获得共享内存区，进行初始化，返回其id
int get_share_memory(int key, int create = 1)
{
    // 尝试获取，如果不存在则返回-1
    int t_shmid = shmget(key, sizeof(Buffer), 0777);
    // 如果返回-1，则说明需要创建此共享内存
    int isInitial = t_shmid == -1;
    // 消费者调用此函数时，create=0，不能由消费者创建共享内存
    if (!create && isInitial)
    {
        printf("消费者获取 KEY=%d 的共享内存失败，需要通过生产者创建！\n", key);
        exit(1);
    }
    // 尝试创建共享内存
    if ((t_shmid = shmget(key, sizeof(Buffer), 0777 | IPC_CREAT)) == -1)
    {
        printf("Share Memory Creat Failed!\n");
        exit(1);
    }
    //  获得共享内存起始位置指针（转为BufferCell指针，方便存取操作）
    buffer = (Buffer *)shmat(t_shmid, 0, 0);
    //  初始化（如果是初次创建共享内存，则初始化， 队列的头尾指针被赋值为0）
    if (isInitial)
```

```
    {
        printf("初次创建 KEY=%d 的共享内存，共享内存初始化……\n", key);
        memset(buffer, 0, sizeof(Buffer));
    }
    else
        printf("已存在 KEY=%d 的共享内存，直接获取……\n", key);

    return t_shmid;
}
```

2.5 删除共享内存与信号量

在生产者和消费者的三个操作中，其操作3均为删除共享内存与信号量，实现如下：

```
// 操作3，删除信号量和共享内存；
void delete_sem_and_shm()
{
    // 删除信号量集
    if (semctl(semid, 3, IPC_RMID) == -1)
    {
        perror("semctl IPC_RMID");
        exit(EXIT_FAILURE);
    }

    // 删除共享内存
    if (shmctl(shmid, IPC_RMID, 0) == -1)
    {
        perror("shmctl IPC_RMID");
        exit(EXIT_FAILURE);
    }
}
```

2.6 生产者 produce 操作

在进行 produce 操作前后，需要进行信号量操作：

- 在生产之前，
    1. 申请空缓冲区
    2. 申请缓冲区互斥操作权限

- 在生产之后，
    1. 释放缓冲区控制权限
    2. 增加满缓冲区个数

produce 实现如下：

```
// 操作1，生产产品
void produce()
{
    // 读入生产产品
    printf("输入产品内容 > ");
    scanf("%s", buf);

    semaphore_P(semid, 1);  // 申请空缓冲区
    semaphore_P(semid, 0);  // 申请缓冲区互斥操作权限

    // 进行缓冲区内互斥操作，装入产品
    buffer->push_back(buf);

    semaphore_V(semid, 0);  // 释放缓冲区控制权限
    semaphore_V(semid, 2);  // 增加满缓冲区个数

    // 输出操作后缓冲区状态
    buffer->print();
}
```

2.7 消费者 consume 操作

消费者 consume 操作与生产者的 produce 同理，实现如下。

```
// 操作1，消费产品
void consume()
{
    // 申请满缓冲区
    semaphore_P(semid, 2);
    // 申请缓冲区互斥操作权限
    semaphore_P(semid, 0);

    // 进行缓冲区内互斥操作，装入产品
    buffer->pop_front(buf);

```

```
    // 释放缓冲区控制权限
    semaphore_V(semid, 0);
    // 增加空缓冲区个数
    semaphore_V(semid, 1);

    // 输出消费产品内容
    printf("消费产品内容 : %s\n", buf);
    // 输出操作后缓冲区状态
    buffer->print();
}
```

2.8 Producer 与 Consumer 主函数

经过上述封装，Producer 与 Consumer 的主函数大致相同：

- 均为先申请信号量和共享内存，只不过Consumer不能主动创建而只能获取；
- 均在准备过后，进入操作循环，且二者的操作2、操作3相同；
- 唯一不同的操作1，produce 与 consume 的区别也仅在于申请的资源相反、对缓冲区的操作相反（装入与拿出）。

这里以 `Consumer` 为例：

```
int main(int argc, char *argv[])
{
    // 获取共享内存区
    shmid = get_share_memory(SHMKEY, 0);  // 若为生产者，则第二个参数为1
    // 获取信号量（3个，分别为缓冲区互斥信号量、满缓冲区数量，空缓冲区数量），并赋初值
    semid = get_semaphore(SEMKEY, 0);     // 若为生产者，则第二个参数为1
    // 输出缓冲区初始状态
    buffer->print();
    // 进入消费者操作循环
    while (1)
    {
        printf("可选择操作：[1] 消费产品；[2] 退出进程；[3] 删除信号量和共享内存；\n"
               "输入操作编号 > ");
        scanf("%s", buf);
        if (buf[0] == '1')
            consume();  // 若为生产者，则改为上述定义的produce()函数
        else if (buf[0] == '2')
            break;
        else if (buf[0] == '3')
```

```
        {
            delete_sem_and_shm();
            break;
        }
        else
            printf("输入格式操作！重新输入……\n");
        printf("\n");
    }
    printf("\nConsumer 退出进程。\n");
    return 0;
}
```

03 实验测试与结果

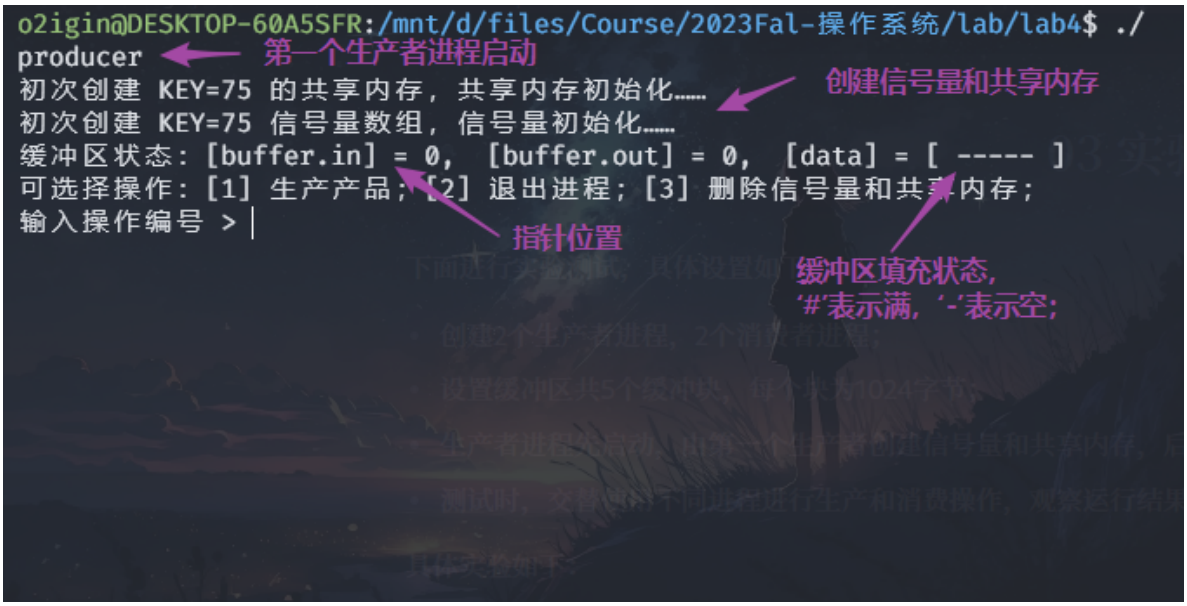
下面进行实验测试，设置如下：

- 创建2个生产者进程，2个消费者进程；
- 设置缓冲区共5个缓冲块，每个块为1024字节；
- 生产者进程先启动，由第一个生产者创建信号量和共享内存，后续进行直接获取；
- 测试时，交替使用不同进程进行生产和消费操作，观察运行结果。

具体实验如下：

1）启动第1个 `producer` 进程，如下图所示。

- 由于是第一个进程，所以由此进程创建信号量和共享内存；
- 缓冲区指针 `in` 、指针 `out` 、5个缓冲块装填状态 `data` 如图所示；



2）在 `producer 1` 中选择操作1，创建一个内容为 `111` 的产品，如下图所示。

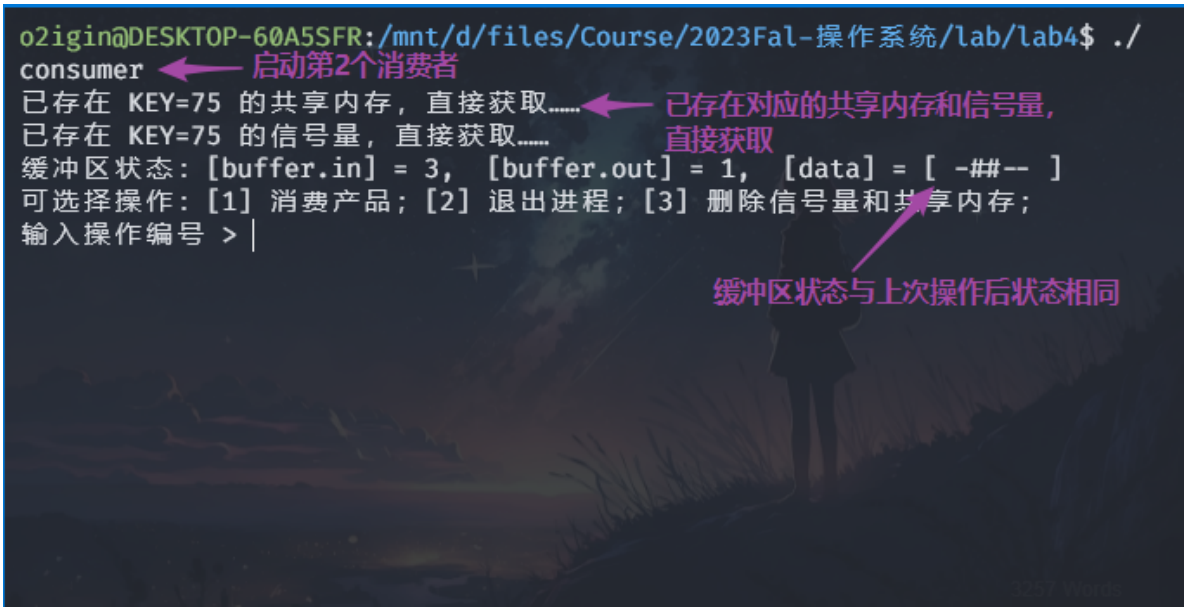
3）启动第2个 `producer` 进程，如下图所示。

4） 在 `producer 2` 中，连续进行两次操作1，依此生产产品 `222` 、 `333` 。

5） 启动第1个消费者 `consumer 1` ，如下图所示。

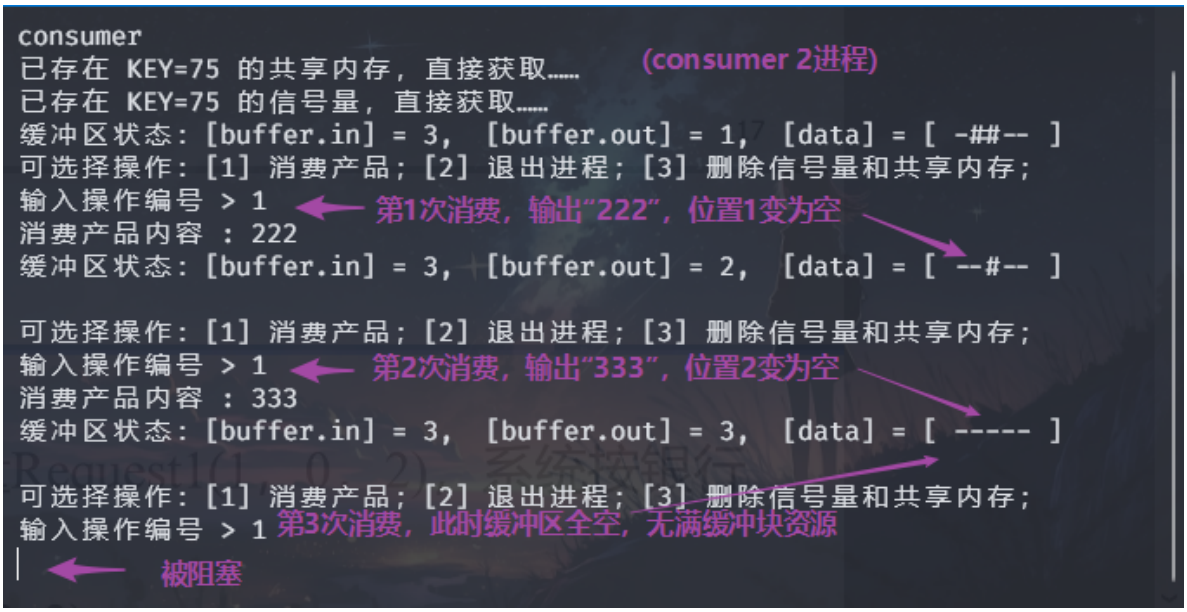
6） `consumer 1` 选择操作1，消费一个产品，打印到屏幕。

7）启动第二个消费者 `consumer 2` ，如下图所示。



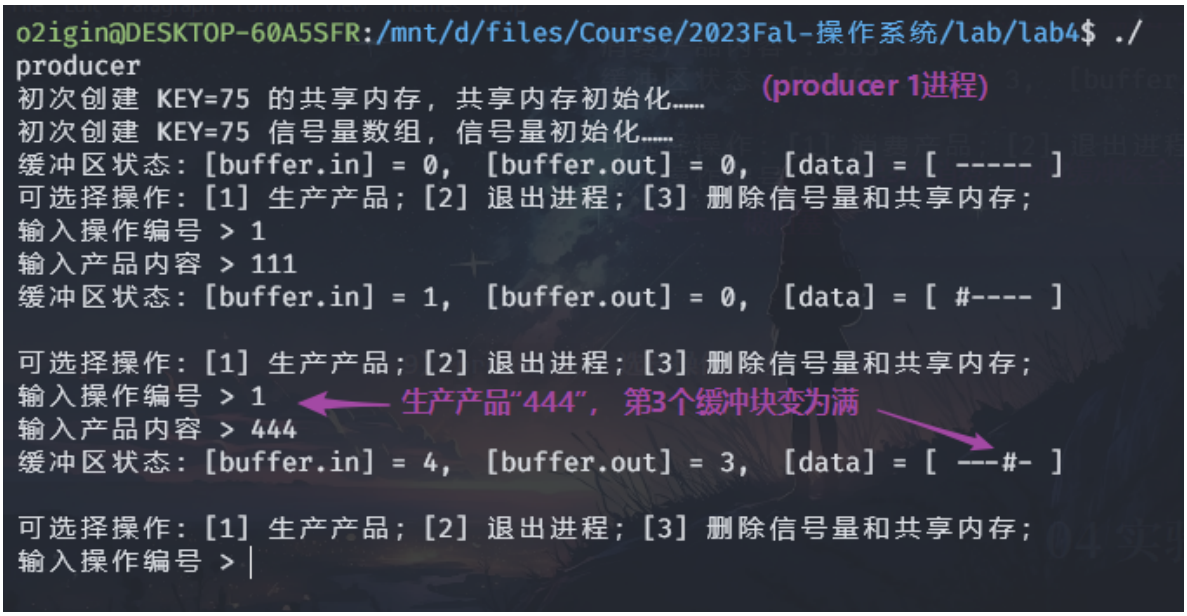
8） `consumer 2` 连续进行3次消费操作，输出如下图所示。

当进行第3此消费时，由于缓冲区全空，所以该 `consumer` 进程 被阻塞。



9） `producer 1` 选择操作1，生产 `444` 。

在步骤8）时，缓冲区被消费至全空，当此步骤生产 `444` 后，缓冲块 `3` 位置变为满缓冲区。



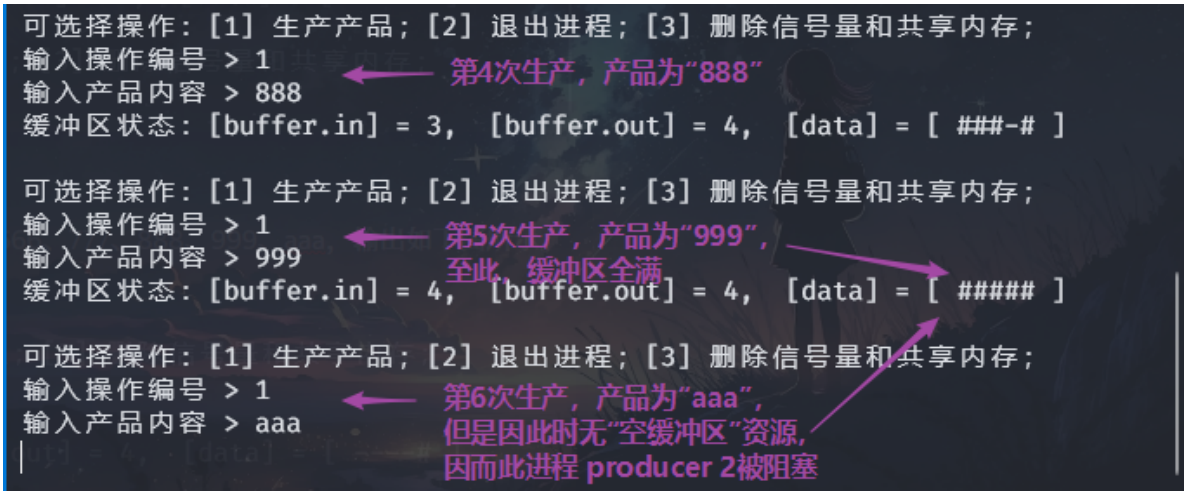
同时，由于有新的满缓冲块资源产生，在步骤8）被阻塞的进程 `consumer 2` 被唤醒，消费这个刚刚生产的 `444` 产品，如下图所示。

10）生产者进程 `producer 2` 连续进行6次操作1，依此生产555、666、777、888、999、aaa，输出如下图所示。

进行前5此操作时，可以顺利生产。

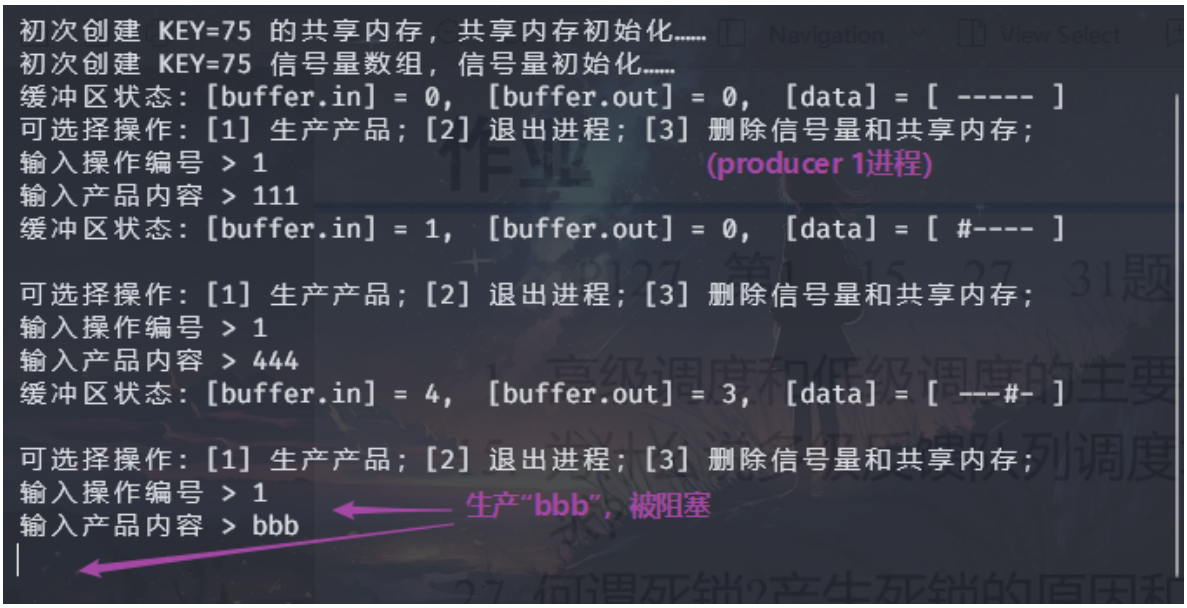
```
可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 555
缓冲区状态：[buffer.in] = 0,  [buffer.out] = 4,  [data] = [ ----# ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 666
缓冲区状态：[buffer.in] = 1,  [buffer.out] = 4,  [data] = [ #---# ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 777
缓冲区状态：[buffer.in] = 2,  [buffer.out] = 4,  [data] = [ ##--# ]
```

当进行第6次操作（生产”aaa“时），因此时已无”空缓冲区“资源，因而 producer 2 进程被阻塞。



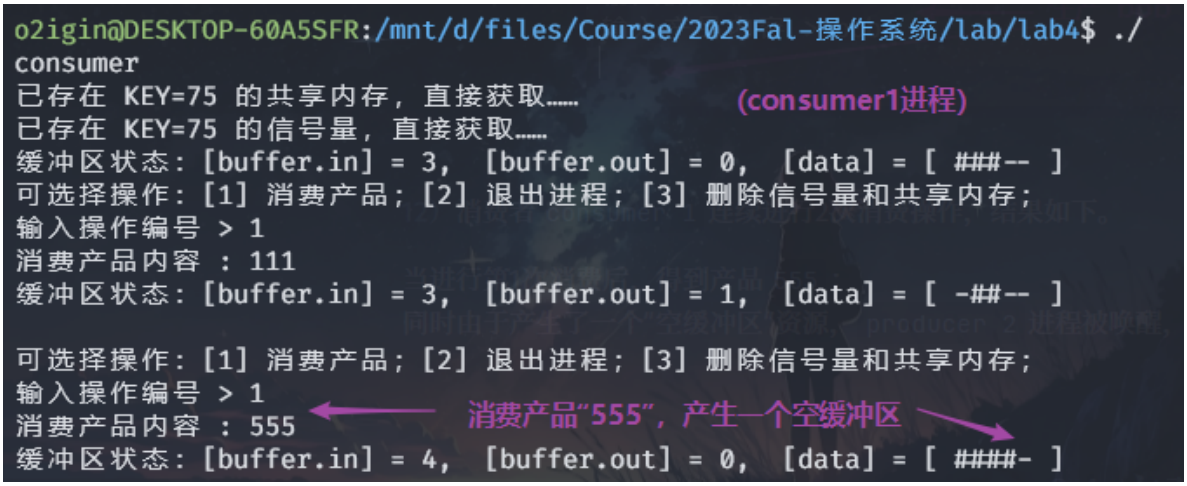
11）生产者 producer 1 进行操作1，生产 bbb 。

由于无”空缓冲区“资源，因此 producer 1 进程也被阻塞；



12）消费者 `consumer 1` 连续进行2次消费操作，结果如下。

当进行第1次消费后，得到产品 `555` ；
同时由于产生了一个"空缓冲区"资源， `producer 2` 进程被唤醒，完成生产产品 `aaa` ；
而此时 `producer 1` 进程仍处于阻塞状态；



```
可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 888                          (producer 2进程)
缓冲区状态：[buffer.in] = 3,  [buffer.out] = 4,  [data] = [ ###-# ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 999
缓冲区状态：[buffer.in] = 4,  [buffer.out] = 4,  [data] = [ ##### ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > aaa      <-- 被唤醒，完成生产产品"aaa"，缓冲区再次变满
缓冲区状态：[buffer.in] = 0,  [buffer.out] = 0,  [data] = [ ##### ]
```

当进行第1次消费后，得到产品 666 ；
同时由于再次产生了一个"空缓冲区"资源， producer 1 进程被唤醒，完成生产产品 bbb ；

```
可选择操作：[1] 消费产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
消费产品内容 ：111                          (consumer 1进程)
缓冲区状态：[buffer.in] = 3,  [buffer.out] = 1,  [data] = [ -##-- ]

可选择操作：[1] 消费产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
消费产品内容 ：555
缓冲区状态：[buffer.in] = 4,  [buffer.out] = 0,  [data] = [ ####- ]

可选择操作：[1] 消费产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
消费产品内容 ：666      <-- 消费产品"666"，再次产生一个空缓冲区
缓冲区状态：[buffer.in] = 0,  [buffer.out] = 1,  [data] = [ -#### ]
```

```
可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 111                          (producer 1进程)
缓冲区状态：[buffer.in] = 1,  [buffer.out] = 0,  [data] = [ #---- ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > 444
缓冲区状态：[buffer.in] = 4,  [buffer.out] = 3,  [data] = [ ---#- ]

可选择操作：[1] 生产产品；[2] 退出进程；[3] 删除信号量和共享内存；
输入操作编号 > 1
输入产品内容 > bbb      <-- 被唤醒，完成产品"bbb"的生产
缓冲区状态：[buffer.in] = 1,  [buffer.out] = 1,  [data] = [ ##### ]
```

13）生产者 producer 2 选择操作3，删除共享内存及信号量，并结束进程。

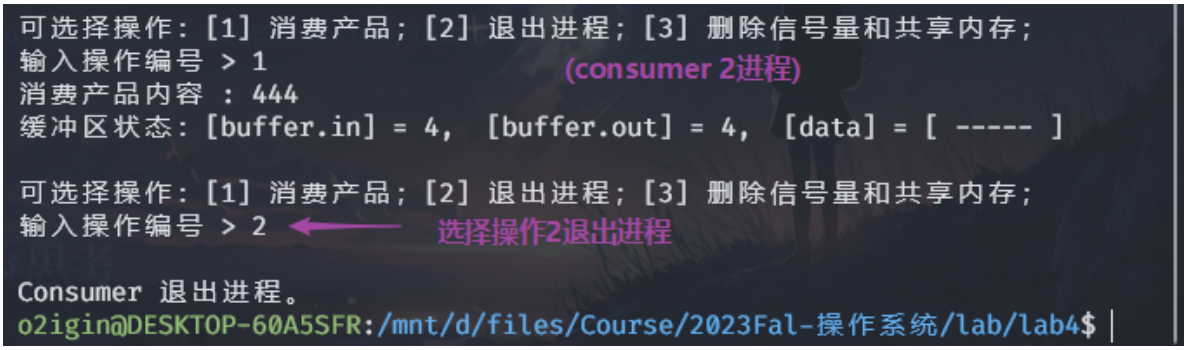
```
缓冲区状态: [buffer.in] = 3,  [buffer.out] = 4,  [data] = [ ###-# ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
输入产品内容 > 999                                  (producer 2进程)
缓冲区状态: [buffer.in] = 4,  [buffer.out] = 4,  [data] = [ ##### ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
输入产品内容 > aaa
缓冲区状态: [buffer.in] = 0,  [buffer.out] = 0,  [data] = [ ##### ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 3  <-- 选择操作3,
                      删除信号量和共享内存，并结束进程。
Producer 退出进程。
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$
```

14）生产者**producer 1**选择操作1，**尝试**生产产品 `ccc` 。

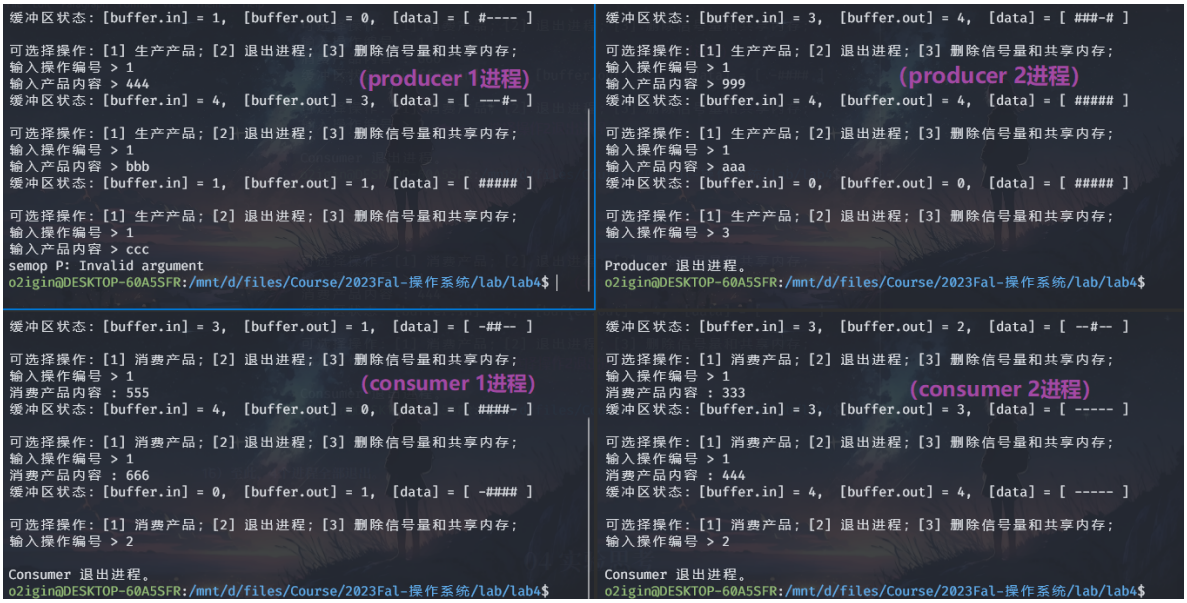
```
缓冲区状态: [buffer.in] = 1,  [buffer.out] = 0,  [data] = [ #---- ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
输入产品内容 > 444                                  (producer 1进程)
缓冲区状态: [buffer.in] = 4,  [buffer.out] = 3,  [data] = [ ---#- ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
输入产品内容 > bbb
缓冲区状态: [buffer.in] = 1,  [buffer.out] = 1,  [data] = [ ##### ]

可选择操作: [1] 生产产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
输入产品内容 > ccc  <-- 尝试生产"ccc"，但是由于信号量和共享内存已被producer 2进程
                        的操作3删除，因而出现异常，进程退出。
semop P: Invalid argument
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$
```

15）消费者 `consumer 1` 、 `consumer 2` 均选择操作2，退出进程。

```
可选择操作: [1] 消费产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 1
消费产品内容 : 666                                  (consumer 1进程)
缓冲区状态: [buffer.in] = 0,  [buffer.out] = 1,  [data] = [ -#### ]

可选择操作: [1] 消费产品; [2] 退出进程; [3] 删除信号量和共享内存;
输入操作编号 > 2  <-- 选择操作2退出进程

Consumer 退出进程。
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$
```



16）至此，4个进程全部退出。



04 实验总结

本实验的难点在于：

- 缓冲区中循环队列的存取逻辑；
- 使用linux的共享内存与信号量接口；
- 生产者与消费者访问资源时的信号量逻辑；

使用如下方式可查看信号量数组的数值，有助于实验调试：

```
# 在终端中查看信号量数组的id
$ ipcs

------ Message Queues --------
```

```
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 24         o2igin     777        5132       1          dest
0x0000004b 34         o2igin     777        5132       1

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x0000004b 41         o2igin     777        3

# 由此可知创建的 key=75（0x4b）的信号量数组的id为41

# 而后查看此信号量数组的具体信息
$ ipcs -s -i 41

Semaphore Array semid=41
uid=1000         gid=1000        cuid=1000       cgid=1000
mode=0777, access_perms=0777
nsems = 3
otime = Not set
ctime = Tue Oct 17 16:28:52 2023
semnum     value      ncount     zcount     pid
0          1          0          0          4219
1          5          0          0          4219
2          0          0          0          4219

# 可见:
# 0号信号量的值为1 （用于共享内存的互斥访问，为初值）
# 1号信号量的值为5 （维护大小为5的缓冲区，这里为空缓冲区个数初值）
# 2号信号量的值为0 （为满缓冲区个数初值）
```