

# 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，每块缓存可存入文本；
- 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。

```

1  #define BUFFER_CELL_SIZE 1024
2  #define BUFFER_CELL_NUM 5
3
4  // 一个缓存块
5  struct BufferCell
6  {
7      char text[BUFFER_CELL_SIZE];
8  };

```

利用多个缓存块组成缓存区结构。

```

1  // 缓存区，包含若干缓存块
2  // 从数据结构角度，实际即循环队列
3  struct Buffer
4  {
5      int in, out, n; // 队列尾，队列头，队列空间大小
6      BufferCell data[BUFFER_CELL_NUM];
7
8      // 将tmp指向的文本装入Buffer队尾
9      void push_back(char *tmp)
10     {
11         // 1. 装入数据
12         strncpy(data[in].text, tmp, (BUFFER_CELL_SIZE-1) * sizeof(char));
13         data[in].text[BUFFER_CELL_SIZE - 1] = '\0';
14         // 2. 更新指针
15         in++;
16         in %= BUFFER_CELL_NUM;
17         n++;
18     }
19
20     // 将Buffer队头的元素移出，并将其内容拷贝到tmp
21     void pop_front(char *tmp)
22     {
23         // 1. 拷贝出数据到data
24         strncpy(tmp, data[out].text, (BUFFER_CELL_SIZE-1) * sizeof(char));
25         tmp[BUFFER_CELL_SIZE - 1] = '\0';
26         // 2. 更新指针
27         out++;
28         out %= BUFFER_CELL_NUM;
29         n--;
30     }
31
32     // 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）
33     void print()

```

```

34     {
35         // 代码略，详见实验源代码
36         // 可实现如下打印效果：（'#'为满缓冲区，`-`为空缓冲区）
37         // 缓冲区状态：[buffer.in] = 1, [buffer.out] = 0, [data] = [ #---- ]
38     }
39 };

```

## 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：

```

1  #include <iostream>
2  #include <unistd.h>
3  #include <sys/sem.h>
4  // 初始化信号量集semid中第index个信号量，赋值为initial_value
5  void initialize_semaphores(int semid,int index,int initial_value)
6  {
7      // 使用 semctl 进行初始化
8      if (semctl(semid, index, SETVAL, initial_value) == -1)
9      {
10         perror("semctl");
11         exit(EXIT_FAILURE);
12     }
13 }
14
15 // 执行 P 操作
16 // 对semid的信号量数组的第semaphore_num信号量操作
17 void semaphore_P(int semid, int semaphore_num)
18 {

```

```

19     struct sembuf operation;
20     operation.sem_num = semaphore_num;
21     operation.sem_op = -1; // P 操作
22     operation.sem_flg = 0;
23     if (semop(semid, &operation, 1) == -1)
24     {
25         perror("semop P");
26         exit(EXIT_FAILURE);
27     }
28 }
29
30 // 执行 V 操作
31 // 对semid的信号量数组的第semaphore_num信号量操作
32 void semaphore_V(int semid, int semaphore_num)
33 {
34     struct sembuf operation;
35     operation.sem_num = semaphore_num;
36     operation.sem_op = 1; // V 操作
37     operation.sem_flg = 0;
38     if (semop(semid, &operation, 1) == -1)
39     {
40         perror("semop V");
41         exit(EXIT_FAILURE);
42     }
43 }

```

## 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）；

```

1 // 获取信号量（3个），赋初值，返回其id
2 // 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
3 int get_semaphore(int key, int create = 1)
4 {
5     // 本实验共需3个信号量
6     int num_semaphores = 3;
7     // 尝试获取，如果不存在则返回-1
8     int t_semid = semget(key, num_semaphores, 0777);
9     // 如果返回-1，则说明需要创建此信号量
10    int isInitial = t_semid == -1;
11    // 消费者调用此函数时，create=0，不能由消费者创建信号量
12    if (!create && isInitial)
13    {
14        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);
15        exit(1);
16    }
17    // 尝试创建信号量
18    if ((t_semid = semget(key, num_semaphores, 0777 | IPC_CREAT)) == -1)
19    {
20        printf("Semaphore Array Creat Failed!\n");
21        exit(1);
22    }
23    // 如果是首次创建信号量，则进行初始化
24    if (isInitial)
25    {
26        printf("初次创建 KEY=%d 信号量数组，信号量初始化.....\n", key);
27        // a. 将第0个信号量，初值设为1（用于缓冲区互斥访问）
28        initialize_semaphores(t_semid, 0, 1);
29        // b. 将第1个信号量，初值设为 BUFFER_CELL_NUM（EMPTY缓冲区个数）
30        initialize_semaphores(t_semid, 1, BUFFER_CELL_NUM);
31        // c. 将第2个信号量，初值设为0（FULL 缓冲区个数）
32        initialize_semaphores(t_semid, 2, 0);
33    }
34    else
35        printf("已存在 KEY=%d 的信号量，直接获取.....\n", key);
36
37    return t_semid;
38 }

```

## 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：

```
1 // 获得共享内存区，进行初始化，返回其id
2 int get_share_memory(int key, int create = 1)
3 {
4     // 尝试获取，如果不存在则返回-1
5     int t_shmid = shmget(key, sizeof(Buffer), 0777);
6     // 如果返回-1，则说明需要创建此共享内存
7     int isInitial = t_shmid == -1;
8     // 消费者调用此函数时，create=0，不能由消费者创建共享内存
9     if (!create && isInitial)
10    {
11        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);
12        exit(1);
13    }
14    // 尝试创建共享内存
15    if ((t_shmid = shmget(key, sizeof(Buffer), 0777 | IPC_CREAT)) == -1)
16    {
17        printf("Share Memory Creat Failed!\n");
18        exit(1);
19    }
20    // 获得共享内存起始位置指针（转为BufferCell指针，方便存取操作）
21    buffer = (Buffer *)shmat(t_shmid, 0, 0);
22    // 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）
23    if (isInitial)
```

```

24     {
25         printf("初次创建 KEY=%d 的共享内存, 共享内存初始化.....\n", key);
26         memset(buffer, 0, sizeof(Buffer));
27     }
28     else
29         printf("已存在 KEY=%d 的共享内存, 直接获取.....\n", key);
30
31     return t_shmid;
32 }

```

## 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：

```

1 // 操作3，删除信号量和共享内存；
2 void delete_sem_and_shm()
3 {
4     // 删除信号量集
5     if (semctl(semid, 3, IPC_RMID) == -1)
6     {
7         perror("semctl IPC_RMID");
8         exit(EXIT_FAILURE);
9     }
10
11     // 删除共享内存
12     if (shmctl(shmid, IPC_RMID, 0) == -1)
13     {
14         perror("shmctl IPC_RMID");
15         exit(EXIT_FAILURE);
16     }
17 }

```

## 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：

```
1 // 操作1，生产产品
2 void produce()
3 {
4     // 读入生产产品
5     printf("输入产品内容 > ");
6     scanf("%s", buf);
7
8     semaphore_P(semid, 1); // 申请空缓冲区
9     semaphore_P(semid, 0); // 申请缓冲区互斥操作权限
10
11     // 进行缓冲区内互斥操作，装入产品
12     buffer->push_back(buf);
13
14     semaphore_V(semid, 0); // 释放缓冲区控制权限
15     semaphore_V(semid, 2); // 增加满缓冲区个数
16
17     // 输出操作后缓冲区状态
18     buffer->print();
19 }
```

## 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。

```
1 // 操作1，消费产品
2 void consume()
3 {
4     // 申请满缓冲区
5     semaphore_P(semid, 2);
6     // 申请缓冲区互斥操作权限
7     semaphore_P(semid, 0);
8
9     // 进行缓冲区内互斥操作，装入产品
10    buffer->pop_front(buf);
11 }
```



```

12 // 释放缓冲区控制权限
13 semaphore_V(semid, 0);
14 // 增加空缓冲区个数
15 semaphore_V(semid, 1);
16
17 // 输出消费产品内容
18 printf("消费产品内容 : %s\n", buf);
19 // 输出操作后缓冲区状态
20 buffer->print();
21 }

```

## 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：

```

1  int main(int argc, char *argv[])
2  {
3      // 获取共享内存区
4      shmid = get_share_memory(SHMKEY, 0); // 若为生产者，则第二个参数为1
5      // 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
6      semid = get_semaphore(SEMKEY, 0); // 若为生产者，则第二个参数为1
7      // 输出缓冲区初始状态
8      buffer->print();
9      // 进入消费者操作循环
10     while (1)
11     {
12         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"
13             "输入操作编号 > ");
14         scanf("%s", buf);
15         if (buf[0] == '1')
16             consume(); // 若为生产者，则改为上述定义的produce()函数
17         else if (buf[0] == '2')
18             break;
19         else if (buf[0] == '3')

```

```
20     {
21         delete_sem_and_shm();
22         break;
23     }
24     else
25         printf("输入格式操作! 重新输入.....\n");
26     printf("\n");
27 }
28 printf("\nConsumer 退出进程.\n");
29 return 0;
30 }
```

## 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；

```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
producer
初次创建 KEY=75 的共享内存，共享内存初始化.....
初次创建 KEY=75 信号量数组，信号量初始化.....
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 > |
```

第一个生产者进程启动

创建信号量和共享内存

缓冲区填充状态, '#表示满, '-'表示空;

指针位置

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。

```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
producer
初次创建 KEY=75 的共享内存，共享内存初始化.....
初次创建 KEY=75 信号量数组，信号量初始化.....
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 > 1
输入产品内容 > 111
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 > |
```

选择操作1, 生产产品"111";

在0位置放入新元素, in指针移动到1;

缓冲区0位置变为'#', 表示变为满缓冲块;

3) 启动第2个 producer 进程, 如下图所示。

```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
producer ← 启动第2个生产者进程
已存在 KEY=75 的共享内存，直接获取..... ← 发现已经存在对应的共享内存和信号量，
已存在 KEY=75 的信号量，直接获取..... 直接获取；
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 > |

启动后，显示缓冲区状态，与步骤 2) 中操作后显示状态相同；
缓冲区中已有1个满缓冲区 (在0位置)；

[3] 删除信号量和共享内存;

ut] = 0, [data] = [ 2---- ]

[3] 删除信号量和共享内存;
```

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

```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
producer (此进程为 producer 2)
已存在 KEY=75 的共享内存，直接获取.....
已存在 KEY=75 的信号量，直接获取.....
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 ← 生产产品“222”
输入产品内容 > 222
缓冲区状态: [buffer.in] = 2, [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 ← 生产产品“333”
输入产品内容 > 333
缓冲区状态: [buffer.in] = 3, [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 > | 至此，缓冲区0、1、2已装填产品
```

5) 启动第1个消费者 consumer 1，如下图所示。

```

o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
consumer
已存在 KEY=75 的共享内存，直接获取.....
已存在 KEY=75 的信号量，直接获取.....
缓冲区状态: [buffer.in] = 3, [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 > |

```

启动第一个消费者

已存在共享内存和信号量，  
直接获取;

缓冲区中已存在3个满缓冲块;

6) consumer 1 选择操作1，消费一个产品，打印到屏幕。

```

o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
consumer
已存在 KEY=75 的共享内存，直接获取.....
已存在 KEY=75 的信号量，直接获取.....
缓冲区状态: [buffer.in] = 3, [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 : 111
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 > |

```

consumer1选择操作1，消费一个产品;

得到第一个生产的产品“111”;

消费后，out指针向后移动一个单位，  
指向下一个满缓冲区位置

0位置产品被消费，变为'-',  
1、2位置仍有产品;

7) 启动第二个消费者 consumer 2，如下图所示。



```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
consumer ← 启动第2个消费者
已存在 KEY=75 的共享内存，直接获取..... ← 已存在对应的共享内存和信号量，
已存在 KEY=75 的信号量，直接获取..... 直接获取
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 > |
```

缓冲区状态与上次操作后状态相同

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。

```
consumer
已存在 KEY=75 的共享内存，直接获取..... (consumer 2进程)
已存在 KEY=75 的信号量，直接获取.....
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 ← 第1次消费，输出“222”，位置1变为空
消费产品内容 : 222
缓冲区状态: [buffer.in] = 3, [buffer.out] = 2, [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 ← 第2次消费，输出“333”，位置2变为空
消费产品内容 : 333
缓冲区状态: [buffer.in] = 3, [buffer.out] = 3, [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 ← 第3次消费，此时缓冲区全空，无满缓冲块资源
| ← 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。

```

o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
producer
初次创建 KEY=75 的共享内存，共享内存初始化..... (producer 1进程)
初次创建 KEY=75 信号量数组，信号量初始化.....
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 > 1
输入产品内容 > 111
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 ← 生产产品“444”，第3个缓冲块变为满
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 > |

```

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。

```

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 : 222
缓冲区状态: [buffer.in] = 3, [buffer.out] = 2, [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 : 333
缓冲区状态: [buffer.in] = 3, [buffer.out] = 3, [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 ← 从阻塞状态被唤醒，消费产品“444”，
消费产品内容 : 444 消费后缓冲区再次变为全空。
缓冲区状态: [buffer.in] = 4, [buffer.out] = 4, [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 > |

```

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。

```

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
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

```

第4次生产，产品为“888”

第5次生产，产品为“999”，至此，缓冲区全满

第6次生产，产品为“aaa”，但是因此时无“空缓冲区”资源，因此此进程 producer 2 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；



```

初次创建 KEY=75 的共享内存，共享内存初始化.....
初次创建 KEY=75 信号量数组，信号量初始化.....
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 > 1
输入产品内容 > 111
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

```

(producer 1进程)

生产“bbb”，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；

```

o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4$ ./
consumer
已存在 KEY=75 的共享内存，直接获取.....
已存在 KEY=75 的信号量，直接获取.....
缓冲区状态: [buffer.in] = 3, [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 : 111
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] = [ #### - ]

```

(consumer1进程)

消费产品“555”，产生一个空缓冲区

```

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$ |

```



```
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 : 444
缓冲区状态: [buffer.in] = 4, [buffer.out] = 4, [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。

<pre>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 &gt; 1 输入产品内容 &gt; 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 &gt; 1 输入产品内容 &gt; 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 &gt; 1 输入产品内容 &gt; ccc semop P: Invalid argument o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4\$  </pre>	<pre>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 &gt; 1 输入产品内容 &gt; 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 &gt; 3 Producer 退出进程。 o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4\$</pre>
<pre>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 &gt; 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 &gt; 1 消费产品内容 : 666 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 &gt; 2 Consumer 退出进程。 o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4\$</pre>	<pre>缓冲区状态: [buffer.in] = 3, [buffer.out] = 2, [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 &gt; 1 消费产品内容 : 333 缓冲区状态: [buffer.in] = 3, [buffer.out] = 3, [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 &gt; 2 Consumer 退出进程。 o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab4\$</pre>

# 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:

```
1 # 在终端中查看信号量数组的id
2 $ ipcs
3
4 ----- Message Queues -----
```

```

5 key      msqid      owner      perms      used-bytes  messages
6
7 ----- Shared Memory Segments -----
8 key      shmid      owner      perms      bytes      nattch     status
9 0x00000000 24      o2igin     777        5132       1          dest
10 0x0000004b 34      o2igin     777        5132       1
11
12 ----- Semaphore Arrays -----
13 key      semid      owner      perms      nsems
14 0x0000004b 41      o2igin     777        3
15
16 # 由此可知创建的 key=75 (0x4b) 的信号量数组的id为41
17
18 # 而后查看此信号量数组的具体信息
19 $ ipcs -s -i 41
20
21 Semaphore Array semid=41
22 uid=1000      gid=1000      cuid=1000      cgid=1000
23 mode=0777, access_perms=0777
24 nsems = 3
25 otime = Not set
26 ctime = Tue Oct 17 16:28:52 2023
27 semnum      value      ncount      zcount      pid
28 0           1          0           0           4219
29 1           5          0           0           4219
30 2           0          0           0           4219
31
32 # 可见:
33 # 0号信号量的值为1 (用于共享内存的互斥访问, 为初值)
34 # 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)
35 # 2号信号量的值为0 (为满缓冲区个数初值)

```