|  |
| --- |
| 北京邮电大学 |
| 进程同步控制 |
| 操作系统 |

|  |
| --- |
| 姓名：陈朴炎 学号：2021211138  2023-11-30 |

目录

[1 实验内容 2](#_Toc152358021)

[1.1 实验介绍 2](#_Toc152358022)

[1.2 实验要求 3](#_Toc152358023)

[1.3 实验提交要求 3](#_Toc152358024)

[2 环境搭配 4](#_Toc152358025)

[2.1 OpenEuler虚拟机安装 4](#_Toc152358026)

[2.2 vscode 远程连接虚拟机、 5](#_Toc152358027)

[3 实验步骤 6](#_Toc152358028)

[4 程序设计 8](#_Toc152358029)

[4.1定义全局变量 8](#_Toc152358030)

[4.2 函数定义 8](#_Toc152358031)

[4.2.1 insert\_item() 8](#_Toc152358032)

[4.2.2 remove\_item() 8](#_Toc152358033)

[4.2.3 main() 主函数 9](#_Toc152358034)

[4.3 线程定义 10](#_Toc152358035)

[4.3.1 生产者线程工作内容 10](#_Toc152358036)

[4.3.2 消费者线程工作内容 11](#_Toc152358037)

[5 测试报告 11](#_Toc152358038)

[5.1 测试用例1 11](#_Toc152358039)

[5.2 测试用例2 14](#_Toc152358040)

[5.3 测试用例3 14](#_Toc152358041)

[6 源码 15](#_Toc152358042)

# 1 实验内容

## 1.1 实验介绍

在第6.6.1节中，我们提出了一种基于信号量的解决方案，使用有界缓冲区解决生产者-消费者问题。在这个项目中，我们将设计一个编程解决方案来解决有界缓冲区问题，使用图6.10和6.11中显示的生产者和消费者进程。第6.6.1节中提出的解决方案使用了三个信号量：empty和full，分别计算缓冲区中空槽和满槽的数量，以及mutex，这是一个二进制（或互斥）信号量，用于保护在缓冲区中插入或删除项。对于这个项目，将使用标准计数信号量来表示empty和full，而不是使用二进制信号量，将使用mutex锁来表示mutex。生产者和消费者作为独立的线程运行，将项目移动到与这些empty、full和mutex结构同步的缓冲区中。

在内部，缓冲区将由buffer\_item类型的固定大小数组组成（将使用typedef def进行定义）。buffer\_item对象数组将作为循环队列进行操作。

缓冲区将通过两个函数insert\_item()和remove\_item()进行操作，这两个函数分别由生产者和消费者线程调用。

insert\_item()和remove\_item()函数将使用图6.10和6.11中概述的算法同步生产者和消费者。缓冲区还将需要一个初始化函数，该函数初始化互斥对象mutex以及empty和full信号量。

main()函数将初始化缓冲区并创建独立的生产者和消费者线程。创建生产者和消费者线程后，main()函数将休眠一段时间，在唤醒后终止应用程序。main()函数将在命令行上传递三个参数：

（1）在终止前休眠的时间

（2）生产者线程的数量

（3）消费者线程的数量

生产者线程将在随机时间间隔内交替休眠，并将随机整数插入缓冲区。将使用rand()函数生成随机数，该函数生成0到RAND\_MAX之间的随机整数。消费者线程也将在随机时间间隔内休眠，并在唤醒后尝试从缓冲区中移除一个项。

## 1.2 实验要求

(1) 缓冲区

(a) 缓冲区存储结构建议采用固定大小的数组表示，并作为环形队列处理。 (b) 缓冲区的访问算法按照课本 6.6.1 节图 6.10、图 6.11 进行设计。

(2) 主函数 main()

(a) 主函数需要创建一定数量的生产者线程与消费者线程。线程创建完毕 后，主函数将睡眠一段时间，并在唤醒时终止应用程序。

(b) 主函数需要从命令行接受三个参数：睡眠时长、生产者线程数量、消 费者线程数量。

(3) 生产者与消费者线程

(a) 生产者线程：随机睡眠一段时间，向缓冲区插入一个随机数。

(b) 消费者线程：随机睡眠一段时间，从缓冲区去除一个随机数。

## 1.3 实验提交要求

1）实验内容

2）程序设计：

（a）用到的API；

（b）程序设计说明

3）测试报告：

（a）测试用例（输入数据）；

（b）运行结果截图；

（c）测试结果分析。

# 2 环境搭配

本次实验在Windows环境下，用VM Ware Workstation pro软件仿真openEuler-20.03-LTS-x86\_64虚拟环境。通过Windows下的VSCode远程连接openEuler虚拟机来完成实验。

## 2.1 OpenEuler虚拟机安装

首先下载VM Ware，可以到[VMware官方网站](https://www.vmware.com/cn/products/workstation-pro/workstation-pro-evaluation.html)进行下载安装。

接着下载openEuler的镜像影响光盘，可以到[镜像下载网站](https://www.openeuler.org/zh/download/)下载相应的镜像文件。

将openEuler-20.03-LTS-x86\_64-dvd.iso 文件放置到某个目录下，打开VMware，添加新的虚拟机，选择刚下好的.iso文件。再给这台虚拟机分配两个cpu，内存分配4GB，网络设置VMnet8 设置为NAT模式，通过本机DHCP分配IP地址。之后设置root用户和密码，完成openEuler的安装。

## 2.2 vscode 远程连接虚拟机、

由于openEuler只有命令行，并且我也不懂如何选中文字复制粘贴，所以我决定使用VScode辅助完成本次实验。

首先为VSCode安装 Remote SSH拓展

之后在openEuler虚拟机中，执行 vi /etc/ssh/sshd\_config 命令，修改该文件内容，进入之后界面如下

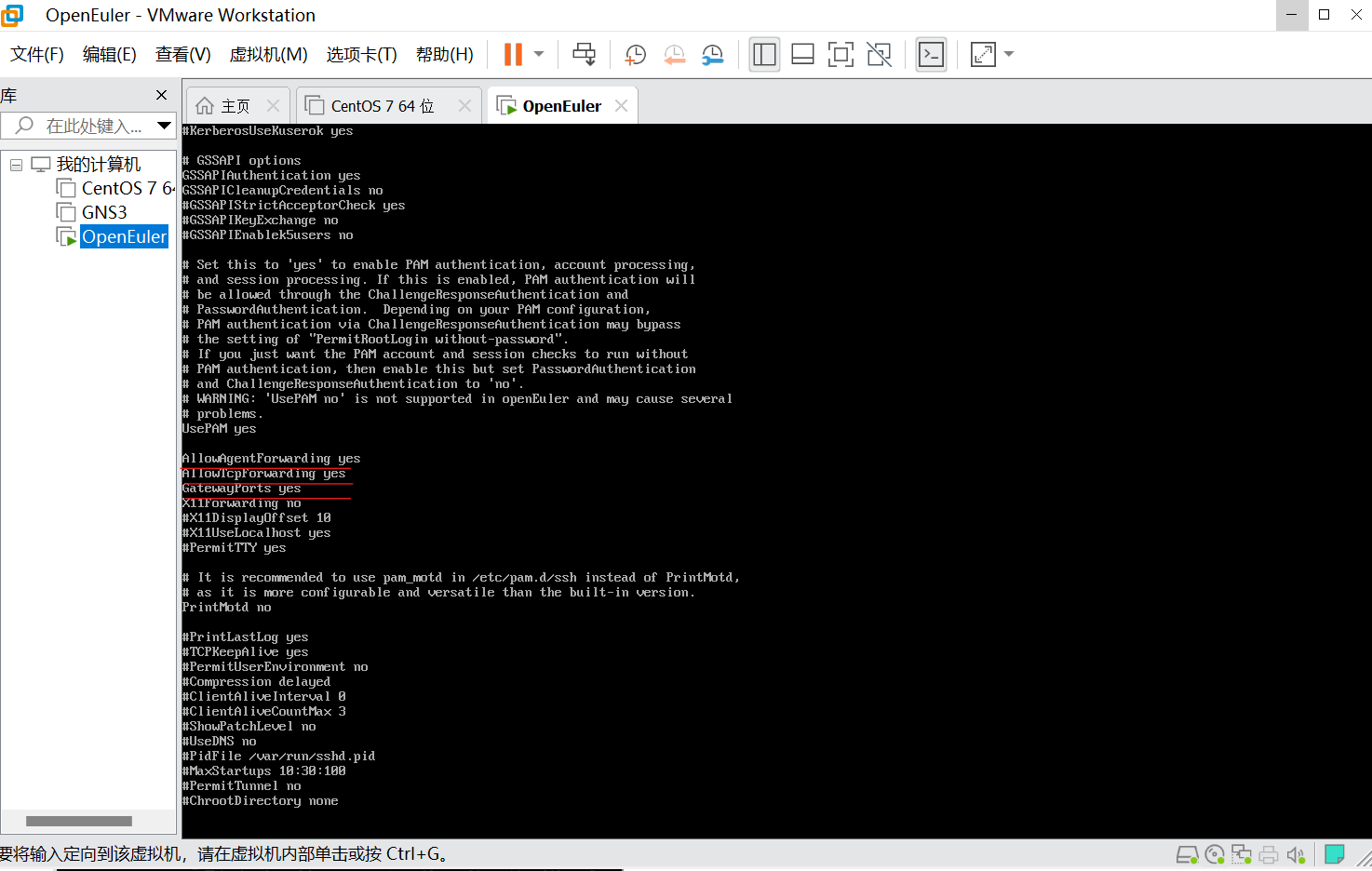


图2-1 修改sshd\_config文件示意图

将这三句话前面的#井号删除，并将后面的修饰符调整成yes，打开AllowAgentForwarding、AllowTcpForwarding、GatewayPorts，并保存文件。

之后输入命令 systemctl restart sshd.service，重新刷新sshd服务状态

输入ip addr查看IP地址，如下：

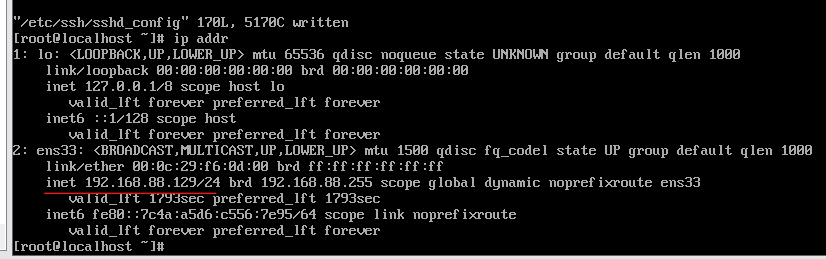


图2-2 查看ip地址示意图

在VScode中，新建终端，并输入ssh [root@192.168.88.129](mailto:root@192.168.88.129) (openEuler的ip地址)，进行远程连接。

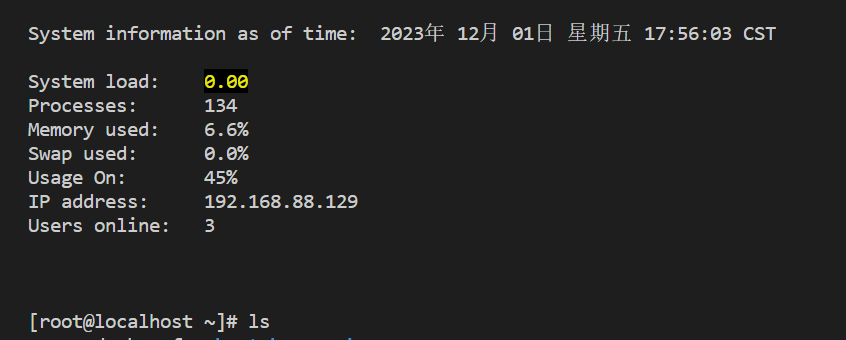


图2-3连接成功示意图

之后就可以在vscode的终端中进行实验了。

# 3 实验步骤

在某个文件夹下通过vi创建C文件：

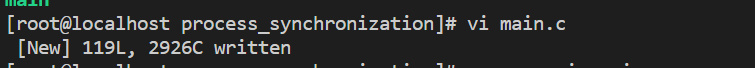


图3-1创建文件示意图

在vi中按下“i”键进入INSERT模式，写入程序源代码

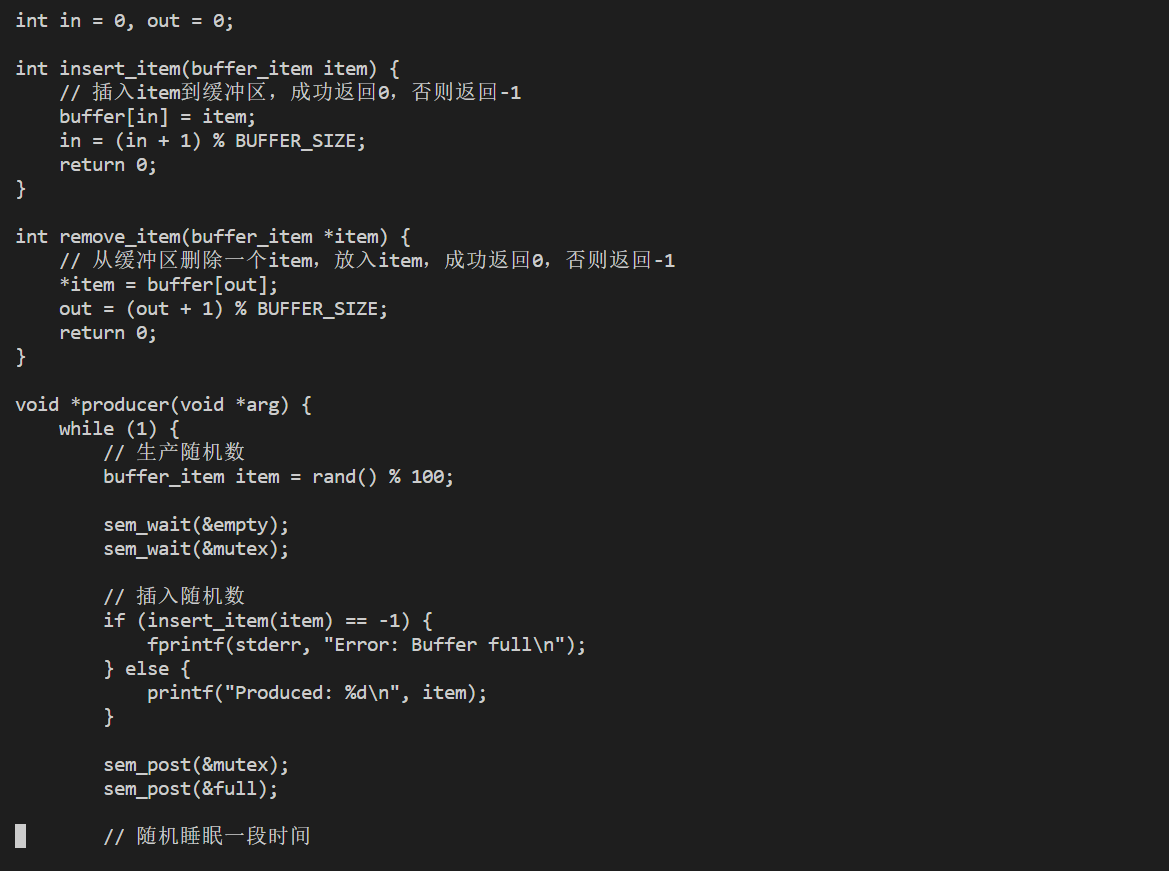


图3-2 在vi中写入文件示意图

完成程序写入后，按下ESC退出INSERT模式，再输入:wq保存并退出。

在文件目录下，将文件编译，由于需要线程和信号量的库，因此我们需要在编译阶段链接到这两个库中，调用库里的函数：-pthread -lrt

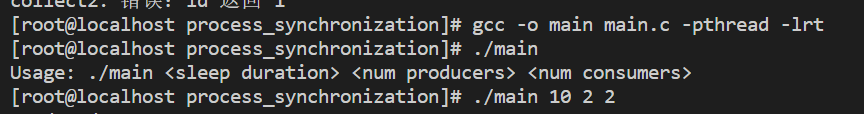


图3-3 编译链接及运行示意图

完成编译后，在命令行中输入./main <sleep duration> <num producers> <num comsumers> ，上例中我设置的主函数睡眠时间为10，生产者进程数2，消费者进程数2。

最后进行测试，并分析结果。

# 4 程序设计

## 4.1定义全局变量

|  |
| --- |
| sem\_t empty, full, mutex;  typedef int buffer\_item;  buffer\_item buffer[BUFFER\_SIZE];  int in = 0, out = 0; |

三个信号量：empty、full、mutex分别代表缓冲区有空闲、缓冲区有数据、缓冲区操作互斥锁。这三个信号量都是在main函数中初始化的，empty被初始化为缓冲区的大小，full被初始化为0，mutex被初始化为1。

in和out是循环数组的指针，代表生产者和消费者将要对哪个下标进行生产或者消费。当in==out的时候，循环数组已空，当in=(out+1)%BUFFER\_SIZE时，代表循环数组已满。

## 4.2 函数定义

### 4.2.1 insert\_item()

1. **int** insert\_item(buffer\_item item) {
2. // 插入item到缓冲区，成功返回0，否则返回-1
3. buffer[in] = item;
4. in = (in + 1) % BUFFER\_SIZE;
5. **return** 0;
6. }

### 4.2.2 remove\_item()

1. **int** remove\_item(buffer\_item \*item) {
2. // 从缓冲区删除一个item，放入item，成功返回0，否则返回-1
3. \*item = buffer[out];
4. out = (out + 1) % BUFFER\_SIZE;
5. **return** 0;
6. }

### 4.2.3 main() 主函数

1. // 解析命令行输入的信息
2. **if** (argc != 4) {
3. fprintf(stderr, "Usage: %s <sleep duration> <num producers> <num consumers>\n", argv[0]);
4. exit(EXIT\_FAILURE);
5. }
7. **int** sleep\_duration = atoi(argv[1]);
8. **int** num\_producers = atoi(argv[2]);
9. **int** num\_consumers = atoi(argv[3]);
11. // 初始化三个信号量
12. sem\_init(&empty, 0, BUFFER\_SIZE);
13. sem\_init(&full, 0, 0);
14. sem\_init(&mutex, 0, 1);
16. // 创建一定数量的生产者
17. pthread\_t producer\_threads[num\_producers];
18. **for** (**int** i = 0; i < num\_producers; ++i) {
19. pthread\_create(&producer\_threads[i], NULL, producer, NULL);
20. }
22. // 创建一定数量的消费者
23. pthread\_t consumer\_threads[num\_consumers];
24. **for** (**int** i = 0; i < num\_consumers; ++i) {
25. pthread\_create(&consumer\_threads[i], NULL, consumer, NULL);
26. }
28. // 主函数休息一会
29. sleep(sleep\_duration);
31. // 在主函数醒来时终结这些线程
32. **for** (**int** i = 0; i < num\_producers; ++i) {
33. pthread\_cancel(producer\_threads[i]);
34. }
36. **for** (**int** i = 0; i < num\_consumers; ++i) {
37. pthread\_cancel(consumer\_threads[i]);
38. }
40. // 销毁信号量
41. sem\_destroy(&empty);
42. sem\_destroy(&full);
43. sem\_destroy(&mutex);

主函数中，首先解析了从命令行传入的参数：主函数睡眠时间、生产者线程数量、消费者线程数量。接着初始化了三个信号量。在创建完相应数量的生产者、消费者线程后，主函数线程进入休眠状态。当时间到时，主函数将所有子线程销毁，并将信号量也销毁。

### 4.3 线程定义

### 4.3.1 生产者线程工作内容

1. **void** \*producer(**void** \*arg) {
2. **while** (1) {
3. // 生产随机数
4. buffer\_item item = rand() % 100;
6. sem\_wait(&empty);
7. sem\_wait(&mutex);
9. // 插入随机数
10. **if** (insert\_item(item) == -1) {
11. fprintf(stderr, "Error: Buffer full\n");
12. } **else** {
13. printf("Produced: %d\n", item);
14. }
16. sem\_post(&mutex);
17. sem\_post(&full);
19. // 随机睡眠一段时间
20. usleep(rand() % 1000000);
21. }
22. }

生产者首先生成一个随机数，然后等待缓冲区有空闲状态，再等待缓冲区的互斥锁，之后将生产的随机数插入到循环数组中。接着释放信号量，并进入到休眠状态中。

### 4.3.2 消费者线程工作内容

1. **void** \*consumer(**void** \*arg) {
2. **while** (1) {
3. sem\_wait(&full);
4. sem\_wait(&mutex);
6. buffer\_item item;
8. // 从缓冲区删除一个数
9. **if** (remove\_item(&item) == -1) {
10. fprintf(stderr, "Error: Buffer empty\n");
11. } **else** {
12. printf("Consumed: %d\n", item);
13. }
15. sem\_post(&mutex);
16. sem\_post(&empty);
18. // 随机睡眠一段时间
19. usleep(rand() % 1000000);
20. }
21. }

消费者等待缓冲区有内容，再等待获取mutex信号量，当能够对缓冲区操作时，消费者获取out指针指向的缓冲区元素，并打印。最后将两个信号量释放。

# 5 测试报告

## 5.1 测试用例1

主函数睡眠时间10s，生产者2个，消费者2个

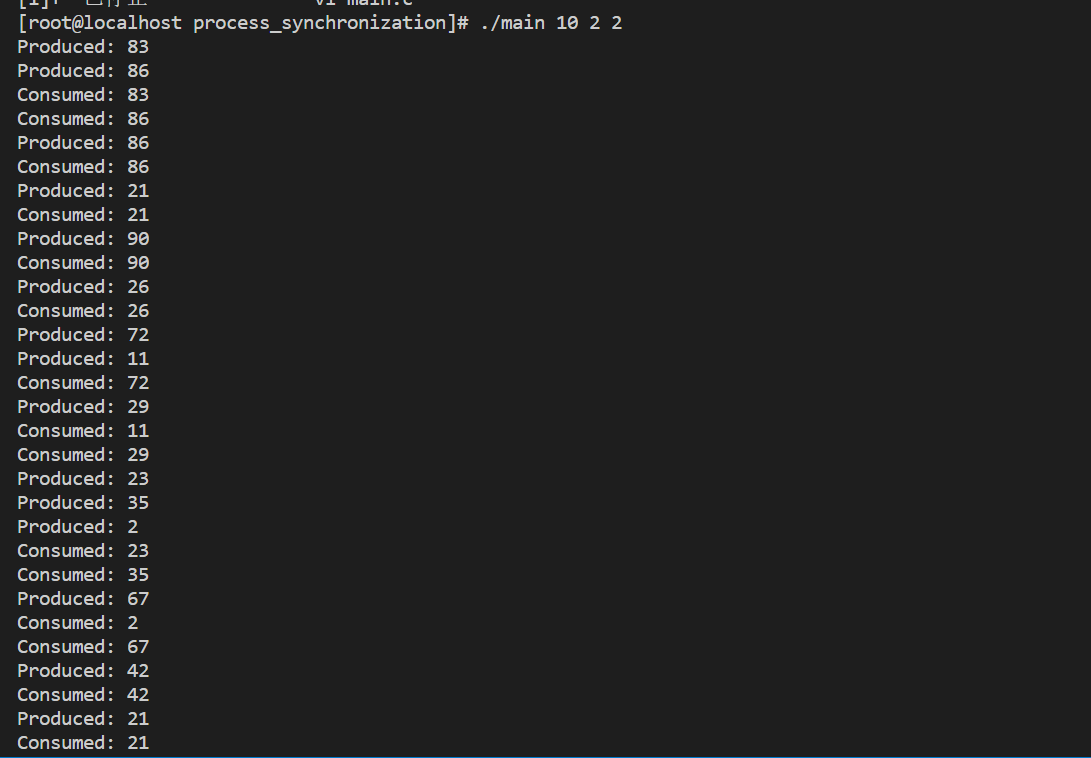


图5-1测试用例1执行结果图1



图5-2测试用例1执行结果图2



图5-3 测试用例1执行结果图3

结果分析：

Produced: XX 表示生产者生产了一个项目，其中 XX 是生成的随机数。

Consumed: XX 表示消费者从缓冲区中取出并消费了一个项目，其中 XX 是被消费的项目的值。

根据输出结果可以分析出：

初始阶段，两个生产者和两个消费者开始运行。

生产者和消费者的执行是交错的，即它们交替执行。

当生产者产生一个项目时，它将其放入缓冲区，并打印 "Produced" 消息。

当消费者从缓冲区中取出并消费一个项目时，它将其从缓冲区中移除，并打印 "Consumed" 消息。

观察到生产者和消费者的行为，以及它们在缓冲区上的交互。在这个测试用例中，每个生产者和消费者都在各自的线程中运行，它们通过信号量来同步对共享缓冲区的访问，以确保生产者不会在缓冲区已满时继续生产，消费者也不会在缓冲区为空时继续消费。

## 5.2 测试用例2

睡眠时长5 生产者数量5 消费者数量1

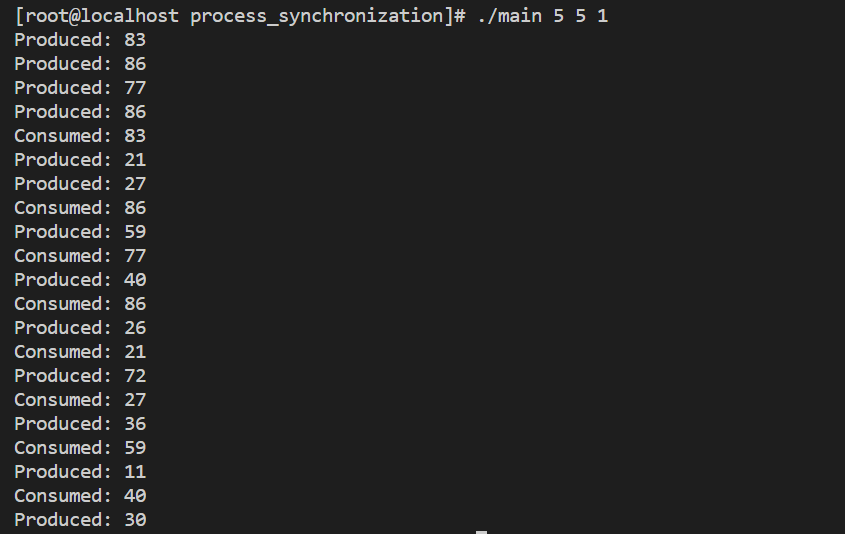


图5-4 测试用例2执行结果图

结果分析：

在这个测试用例中，由于生产者数量较多，总体的生产速度较快，可能导致缓冲区在某一时刻被填满，此时生产者需要等待消费者释放空槽位。这种交互性是由信号量 empty 和 full 的控制来确保的。empty 信号量确保在缓冲区有空槽位之前，生产者不会继续生产；而 full 信号量确保在缓冲区有项目之前，消费者不会继续消费。

## 5.3 测试用例3

睡眠时长5 生产者1 消费者5

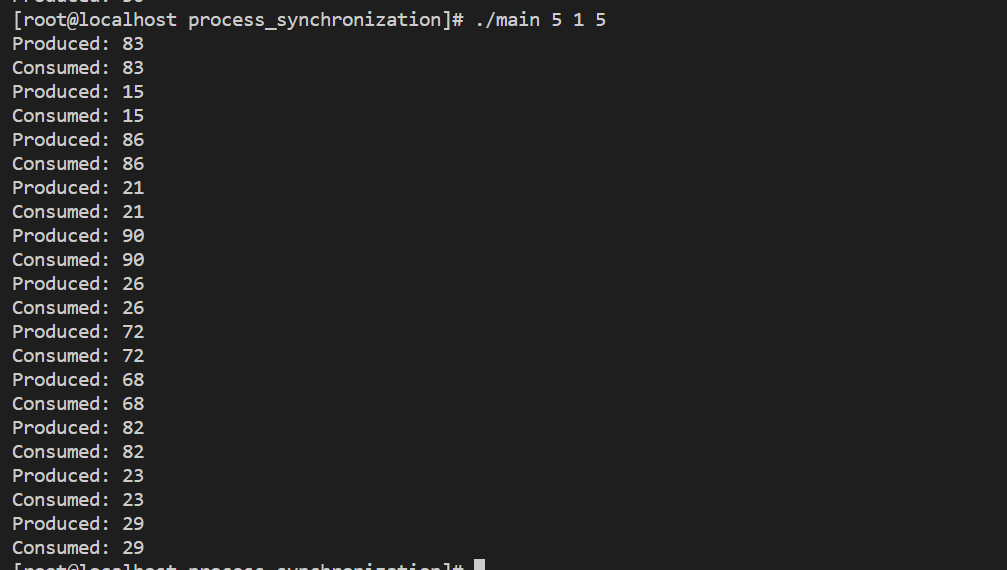


图5-5 测试用例2 执行结果图

结果分析：

在这个测试用例中，由于消费者的速度较快，导致缓冲区在某一时刻被完全清空，此时消费者需要等待生产者生产新的项目。这种交互性是由信号量 empty 和 full 的控制来确保的。empty 信号量确保在缓冲区有项目之前，消费者不会继续消费；而full 信号量确保在缓冲区有空槽位之前，生产者不会继续生产。

# 6 源码

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <pthread.h>
5. #include <semaphore.h>
7. #define BUFFER\_SIZE 5
9. sem\_t empty, full, mutex;
10. **typedef** **int** buffer\_item;  // 实验指导的定义
11. buffer\_item buffer[BUFFER\_SIZE];
12. **int** in = 0, out = 0;
14. **int** insert\_item(buffer\_item item) {
15. // 插入item到缓冲区，成功返回0，否则返回-1
16. buffer[in] = item;
17. in = (in + 1) % BUFFER\_SIZE;
18. **return** 0;
19. }
21. **int** remove\_item(buffer\_item \*item) {
22. // 从缓冲区删除一个item，放入item，成功返回0，否则返回-1
23. \*item = buffer[out];
24. out = (out + 1) % BUFFER\_SIZE;
25. **return** 0;
26. }
28. **void** \*producer(**void** \*arg) {
29. **while** (1) {
30. // 生产随机数
31. buffer\_item item = rand() % 100;
33. sem\_wait(&empty);
34. sem\_wait(&mutex);
36. // 插入随机数
37. **if** (insert\_item(item) == -1) {
38. fprintf(stderr, "Error: Buffer full\n");
39. } **else** {
40. printf("Produced: %d\n", item);
41. }
43. sem\_post(&mutex);
44. sem\_post(&full);
46. // 随机睡眠一段时间
47. usleep(rand() % 1000000);
48. }
49. }
51. **void** \*consumer(**void** \*arg) {
52. **while** (1) {
53. sem\_wait(&full);
54. sem\_wait(&mutex);
56. buffer\_item item;
58. // 从缓冲区删除一个数
59. **if** (remove\_item(&item) == -1) {
60. fprintf(stderr, "Error: Buffer empty\n");
61. } **else** {
62. printf("Consumed: %d\n", item);
63. }
65. sem\_post(&mutex);
66. sem\_post(&empty);
68. // 随机睡眠一段时间
69. usleep(rand() % 1000000);
70. }
71. }
73. **int** main(**int** argc, **char** \*argv[]) {
74. // 解析命令行输入的信息
75. **if** (argc != 4) {
76. fprintf(stderr, "Usage: %s <sleep duration> <num producers> <num consumers>\n", argv[0]);
77. exit(EXIT\_FAILURE);
78. }
80. **int** sleep\_duration = atoi(argv[1]);
81. **int** num\_producers = atoi(argv[2]);
82. **int** num\_consumers = atoi(argv[3]);
84. // 初始化三个信号量
85. sem\_init(&empty, 0, BUFFER\_SIZE);
86. sem\_init(&full, 0, 0);
87. sem\_init(&mutex, 0, 1);
89. // 创建一定数量的生产者
90. pthread\_t producer\_threads[num\_producers];
91. **for** (**int** i = 0; i < num\_producers; ++i) {
92. pthread\_create(&producer\_threads[i], NULL, producer, NULL);
93. }
95. // 创建一定数量的消费者
96. pthread\_t consumer\_threads[num\_consumers];
97. **for** (**int** i = 0; i < num\_consumers; ++i) {
98. pthread\_create(&consumer\_threads[i], NULL, consumer, NULL);
99. }
101. // 主函数休息一会
102. sleep(sleep\_duration);
104. // 在主函数醒来时终结这些线程
105. **for** (**int** i = 0; i < num\_producers; ++i) {
106. pthread\_cancel(producer\_threads[i]);
107. }
109. **for** (**int** i = 0; i < num\_consumers; ++i) {
110. pthread\_cancel(consumer\_threads[i]);
111. }
113. // 销毁信号量
114. sem\_destroy(&empty);
115. sem\_destroy(&full);
116. sem\_destroy(&mutex);
118. **return** 0;
119. }