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

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

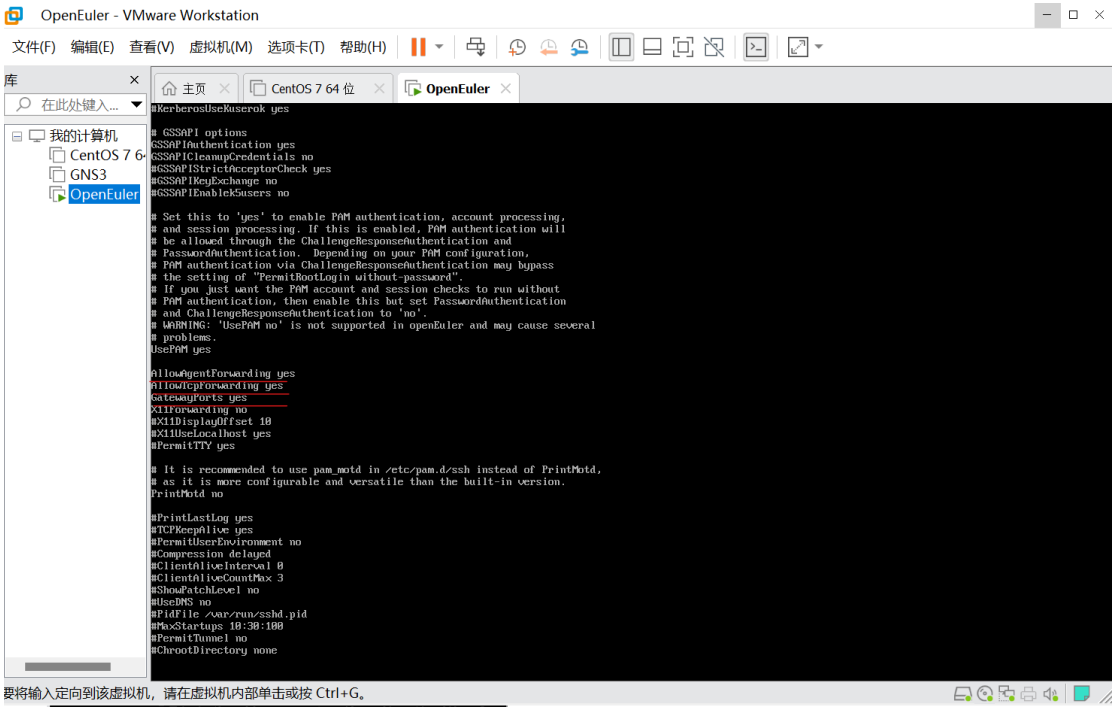

图 2-1 修改 sshd_config 文件示意图

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：

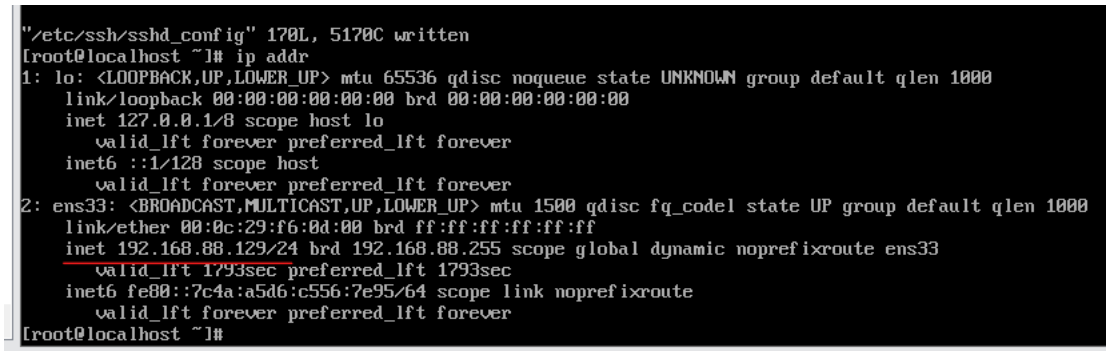

图 2-2 查看 ip 地址示意图

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

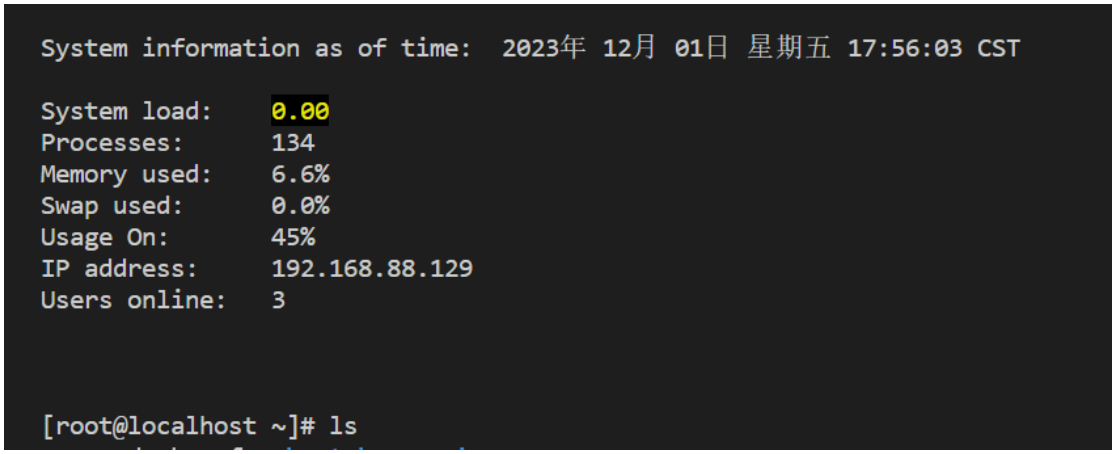

图 2-3 连接成功示意图

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：

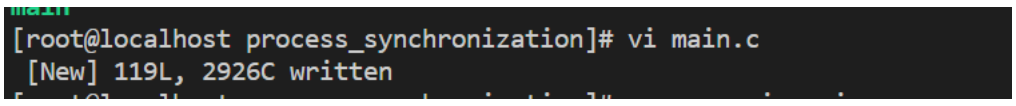

图 3-1 创建文件示意图

在 vi 中按下“i”键进入 INSERT 模式，写入程序源代码

```
int in = 0, out = 0;

int insert_item(buffer_item item) {
    // 插入item到缓冲区，成功返回0，否则返回-1
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    return 0;
}

int remove_item(buffer_item *item) {
    // 从缓冲区删除一个item，放入item，成功返回0，否则返回-1
    *item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return 0;
}

void *producer(void *arg) {
    while (1) {
        // 生产随机数
        buffer_item item = rand() % 100;

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

        // 插入随机数
        if (insert_item(item) == -1) {
            fprintf(stderr, "Error: Buffer full\n");
        } else {
            printf("Produced: %d\n", item);
        }

        sem_post(&mutex);
        sem_post(&full);

        // 随机睡眠一段时间
```

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

```
collect2: 错误: ld 返回 1
[root@localhost process_synchronization]# gcc -o main main.c -pthread -lrt
[root@localhost process_synchronization]# ./main
Usage: ./main <sleep duration> <num producers> <num consumers>
[root@localhost process_synchronization]# ./main 10 2 2
```

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

```
// 解析命令行输入的信息
if (argc != 4) {
    fprintf(stderr, "Usage: %s <sleep duration> <num producers> <num consumers>\n", argv[0]);
    exit(EXIT_FAILURE);
}

int sleep_duration = atoi(argv[1]);
int num_producers = atoi(argv[2]);
int num_consumers = atoi(argv[3]);

// 初始化三个信号量
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
sem_init(&mutex, 0, 1);

// 创建一定数量的生产者
pthread_t producer_threads[num_producers];
for (int i = 0; i < num_producers; ++i) {
    pthread_create(&producer_threads[i], NULL, producer, NULL);
}

// 创建一定数量的消费者
pthread_t consumer_threads[num_consumers];
for (int i = 0; i < num_consumers; ++i) {
    pthread_create(&consumer_threads[i], NULL, consumer, NULL);
}

// 主函数休息一会
sleep(sleep_duration);

// 在主函数醒来时终结这些线程
for (int i = 0; i < num_producers; ++i) {
    pthread_cancel(producer_threads[i]);
}

for (int i = 0; i < num_consumers; ++i) {
    pthread_cancel(consumer_threads[i]);
}

```

```
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;
5.
6.          sem_wait(&empty);
7.          sem_wait(&mutex);
8.
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.         }
15.
16.         sem_post(&mutex);
17.         sem_post(&full);
18.
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);
5.
6.          buffer_item item;
7.
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.         }
14.
15.         sem_post(&mutex);
16.         sem_post(&empty);
17.
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

```
[root@localhost process_synchronization]# ./main 10 2 2
Produced: 83
Produced: 86
Consumed: 83
Consumed: 86
Produced: 86
Consumed: 86
Produced: 21
Consumed: 21
Produced: 90
Consumed: 90
Produced: 26
Consumed: 26
Produced: 72
Produced: 11
Consumed: 72
Produced: 29
Consumed: 11
Consumed: 29
Produced: 23
Produced: 35
Produced: 2
Consumed: 23
Consumed: 35
Produced: 67
Consumed: 2
Consumed: 67
Produced: 42
Consumed: 42
Produced: 21
Consumed: 21
```

图 5-1 测试用例 1 执行结果图 1

```
Produced: 37
Consumed: 37
Produced: 15
Consumed: 15
Produced: 26
Consumed: 26
Produced: 56
Consumed: 56
Produced: 70
Consumed: 70
Produced: 5
Consumed: 5
Produced: 27
Consumed: 27
Produced: 46
Consumed: 46
Produced: 57
Consumed: 57
Produced: 82
Consumed: 82
Produced: 67
Consumed: 67
Produced: 43
Produced: 87
Produced: 76
Consumed: 43
Produced: 84
Consumed: 87
```

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

```
[root@localhost process_synchronization]# ./main 5 5 1
Produced: 83
Produced: 86
Produced: 77
Produced: 86
Consumed: 83
Produced: 21
Produced: 27
Consumed: 86
Produced: 59
Consumed: 77
Produced: 40
Consumed: 86
Produced: 26
Consumed: 21
Produced: 72
Consumed: 27
Produced: 36
Consumed: 59
Produced: 11
Consumed: 40
Produced: 30
```

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

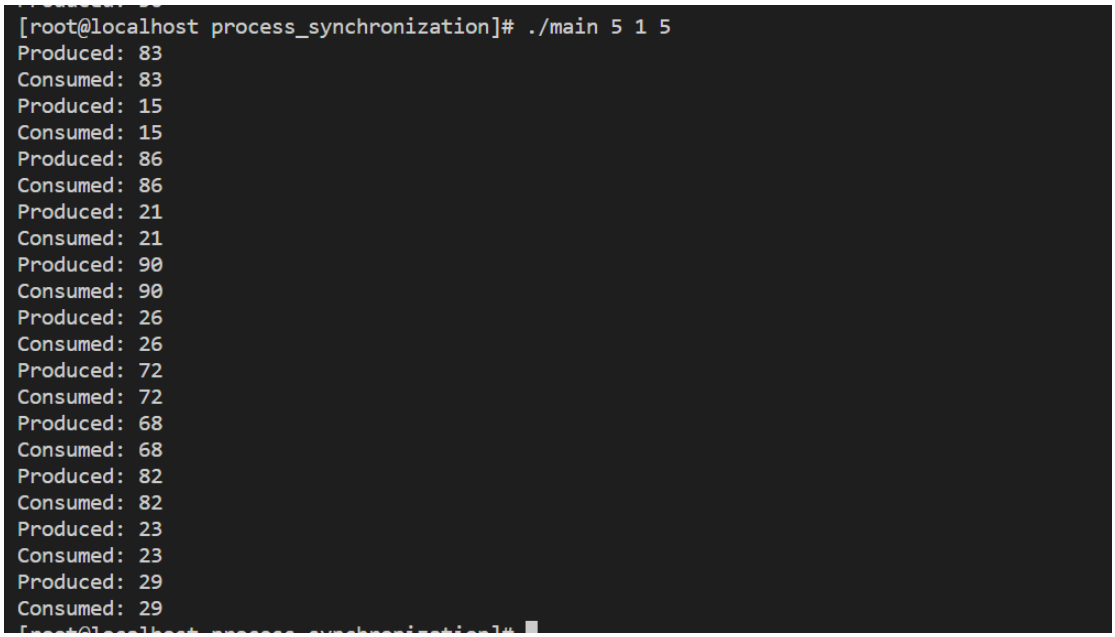

图 5-5 测试用例 2 执行结果图

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5

sem_t empty, full, mutex;
typedef int buffer_item;  // 实验指导的定义
buffer_item buffer[BUFFER_SIZE];
int in = 0, out = 0;

```

```c
int insert_item(buffer_item item) {
    // 插入 item 到缓冲区，成功返回 0，否则返回-1
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    return 0;
}

int remove_item(buffer_item *item) {
    // 从缓冲区删除一个 item，放入 item，成功返回 0，否则返回-1
    *item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return 0;
}

void *producer(void *arg) {
    while (1) {
        // 生产随机数
        buffer_item item = rand() % 100;

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

        // 插入随机数
        if (insert_item(item) == -1) {
            fprintf(stderr, "Error: Buffer full\n");
        } else {
            printf("Produced: %d\n", item);
        }

        sem_post(&mutex);
        sem_post(&full);

        // 随机睡眠一段时间
        usleep(rand() % 1000000);
    }
}

void *consumer(void *arg) {
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

```

```
56.         buffer_item item;
57.
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.         }
64.
65.         sem_post(&mutex);
66.         sem_post(&empty);
67.
68.         // 随机睡眠一段时间
69.         usleep(rand() % 1000000);
70.     }
71. }
72.
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 consum
    ers>\n", argv[0]);
77.         exit(EXIT_FAILURE);
78.     }
79.
80.     int sleep_duration = atoi(argv[1]);
81.     int num_producers = atoi(argv[2]);
82.     int num_consumers = atoi(argv[3]);
83.
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);
88.
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.     }
94.
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.     }
100.
101.    // 主函数休息一会
102.    sleep(sleep_duration);
103.
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.    }
108.
109.    for (int i = 0; i < num_consumers; ++i) {
110.        pthread_cancel(consumer_threads[i]);
111.    }
112.
113.    // 销毁信号量
114.    sem_destroy(&empty);
115.    sem_destroy(&full);
116.    sem_destroy(&mutex);
117.
118.    return 0;
119. }
```