实验三 进程同步实验

班级：2022211304

姓名：赵宇鹏

学号：2022211119

实验内容

基于 openEuler 操作系统，设计一个 C 语言程序，利用信号量机制解决有限

缓冲的生产者-消费者问题。

实验要求

1. **缓冲区**
   - 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。
   - 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()**
   - 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。
   - 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**
   - 生产者线程：随机睡眠一段时间，向缓冲区插入一个随机数。
   - 消费者线程：随机睡眠一段时间，从缓冲区去除一个随机数。

程序设计说明

API介绍

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

   (a) 头文件：`<semaphore.h>`

   (b) 功能：初始化信号量

   (c) 参数：

   `sem_t *sem`：所需初始化信号量 `sem` 的地址。

   `int pshared`：表明该信号量是否被同一进程下的线程或其他进程共享。0 表示该信号量可以在同一进程下的线程所共享；如果不是 0 则表示该信号量可以在进程间共享。

   `unsigned int value`：信号量初始值。

   (d) 返回值：初始化成功则返回 0，失败则返回-1。

2. `int sem_wait(sem_t *sem)`

   (a) 头文件：`<semaphore.h>`

(b) 功能：如果信号量的值大于零，则减量继续进行，函数立即返回。 如果信号量当前的值为零，则调用将阻塞，直到有可能执行减量操作为止。

(c) 参数：信号量 `sem` 的地址。

(d) 返回值：运行成功则返回 0，失败返回-1。

3. `int sem_post(sem_t *sem)`

   (a) 头文件：`<semaphore.h>`

   (b) 功能：解锁信号量`sem`。

   (c) 参数：信号量 `sem`的地址。

   (d) 返回值：运行成功则返回 0，失败返回-1。

4. `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`

   (a) 头文件：`<pthread.h>`

   (b) 功能：在调用 `pthread_create()`函数的进程中创建一个新的线程。

   (c) 参数：

   `pthread_t *thread`：当线程创建成功时，存储新线程的 ID。

   `const pthread_attr_t *attr`：`attr` 指向一个 `pthread_attr_t`结构，该结构的内容用于决定新建的线程的属性。如果`attr`为 NULL，则新线程为默认属性。

   `void *(*start_routine)(void *)`：新线程通过调用 `strat_routine()`函数开始执行。

   `void *arg`：向函数 start_routine()传递的唯一参数。当 `arg` 为 NULL 时，表明没有参数传递。

   (d) 返回值：函数运行成功返回 0，失败则返回一个错误编号。

代码结构说明

全局变量

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;
```

- `buffer`：循环缓冲区，用于存放产品。
- `in`、`out`：循环缓冲区的指针，用于指示生产者和消费者所修改的位置。
- `empty`、`full`：信号量，用于控制缓冲区的空满状态。
- `mutex`：互斥锁，用于保证缓冲区操作的原子性。

信号量的初始化如下

```
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&full, 0, 0);
pthread_mutex_init(&mutex, NULL);
```

`empty`初始化为数组的大小（此处为5），代表还有五个空位，`full`初始化为0，代表目前缓冲区中没有数据。

生产者

生产者的实现结构主要参照课本上所展示伪代码，首先生成一个随机数为生产者的产品，等待空位，等待进入临界缓冲区，进入缓冲区后禁止其他进程进入临界区，进行插入操作，生产者的指针后移，退出临界区，然后发出填充消息，为了模仿真实生产场景及方便查看输出，特地加入了一个随机睡眠时间来约束生产者的输出。

具体代码如下：

```
void* producer(void* arg) {
    while (1) {
        int item = rand() % 100;  // 生成一个随机数作为产品
        sleep(rand() % 3);

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

        buffer[in] = item;
        printf("生产者 %ld 生产了 %d 在位置 %d\n", pthread_self(), item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    return NULL;
}
```

消费者

消费者的思路与生产者总体相同，只不过由插入一个随机数据变成取出一个数，消费者的指针后移，代码如下：

```
void* consumer(void* arg) {
    while (1) {
        sleep(rand() % 3);
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("消费者 %ld 消费了 %d 从位置 %d\n", pthread_self(), item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    return NULL;
}
```

主函数

首先读取用户输入的睡眠时长，生产者线程数，消费者线程数，并根据这些值来创建相应的线程以及睡眠时间，然后执行这些线程，最后睡眠时间到则结束。

代码实现如下：

```
int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "用法: %s <睡眠时长> <生产者线程数> <消费者线程数>\n",
argv[0]);
        return 1;
    }

    int sleep_time = atoi(argv[1]);
    int producer_count = atoi(argv[2]);
    int consumer_count = atoi(argv[3]);

    pthread_t producers[producer_count], consumers[consumer_count];

    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < producer_count; i++) {
        pthread_create(&producers[i], NULL, producer, NULL);
    }

    for (int i = 0; i < consumer_count; i++) {
        pthread_create(&consumers[i], NULL, consumer, NULL);
    }

    sleep(sleep_time);

    printf("时间到，结束程序\n");
    exit(0);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

程序编译及运行

编译方法

由于在实验中使用了 POSIX 线程（`pthread`）和信号量（`sem_t`），这通常在 Linux 系统中比较常见，而实验要求要脱离开发环节正常运行，所以为了使可执行文件可以跨平台运行，我使用了交叉编译器`MinGW`在Linux上编译出能够在Windows操作系统上的可执行文件`os_lab3.exe`

首先需要下载`mingw-w64`

```
sudo apt update
sudo apt install migw-w64
```

下载完成后，使用以下指令进行编译，这里使用`-static`选项将标准库（如`libgcc`、`libstdc++`等）静态链接到可执行文件中，通过链接`pthread`库来解决线程和信号量的使用

```
x86_64-w64-mingw32-gcc -o os_lab3.exe os_lab3.c -static -lpthread
```

最后我们就可以使用如下命令在Windows下执行文件了

```
./os_lab3.exe 8 4 1
```

其中第一个参数为系统休眠时间，第二个参数为生产者线程数目，第三个参数为消费者线程数目。

程序运行结果及分析

测试一

输入

```
./os_lab3.exe 8 4 1
```

输出

```
生产者 140097438320320 生产了 86 在位置 0
消费者 140097429927616 消费了 86 从位置 0
生产者 140097446713024 生产了 93 在位置 1
消费者 140097429927616 消费了 93 从位置 1
生产者 140097455105728 生产了 77 在位置 2
生产者 140097455105728 生产了 26 在位置 3
生产者 140097463498432 生产了 83 在位置 4
生产者 140097438320320 生产了 21 在位置 0
生产者 140097438320320 生产了 68 在位置 1
消费者 140097429927616 消费了 77 从位置 2
生产者 140097446713024 生产了 90 在位置 2
消费者 140097429927616 消费了 26 从位置 3
生产者 140097455105728 生产了 26 在位置 3
消费者 140097429927616 消费了 83 从位置 4
生产者 140097463498432 生产了 36 在位置 4
消费者 140097429927616 消费了 21 从位置 0
生产者 140097463498432 生产了 22 在位置 0
时间到，结束程序
```

分析

此次输入共创建了4个生产者，1个消费者，主要检验程序是否运行正常，从输出可以看到生产者编号有4个，而消费者编号仅有一个，线程创建成功，从输出可以看到，由于生产者更多，所以毫无疑问生产者生产速度是大于消费者消费速度的，而当生产者生产数据将缓冲区填满后（即生产者指针追上消费者指针），生产者就会开始等待消费者消费完成后再进行生产，由于消费者只有一个，所以在后期结果为生产者和消费者轮流运行，程序结果符合预期，运行正常。

测试二

输入

```
./os_lab3.exe 8 2 1
```

输出

```
生产者 140156322809536 生产了 83 在位置 0
消费者 140156306024128 消费了 83 从位置 0
生产者 140156314416832 生产了 77 在位置 0
消费者 140156306024128 消费了 77 从位置 0
生产者 140156322809536 生产了 35 在位置 0
消费者 140156306024128 消费了 35 从位置 0
生产者 140156314416832 生产了 49 在位置 0
消费者 140156306024128 消费了 49 从位置 0
生产者 140156322809536 生产了 27 在位置 0
消费者 140156306024128 消费了 27 从位置 0
生产者 140156314416832 生产了 63 在位置 0
消费者 140156306024128 消费了 63 从位置 0
生产者 140156322809536 生产了 26 在位置 0
消费者 140156306024128 消费了 26 从位置 0
生产者 140156314416832 生产了 11 在位置 0
时间到，结束程序
```

分析

本次测试我修改了代码结构，即将缓冲区数组大小设置为1，这样无论有多少生产者和消费者，都只能轮流运行，程序输出结果也符合预期。

测试三

输入

```
./os_lab3.exe 8 3 3
```

输出

```
生产者 139939835115200 生产了 83 在位置 0
生产者 139939818329792 生产了 93 在位置 1
生产者 139939826722496 生产了 77 在位置 2
消费者 139939801544384 消费了 83 从位置 0
消费者 139939809937088 消费了 93 从位置 1
消费者 139939793151680 消费了 77 从位置 2
生产者 139939818329792 生产了 27 在位置 3
生产者 139939826722496 生产了 59 在位置 4
生产者 139939835115200 生产了 21 在位置 0
消费者 139939801544384 消费了 27 从位置 3
消费者 139939809937088 消费了 59 从位置 4
消费者 139939793151680 消费了 21 从位置 0
生产者 139939818329792 生产了 72 在位置 1
消费者 139939801544384 消费了 72 从位置 1
生产者 139939826722496 生产了 11 在位置 2
生产者 139939835115200 生产了 29 在位置 3
```

```
消费者 139939809937088 消费了 11 从位置 2
生产者 139939826722496 生产了 29 在位置 4
消费者 139939793151680 消费了 29 从位置 3
生产者 139939835115200 生产了 22 在位置 0
生产者 139939818329792 生产了 23 在位置 1
生产者 139939818329792 生产了 29 在位置 2
消费者 139939801544384 消费了 29 从位置 4
消费者 139939793151680 消费了 22 从位置 0
生产者 139939835115200 生产了 11 在位置 3
生产者 139939826722496 生产了 67 在位置 4
生产者 139939826722496 生产了 15 在位置 0
消费者 139939809937088 消费了 23 从位置 1
消费者 139939793151680 消费了 29 从位置 2
生产者 139939818329792 生产了 21 在位置 1
消费者 139939801544384 消费了 11 从位置 3
生产者 139939818329792 生产了 56 在位置 2
生产者 139939826722496 生产了 13 在位置 3
时间到，结束程序
```

分析

若生产者消费者数目不等的话，最后输出结构的后面一定是生产者和消费者轮流执行的输出，所以本输入改为设置生产者和消费者数目都为3，可以看到虽然由于每次生产和消费的时间都是随机的而导致后面开始失衡，但是输出总体上是均衡执行的，程序运行正常。

实验总结

本次实验我设计并实现了一个基于多线程和信号量的生产者-消费者模型，主要目的是通过模拟生产者和消费者对共享缓冲区的并发访问，掌握线程同步的基本方法，理解缓冲区在多线程编程中的重要作用，通过本次实验，我对课程中生产者-消费者问题的解决有了更深刻的了解，也对进程同步这一章节进行了进一步的学习，收获颇丰。