《操作系统原理》实验报告

实验名称	多线程与信号量编程	实验序号	4
实验日期	2023/4/19	实验人	盖乐

一、实验题目

在第 5.7.1 节中，我们提出了一种使用有界缓冲区的信号量解决方案来解决生产者消费者问题。你将使用图 5.9 和 5.10 中所示的生产者和消费者进程来设计有界缓冲区问题的编程解决方案。5.7.1 节中介绍的解决方案使用三个信号量：empty 和 full，它们计算缓冲区中空槽和满槽的数量，以及 mutex，它是一个二进制（或互斥）信号量，用于保护实际插入或删除缓冲区中的项目。你将使用标准计数信号量表示 empty 和 full，并使用互斥锁而不是二进制信号量来表示 mutex。生产者和消费者作为单独的线程运行——将产品移入和移出与 empty、full 和 mutex 结构同步的缓冲区。可以使用 Pthreads 或 WindowsAPI 实现。

要求：

生产者线程将在随机休眠一段时间和将随机整数插入缓冲区之间交替。随机数将使用 rand()函数产生，该函数产生介于 O 和 RAND_MAX 之间的随机整数。消费者也会随机休眠一段时间，醒来后会尝试从缓冲区中删除一个项目。生产者和消费者线程的概要如图 5.26 所示。

二、相关原理与知识

1. 临界区段概述：

用于多线程的互斥访问。如果有多个线程试图同时访问临界区，那么在有一个线程进入临界区后，其他试图访问的线程将被挂起，直到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到对临界区的互斥访问。（临界区中一般都是一个简短的代码段）在 WINDOWS 中，临界区是一种应用层的同步对象，非内核对象。并且临界区优先采用自旋的方式进行抢占。

2. 临界区 API 及数据结构：

临界区初始化以及删除：InitializeCriticalSection（）；DeleteCriticalSection（）；

临界区两个操作原语：EnterCriticalSection（）；LeaveCriticalSection（）

临界区数据结构：typedefstruct_RTL_CRITICAL_SECTION

{PRTL_CRITICAL_SECTION_DEBUGDebugInfo;

LONGLockCount;

LONGRecursionCount;

HANDLEOwningThread;//fromthethread'sClientId->UniqueThread

HANDLELockSemaphore;

ULONG_PTRSpinCount;//forcesizeon64-bitsystemswhenpacked

}RTL_CRITICAL_SECTION,*PRTL_CRITICAL_SECTION;

3. 信号量与互斥锁

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设

施,它负责协调各个线程,以保证它们能够正确、合理的使用公共资源。信号量，是可以用来保护两个或多个关键代码段，这些关键代码段不能并发调用。在进入一个关键代码段之前，线程必须获取一个信号量。如果关键代码段中没有任何线程，那么线程会立即进入该框图中的那个部分。一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量，然后将 AcquireSemaphoreVI 以及 ReleaseSemaphoreVI 分别放置在每个关键代码段的首末端。确认这些信号量 VI 引用的是初始创建的信号量。

互斥量(Mutex)：互斥量表现互斥现象的数据结构，也被当作二元信号灯。一个互斥基本上是一个多任务敏感的二元信号，它能用作同步多任务的行为，它常用作保护从中断来的临界段代码并且在共享同步使用的资源。Mutex 本质上说就是一把锁，提供对资源的独占访问，所以 Mutex 主要的作用是用于互斥。Mutex 对象的值，只有 0 和 1 两个值。这两个值也分别代表 Mutex 的两种状态。值为 0,表示锁定状态，当前对象被锁定，用户进程/线程如果试图 Lock 临界资源，则进入排队等待；值为 1，表示空闲状态，当前对象为空闲，用户进程/线程可以 Lock 临界资源，之后 Mutex 值减 1 变为 0。

信号量与互斥锁之间的区别：

1.互斥量用于线程的互斥，信号线用于线程的同步。这是互斥量和信号量的根本区别，也就是互斥和同步之间的区别。互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

2.互斥量值只能为 0/1，信号量值可以为非负整数。也就是说，一个互斥量只能用于一个资源的互斥访问，它不能实现多个资源的多线程互斥问题。信号量可以实现多个同类资源的多线程互斥和同步。当信号量为单值信号量是，也可以完成一个资源的互斥访问。

3.互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

4. 信号量与互斥锁操作：

Semaphore 可以被抽象为五个操作：创建 Create；等待 Wait：线程等待信号量，如果值大于 0，则获得，值减一；如果只等于 0，则一直线程进入睡眠状态，知道信号量值大于 0 或者超时；释放 Post：执行释放信号量，则值加一；如果此时有正在等待的线程，则唤醒该线程；试图等待 TryWait：如果调用 TryWait，线程并不真正的去获得信号量，还是检查信号量是否能够被获得，如果信号量值大于 0，则 TryWait 返回成功；否则返回失败；销毁 Destroy。

动作系统：创建：CreateSemaphore/sem_init；等待：WaitForSingleObject/sem_wait；释放：ReleaseMutex/sem_post；试图等待：WaitForSingleObject/sem_trywait；销毁：CloseHandle/sem_destroy

Mutex 可以被抽象为四个操作：创建 Create；加锁 Lock；解锁 Unlock；销毁 Destroy

Mutex 被创建时可以有初始值，表示 Mutex 被创建后，是锁定状态还是空闲状态。在同一个线程中，为了防止死锁，系统不允许连续两次对 Mutex 加锁(系

统一般会在第二次调用立刻返回)。也就是说，加锁和解锁这两个对应的操作，需要在同一个线程中完成。

不同操作系统中提供的 Mutex 函数:动作系统：

创建：CreateMutex/pthread_mutex_init/mutex_init；

加锁：WaitForSingleObject/pthread_mutex_lock/mutex_loc；

解锁：ReleaseMutex/pthread_mutex_unlock/mutex_unlock；

销毁：CloseHandle/pthread_mutex_destroy/mutex_destroy

5.生产者消费者问题

有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费，为使生产者与消费者能并发的执行，在两者之间设置了具有 n 个缓冲区的缓冲池，生产者进程将其所生产的产品放入一个缓冲区中；消费者进程可从一个缓冲区中取走产品。他们之间必须保持同步，即：不允许消费者到一个空的缓冲区中去取产品，也不允许生产者向一个已经装满产品的缓冲区中投放产品。当缓冲区为空时，消费者进程需要挂起休眠，直至生产者进程将产品放入缓冲区，消费者才能被唤醒；相反，如果缓冲区满时，消费者进程需要挂起，直至缓冲区中的产品被消费者拿走，才可以被唤醒。

三、实验过程

实验设计：

这个实验的思路是利用两个信号量"empty"和"full"来记录空缓冲区和满缓冲区，同时使用三个互斥锁"mutex"、"pro_mutex"和"con_mutex"分别作为缓冲区、生产者计数和消费者计数的互斥锁。通过线程指针可以实现动态个数的消费者和生产者进程，并通过循环进行创建。在生产和消费的过程中，首先获取到自己消费者或生产者的编号，然后进入死循环，在随机时间段内进行生产或消费操作。为了避免消费和生产过于一致，使得生产者和消费者睡眠的时间长度不同。

首先使用一个循环队列用以表示一个共享资源区。

```
#define BUFFER_SIZE 0x100

int items_queue[BUFFER_SIZE];
pthread_mutex_t queue_mutex, con_no_mutex, pro_no_mutex;
sem_t empty, full;

int front = 0, end = 0;
```

为了保证多线程程序的正确性，我们需要在代码中设置相应的信号量和互斥锁。在此，我们将它们设置为全局变量。其中，empty 和 full 使用信号量进行标识。如果生产者数量小于队列大小，则将 empty 信号量的数量初始化为生产者数量；否则，将其初始化为队列大小。而 full 信号量的数量则始终初始化为 0。为了确保对队列的操作是线程安全的，我们使用一个互斥锁进行标识。

```
sem_t empty, full;
```

```
sem_init(&empty, 0, (BUFFER_SIZE > producer_nums ? producer_nums : BUFFER_SIZE));
sem_init(&full, 0, 0);
```

设置对应的向队列中插入、从队列中取出 item 的函数，在这里选择将多线程相关操作放在其上层调用函数中，因此这两个函数并未设置相应的信号量、互斥锁等的判断

```
void insertItem(int item)
{
    items_queue[front++] = item;
    front %= BUFFER_SIZE;
}

int removeItem(void)
{
    end %= BUFFER_SIZE;
    return items_queue[end++];
}
```

接下来，我们要设计生产者线程。在获取到 empty 信号量后，我们会根据一个随机数来决定该线程是进入睡眠状态还是进行生产操作。具体来说，如果随机数为奇数，则线程会进入睡眠状态；如果是偶数，则线程会获取互斥锁，并向队列中插入数据。最后，线程会增加 full 信号量的数量。需要注意的是，由于 empty 信号量被初始化为适当的大小，因此生产者线程会一开始就开始工作，而不会发生队列溢出的情况。

```
void * producerThread(void * args)
{
    int thread_no, item;

    pthread_mutex_lock(&pro_no_mutex);
    thread_no = ++producer_no;
    pthread_mutex_unlock(&pro_no_mutex);

    while (1)
    {
        if (rand() % 2)
        {
            sleep(rand() % 10);
        }
        else
        {
            sem_wait(&empty);
            pthread_mutex_lock(&queue_mutex);
            do
            {
                item = rand() % 100;
            } while (item == 0);
            insertItem(item);
            printf("=== producer %d produce: %d ===\n", thread_no, item);
            total_produce += item;
            pthread_mutex_unlock(&queue_mutex);
            sem_post(&full);
        }
    }
}
```

与生产者线程类似，消费者线程的逻辑也比较简单。它会先获取 full 信号量，如果获取成功，则会从队列中取出数据。需要注意的是，在初始状态下，full 信号量的数量为 0，因此消费者线程一开始处于等待状态。只有当第一个生产者向队列中插入数据后，才会激活消费者线程。这样，就成功解决了生产者-消费者问题。

```c
void * consumerThread(void * args)
{
    int thread_no, item;

    pthread_mutex_lock(&con_no_mutex);
    thread_no = ++consumer_no;
    pthread_mutex_unlock(&con_no_mutex);

    while (1)
    {
        if (rand() % 2)
        {
            sleep(rand() % 10);
        }
        else
        {
            sem_wait(&full);
            pthread_mutex_lock(&queue_mutex);
            item = removeItem();
            printf("=== consumer %d consumes: %d ===\n", thread_no, item);
            total_consume += item;
            pthread_mutex_unlock(&queue_mutex);
            sem_post(&empty);
        }
    }
}
```

四、实验结果与分析

1. 让程序运行 10s，创建生产者 5 个，消费者 8 个。

```
gaile@gaile-vmwarevirtualplatform:~/桌面/demo$ ./ex4 10 5 8
=== producer 2 produce: 13 ===
=== consumer 1 consumes: 13 ===
=== producer 3 produce: 82 ===
=== consumer 3 consumes: 82 ===
=== producer 2 produce: 86 ===
=== consumer 5 consumes: 86 ===
=== producer 1 produce: 52 ===
=== producer 1 produce: 29 ===
=== consumer 8 consumes: 52 ===
=== producer 3 produce: 7 ===
=== producer 3 produce: 39 ===
=== consumer 1 consumes: 29 ===
=== consumer 1 consumes: 7 ===
=== consumer 6 consumes: 39 ===
=== producer 4 produce: 24 ===
=== consumer 2 consumes: 24 ===
totally produce: 332, totally consume: 332
```

我们可以看到，生产者<消费者，8 个消费者都在争抢着 5 个生产者所生产的资源，所以最后的结果便是每次生产者一生产出资源，就会被消费者所消耗，最后所有的资源都被消耗

2. 让程序运行 10s，创建生产者 8 个，消费者 5 个。

```
gaile@gaile-vmwarevirtualplatform:~/桌面/demo$ ./ex4 10 8 5
=== producer 1 produce: 98 ===
=== producer 1 produce: 87 ===
=== consumer 2 consumes: 98 ===
=== producer 2 produce: 60 ===
=== consumer 1 consumes: 87 ===
=== consumer 1 consumes: 60 ===
=== producer 3 produce: 25 ===
=== producer 2 produce: 55 ===
=== consumer 5 consumes: 25 ===
=== producer 4 produce: 48 ===
=== producer 5 produce: 29 ===
=== producer 5 produce: 86 ===
=== producer 5 produce: 16 ===
=== producer 7 produce: 41 ===
=== consumer 3 consumes: 55 ===
=== consumer 3 consumes: 48 ===
=== producer 7 produce: 4 ===
=== producer 7 produce: 27 ===
=== producer 6 produce: 77 ===
=== producer 3 produce: 45 ===
=== consumer 4 consumes: 29 ===
=== producer 3 produce: 1 ===
=== consumer 4 consumes: 86 ===
=== consumer 4 consumes: 16 ===
=== producer 6 produce: 56 ===
=== producer 5 produce: 14 ===
=== consumer 1 consumes: 41 ===
=== producer 3 produce: 90 ===
=== consumer 5 consumes: 4 ===
=== producer 7 produce: 50 ===
=== consumer 5 consumes: 27 ===
=== consumer 5 consumes: 77 ===
=== consumer 5 consumes: 45 ===
=== producer 4 produce: 39 ===
=== producer 4 produce: 26 ===
=== producer 5 produce: 28 ===
=== consumer 2 consumes: 1 ===
=== consumer 2 consumes: 56 ===
=== producer 5 produce: 73 ===
=== producer 5 produce: 36 ===
totally produce: 1111, totally consume: 755
```

我们可以看到，由于生产者>消费者，因此最后并非所有生产的资源都被消耗，而是还有着一定的剩余，但是由于生产者数量与消费者数量差距不大，因而剩余也并不多。

由以上两次测试我们也可以看出我们的程序很好地成功模拟了生产者—消费者问题。

五、问题总结

1.在第一次写生产者消费者模型时，我没有完全理解 empty 和 full 的作用。我使用了一个计数器 count 来记录缓冲区中资源的数量，以防止缓冲区溢出。后来我发现，在每次生产者生产资源时，需要先减少一个 empty 信号量来告诉其他生产者现在正在生产一个资源，然后再增加一个 full 信号量来告诉消费者已经成功生产了一个资源，可以进行消费了。这样就会导致缓冲区永远不会溢出的问题。

2.在实验过程中，我在 win11 系统上运行程序时一直报错，但是我无法找出原因。后来我发现，windows 系统不使用 sem_init 函数创建信号量，而是采用 CreateSemaphore 函数。这一点需要注意。

六、源代码

#include <stdio.h>

```
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>
#include <semaphore.h>

#define BUFFER_SIZE 0x100

int items_queue[BUFFER_SIZE];
pthread_mutex_t queue_mutex, con_no_mutex, pro_no_mutex;
sem_t empty, full;

int front = 0, end = 0;
int producer_nums, consumer_nums;
int producer_no = 0, consumer_no = 0;
int running_time;
int total_consume = 0, total_produce = 0;

pthread_t * consumer, *producer;

int main(int argc, char ** argv)
{
    if (argc != 4)
    {
        puts("Usage: ./c_p time producer_nums consumer_nums");
        exit(0);
    }
```

```
    running_time = atoi(argv[1]);
    producer_nums = atoi(argv[2]);a
    consumer_nums = atoi(argv[3]);
    if (running_time < 0 || producer_nums < 0 || consumer_nums < 0)
    {
        puts("Invalid arguments!");
        return 0;
    }

    sem_init(&empty, 0, (BUFFER_SIZE > producer_nums ? producer_nums :
BUFFER_SIZE));
    sem_init(&full, 0, 0);
    pthread_mutex_init(&queue_mutex, NULL);
    pthread_mutex_init(&con_no_mutex, NULL);
    pthread_mutex_init(&pro_no_mutex, NULL);

    srand(time(NULL));

    consumer = (pthread_t*) malloc(sizeof(pthread_t) * consumer_nums);
    producer = (pthread_t*) malloc(sizeof(pthread_t) * producer_nums);

    for (int i = 0; i < consumer_nums;i++)
        pthread_create(consumer + i, NULL, consumerThread, NULL);

    for (int i = 0; i < producer_nums;i++)
        pthread_create(producer + i, NULL, producerThread, NULL);

    sleep(running_time);
    pthread_mutex_lock(&queue_mutex);
    printf("totally produce: %d, totally consume: %d\n", total_produce,
total_consume);
    exit(0);
```

```
}

void insertItem(int item)
{
	items_queue[front++] = item;
	front %= BUFFER_SIZE;
}

int removeItem(void)
{
	end %= BUFFER_SIZE;
	return items_queue[end++];
}

void * consumerThread(void * args)
{
	int thread_no, item;

	pthread_mutex_lock(&con_no_mutex);
	thread_no = ++consumer_no;
	pthread_mutex_unlock(&con_no_mutex);

	while (1)
	{
		if (rand() % 2)
		{
			sleep(rand() % 10);
		}
		else
		{
			sem_wait(&full);
			pthread_mutex_lock(&queue_mutex);
```

```
                item = removeItem();
                printf("=== consumer %d consumes: %d ===\n", thread_no,
item);
                total_consume += item;
                pthread_mutex_unlock(&queue_mutex);
                sem_post(&empty);
            }
        }
}

void * producerThread(void * args)
{
    int thread_no, item;

    pthread_mutex_lock(&pro_no_mutex);
    thread_no = ++producer_no;
    pthread_mutex_unlock(&pro_no_mutex);

    while (1)
    {
        if (rand() % 2)
        {
            sleep(rand() % 10);
        }
        else
        {
            sem_wait(&empty);
            pthread_mutex_lock(&queue_mutex);
            do
            {
                item = rand() % 100;
            } while (item == 0);
```

```
                insertItem(item);
                printf("=== producer %d produce: %d ===\n", thread_no, item);
                total_produce += item;
                pthread_mutex_unlock(&queue_mutex);
                sem_post(&full);
            }
        }
}
```