北京理工大學

操作系统课程设计

实验三、生产者消费者问题	**实验三：生产者消费者问题**

学院：计算机学院
专业：计算机科学与技术
学生姓名：夏奇拉
学号：　1820171025
：　07111705

目的

1 Windows 和 Linux 系统；

2 创建 4 个缓冲区，开头为“-”；

3 创建 3 个生产者。对于每个生产者：

-等待 3 秒内（含 3 秒）的随机时间，随机添加一个你名字的首字母（大写）到缓冲区。例如，王全宇的姓名首字母是 W、Q、Y，即从这三个字母中每次取一个；

- 如果缓冲区已满，则等待消费者取走字母后再添加；

- 重复 4 次；

4.创建 4 个消费者，每个消费者

-等待一段随机的时间从缓冲区中读取字母

- 如果缓冲区为空，则等待生产者添加字母后再读取

- 重复 3 次

5.每一个操作的内容都需要打印出来

- 生产者打印：生产者写入缓冲区的信

-Consumer Print：消费者所拍的信件

- 需要打印缓冲区内容

- 按照先生产的商品先消费的原则

问题讨论

在操作系统中，生产者是能够产生数据的进程。消费者是能够消费生产者生产的数据的进程。生产者和消费者共享一个公共内存缓冲区。该缓冲区是系统内存中一定大小的地方，用于存储。生产者将数据生产到缓冲区中，消费者从缓冲区中消费数据。

生产者-消费者问题检查生产者和消费者之间可能出现的问题并提出解决方案。这些问题包括：

- 当缓冲区已满时，生产者进程不应产生数据
- 当共享缓冲区为空时，消费者进程不应消费数据。
- 对共享缓冲区的访问应该是互斥的，即一次只有一个进程应该能够访问共享缓冲区并对其进行更改。

为了使生产者和消费者之间的数据同步一致，需要解决这些问题。

求解算法

为了解决生产者-消费者问题，使用了三个信号量。信号量是用于指示特定时间系统中可用资源数量的变量。它们用于实现进程同步。用于解决该问题的三个信号量是：

1. *FULL*： FULL 变量用于跟踪生产者进程填充在缓冲区中的空间。它最初被初始化为 0，因为最初没有空间被生产者进程填充。
2. *EMPTY*： EMPTY 变量用于跟踪缓冲区中的空白空间。 empty 变量最初被初始化为 *BUFFER_SIZE*，就像最初一样，整个缓冲区是空的。
3. *MUTEX* : *MUTEX* 用于实现互斥。 *MUTEX* 确保在任何特定时间只有生产者或消费者正在访问缓冲区。 *MUTEX* 是一个二进制信号量变量，其值为 0 或 1。

*SIGNAL()*和 *WAIT()*操作。

*SIGNAL()*将信号量值增加 1。

*WAIT()*将信号量值减 1。

生产者-消费者问题的算法如下。我将在 Windows 和 Linux 中实现这个算法。

```
共享内存：
字符缓冲区 [BUFFER_SIZE]
输入=0，输出=0
信号量满=0
信号量为空=BUFFER_SIZE
信号量互斥量=1
```

```
制片人
无效生产者（）{
        同时（真）{
                等待（空）
                等待（互斥量）
                缓冲区 [输入] = c
                输入 = (输入+1) % BUFFER_S
                IZE
                信号（互斥量）
                信号（满）
        }
}
```

```
消费者
无效消费者（）{
        同时（真）{
                等待（满）
                等待（互斥量）
                c = 缓冲区[输出]
                输出 = (输出+1) % BUFFER_S
                IZE
                信号（互斥量）
                信号（空）
                返回 c
        }
}
```

在该算法中，生产者在循环中运行，循环执行计算以生成信息。在这种情况下，它会等待 3 秒（包括 3 秒）内的随机时间段，并随机将我的首字母 XQL 中的单个字符添加到缓冲区。

FIFO 缓冲区实现为具有两个索引的 BUFFER_SIZE 元素字符数组。

IN：写入索引表示下一个要写入的字符

OUT：读取索引指示下一个要读取的字符

IN 和 *OUT* 索引以 BUFFER_SIZE 为模递增，即在 BUFFER_SIZE+1 元素之后要访问的下一个元素是第 0 个元素，因此得名“循环缓冲区”。

在制作人中：

WAIT(EMPTY) - 在生产项目之前，生产者进程检查缓冲区中的空白空间。如果缓冲区已满，生产者进程等待消费者进程使用缓冲区中的项目。因此，生产者进程在生产任何项目之前执行 *WAIT(EMPTY)* 。

WAIT(MUTEX) - 一次只有一个进程可以访问缓冲区。因此，一旦生产者进程进入代码的临界区，它就会通过执行 *WAIT(MUTEX)*来降低 *MUTEX* 的值，这样其他进程就无法同时访问该缓冲区。

BUF[IN] = C - 将项目添加到生产者进程产生的缓冲区中。一旦生产者进程在代码中达到这一点，就可以保证没有其他进程能够同时访问共享缓冲区，这有助于数据一致性。

SIGNAL(MUTEX) - 现在，一旦 Producer 进程将项目添加到缓冲区中，它会将 *MUTEX* 值增加 1，以便处于忙等待状态的其他进程可以访问临界区。

SIGNAL(FULL) - 当生产者进程将一个项目添加到缓冲区空间时，缓冲区空间被一个项目填充，因此它会增加 Full 信号量，以便它正确指示缓冲区中已填充的空间。

在消费者中：

WAIT(FULL) - 在消费者进程开始使用缓冲区中的任何项目之前，它会检查缓冲区是空的还是其中有一些项目。因此，消费者进程在缓冲区中再创建一个空白空间，这由 full 变量指示。执行 *WAIT(FULL)*时，full 变量的值减一。如果 Full 变量已经为零，即缓冲区为空，则消费者进程无法使用缓冲区中的任何项目，它会进入忙等待状态。

WAIT(MUTEX) - 它与生产者进程中解释的相同。它会将 *MUTEX* 减 1，并限制另一个进程进入临界区，直到消费者进程将 *MUTEX* 的值加 1。

C = BUF[OUT] - 此函数使用缓冲区中的一个项目。当代码到达这一点时，它将不允许任何其他进程访问保持数据一致性的临界区。

SIGNAL(MUTEX) - 在使用该项目后，它将 *MUTEX* 值增加 1，以便其他处于忙等待状态的进程现在可以访问临界区。

SIGNAL(EMPTY) - 当消费者进程消费一个项目时，它会增加 *EMPTY* 变量的值，表示缓冲区中的空白空间增加 1。

执行 [视窗]

按照此处列出的步骤设置用于编译 C 程序的 Windows 环境：演练：在命令行上编译 C 程序 | 微软学习。

第一步：创建 producerconsumer.c 程序

在 Visual Studio 2022 的开发人员命令提示中，使用记事本创建一个名为 parentprocess.c 的 C 文件，方法是运行以下命令：

```
> 记事本生产者消费者.c
```

在文件中，写入以下代码：

```
// BIT 100073007 操作系统课程实验室 3：生产者-消费者问题
```

```
// Win32

#include <stdio.h> // printf()、fprintf()
#include <windows.h> //
#include <stdlib.h> // rand()
#include <string.h> // strlen()
#include <tchar.h> // ?
#include <strsafe.h> // ?

#define BUFFER_SIZE 4

#define 制作人 3
#define 消费者 4

#define PRODUCER_ITERATIONS 4 // 生产者循环次数
#define CONSUMER_ITERATIONS 3 //消费者循环次数

// 函数原型
DWORD WINAPI 生产者(LPVOID lpParam);
DWORD WINAPI 消费者(LPVOID lpParam);

// 定义线程
DWORD consumerId[消费者], producerId[生产者];
处理生产者线程[生产者];
处理消费者线程[消费者];

// 定义信号量空、满、互斥量
处理空的、满的、互斥体;

类型定义结构
{
字符值[BUFFER_SIZE];
int next_in, next_out;
} 缓冲区_t;

buffer_t 缓冲区;

// insertInitial 函数用于向缓冲区添加一个初始值
int insertInitial(char initial, long int id)
{
buffer.value[buffer.next_in] = initial;
buffer.next_in = (buffer.next_in + 1) % BUFFER_SIZE;
printf("producer %ld: produced %c\n", id, initial);

printf("\t\t\t\t\t\t缓冲区:");
for (int i = 0; i < BUFFER_SIZE; i++)
{
printf("%c", buffer.value[i]);
}
printf("\n");

返回 0;
```

```
}

// consumeInitial 函数用于从缓冲区中消费一个初始值
int consumeInitial(char *initial, long int id)
{
*initial = buffer.value[buffer.next_out];
buffer.value[buffer.next_out] = '-';
buffer.next_out = (buffer.next_out + 1) % BUFFER_SIZE;
printf("\t\t\tconsumer %ld: consumed %c\n", id, *initial);

printf("\t\t\t\t\t\t缓冲区:");

for (int i = 0; i < BUFFER_SIZE; i++)
{
printf("%c", buffer.value[i]);
}
printf("\n");

返回 0;
}

// Producer 将迭代 PRODUCER_ITERATIONS 次并调用 insertInitial 函数向缓
冲区插入一个初始值
// Producer argument param 是生产者的整数 id，用于区分多个生产者线程
DWORD WINAPI 生产者(LPVOID lpParam)
{

字符首字母 [] = "XQL";
int length = strlen(首字母);

初始字符;
long int id = (long int)lpParam;

int j = PRODUCER_ITERATIONS;

而 (j--)
{
// 等待从 0 到 3 秒的随机时间长度
int randnum = rand() % 4; // 范围在 0 到 3 之间

// 将随机初始值插入缓冲区
int randInitial = rand() % 长度;

WaitForSingleObject(空，无限);
WaitForSingleObject(互斥量，无限);
首字母 = 首字母 [randInitial];
如果 (insertInitial(initial, id))
fprintf(stderr, "插入缓冲区时出错\n");

ReleaseSemaphore(mutex, 1, NULL);
ReleaseSemaphore(满, 1, NULL);
}
```

```
退出线程(0);
}

// Consumer 将迭代 CONSUMER_ITERATIONS 次并调用 consumeInitial 函数向
缓冲区插入一个初始值
// Consumer argument param 是消费者的整型 id, 用于区分多个消费者线程
DWORD WINAPI 消费者(LPVOID lpParam)
{

初始字符;
long int id = (long int)lpParam;

int k = CONSUMER_ITERATIONS;

而 (k--)
{
睡眠((rand() % 6) * 1000);

// 从缓冲区读取
WaitForSingleObject(满, 无限);
WaitForSingleObject(互斥量, 无限);
如果 (consumeInitial(&initial, id))
fprintf(stderr, "从缓冲区中删除时出错\n");
ReleaseSemaphore(mutex, 1, NULL);
释放信号量(空, 1, NULL);
}

退出线程(0);
}

int_tmain()

{
buffer.next_in = 0;
buffer.next_out = 0;

// 用'-'初始化缓冲区
printf("\n 用 '-' 初始化大小为 %d 的缓冲区\n", BUFFER_SIZE);

printf("\t\t\t\t\t\t 缓冲区:");
for (int i = 0; i < BUFFER_SIZE; i++)
{
buffer.value[i] = '-';
printf("%c", buffer.value[i]);
}
printf("\n");

// 创建生产者和消费者线程
长整数我;

printf("\n 正在创建 %d 个消费者和 %d 个生产者\n\n", CONSUMERS, PRODUCER
S);
```

```
full = CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
empty = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
mutex = CreateSemaphore(NULL, 1, 1, NULL);

如果(满 == NULL)
{
printf("创建信号量错误: %d\n", GetLastError());
返回 1;
}

如果(空 == NULL)
{
printf("创建信号量错误: %d\n", GetLastError());
返回 1;
}

如果(互斥量 == NULL)
{
printf("创建信号量错误: %d\n", GetLastError());
返回 1;
}

// 创建生产者线程
对于 (i = 0; i < 生产者; i++)
{
producerthreads[i] = CreateThread(NULL, 0, Producer, (LPVOID)i, 0,
&producerId[i]);

如果(生产者线程[i] == NULL)
{
printf("第 197 行错误\n");
退出进程(3);
}
}

// 创建消费者线程
for (i = 0; i < 消费者; i++)
{
consumerthreads[i] = CreateThread(NULL, 0, Consumer, (LPVOID)i, 0,
&consumerId[i]);

如果(消费者线程[i] == NULL)
{
printf("第 197 行错误\n");
退出进程(3);
}
}

// 等待线程完成
对于 (i = 0; i < 生产者; i++)
{
```

```
WaitForSingleObject(producerthreads[i], INFINITE);
}

for (i = 0; i < 消费者; i++)
{
WaitForSingleObject(消费者线程[i], 无限);
}

for (int i = 0; i < 生产者; i++)
{
CloseHandle(producerthreads[i]);
}

for (int i = 0; i < 消费者; i++)
{
关闭句柄(消费者线程[i]);
}

关闭句柄(满);
关闭句柄(空);
关闭句柄(互斥体);

返回 0;
}
```

创建线程

CreateThread 函数创建一个线程以在调用进程的虚拟地址空间内执行。它接受 6 个参数，3 个可选，3 个必需。

1. [in, optional] LPSECURITY_ATTRIBUTES lpThreadAttributes：指向 SECURITY_ATTRIBUTES 结构的指针，该结构确定返回的句柄是否可以被子进程继承。如果 lpThreadAttributes 为 NULL，则句柄不能被继承。
2. [in] SIZE_T dwStackSize：堆栈的初始大小，以字节为单位。系统将该值四舍五入到最接近的页面。如果此参数为零，则新线程使用可执行文件的默认大小。
3. [in] LPTHREAD_START_ROUTINE lpStartAddress：指向要由线程执行的应用程序定义函数的指针。这个指针代表线程的起始地址。
4. [输入，可选] __drv_aliasesMem LPVOID lpParameter：指向要传递给线程的变量的指针。
5. [in] DWORD dwCreationFlags：控制线程创建的标志。
6. [输出，可选] LPDWORD lpThreadId：指向接收线程标识符的变量的指针。如果此参数为 NULL，则不返回线程标识符。

创建线程必须指定新线程要执行的代码的起始地址。通常，起始地址是程序代码中定义的函数的名称。此函数采用单个参数并返回 DWORD 值。一个进程可以有多个线程同时执行相同的功能。

双字

DWORD 是一个 32 位无符号整数（范围：0 到 4294967295 十进制）。因为 DWORD 是无符号的，所以它的第一位（最高有效位 (Most Significant Bit, MSB)）不保留用于签名。

该类型声明如下：

typedef unsigned long DWORD, *PDWORD, *LPDWORD;

WinAPI

系统函数的调用约定。

LP 无效

指向任何类型的指针。

该类型在 WinDef.h 中声明如下：

typedef void *LPVOID;

逻辑参数

消息参数。

创建信号量

CreateSemaphoreA 函数创建或打开一个命名或未命名的信号量对象。它接受 4 个参数。

1. [输入，可选] LPSECURITY_ATTRIBUTES lpSemaphoreAttributes：指向 SECURITY_ATTRIBUTES 结构的指针。如果该参数为 NULL，则句柄不能被子进程继承。
2. [in] LONG lInitialCount：信号量对象的初始计数。该值必须大于或等于零且小于或等于 lMaximumCount。信号量的状态在其计数大于零时发出信号，而在其为零时不发出信号。每当等待函数释放等待信号量的线程时，计数就会减一。通过调用 ReleaseSemaphore 函数，计数会增加指定的数量。
3. [in] LONG lMaximumCount：信号量对象的最大计数。该值必须大于零。
4. [in, optional] LPCSTR lpName：信号量对象的名称。

WaitForSingleObject 将信号量的计数减一。

ReleaseSemaphore 函数将指定信号量对象的计数增加指定数量。

第二步：编译 C 程序

使用以下命令制作程序的可执行版本。

```
> cl 生产者消费者.c
```

这将输出 producerconsumer.exe

第四步：运行程序

使用以下命令运行新创建的可执行 .exe 程序。

```
> 生产者消费者
```

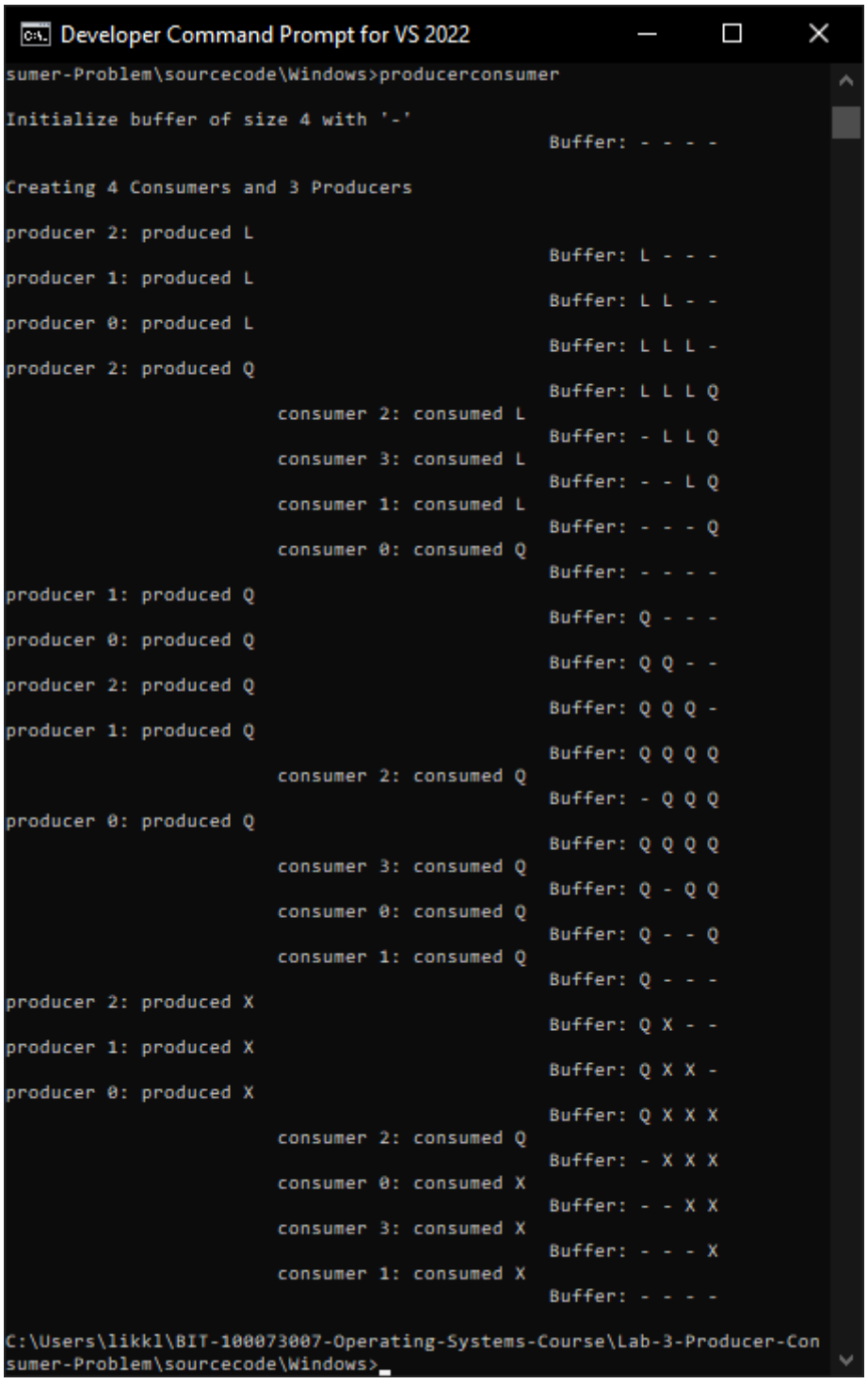
产生以下输出：

```
使用“-”初始化大小为 4 的缓冲区
缓冲: - - - -

创建 4 个消费者和 3 个生产者

生产者 2:生产 L
缓冲器:L - - -
生产者 1:生产 L
缓冲器:LL - -
生产者 0:生产 L
缓冲器:LLL-
生产者 2:生产 Q
缓冲器:LLLQ
消费者 2:消费了 L
缓冲区:- LLQ
消费者 3:消费了 L
缓冲器: - - LQ
消费者 1:消费 L
缓冲区: - - - Q
消费者 0:消费 Q
缓冲: - - - -
生产者 1:生产 Q
缓冲器:Q - - -
生产者 0:生产 Q
缓冲区:QQ - -
生产者 2:生产 Q
缓冲区:QQQ -
生产者 1:生产 Q
缓冲区:QQQQ
消费者 2:消费 Q
缓冲区:-QQQ
生产者 0:生产 Q
缓冲区:QQQQ
消费者 3:消费 Q
缓冲区:Q - QQ
消费者 0:消费 Q
缓冲器:Q - - Q
消费者 1:消费 Q
缓冲器:Q - - -
生产者 2:生产 X
缓冲器:QX - -
生产者 1:生产 X
缓冲区:QXX -
生产者 0:生产 X
缓冲区:QXXX
消费者 2:消费 Q
缓冲区:- XXX
消费者 0:消费 X
缓冲器:- - XX
消费者 3:消费 X
```

缓冲区: - - - X
消费者 1:消费 X
缓冲: - - - -

结果与分析 [Windows]

在命令行运行 producerconsumer，就会启动 producerconsumer 程序。该程序创建了 4 个消费者线程和 3 个生产者线程。它们共享一个大小为 3 的缓冲区，该缓冲区用“-”初始化。生产者等待 3 秒（包括 3 秒）之间的随机时间长度，然后将我的名字 XiaQiLa 的随机首字母添加到缓冲区。每个生产者重复这个过程 3 次。并打印它添加到缓冲区的首字母。每个消费者读取缓冲区并打印它消费的初始值。每次操作后，缓冲区打印其内容。



执行 [Linux]

第 1 步:创建 C 程序

在终端中，通过运行以下命令使用名称为 producerconsumer.c 的 nano 编辑器创建一个文件:

```
$ nano 生产者消费者.c
```

在文件中我们编写如下代码:

```
// BIT 100073007 操作系统课程实验室 3:生产者-消费者问题

#include <stdio.h> // printf()、fprintf()
#include <stdlib.h> // rand()
#include <pthread.h> // pthread_...
#include <unistd.h> // 睡眠()
#include <string.h> // strlen()
#include <semaphore.h> // sem_...

#define BUFFER_SIZE 4

#define 制作人 3
#define 消费者 4

#define PRODUCER_ITERATIONS 4 // 生产者循环次数
#define CONSUMER_ITERATIONS 3 //消费者循环次数

pthread_t consumerId[CONSUMERS], producerId[生产者]; //定义线程

sem_t 空, 满, 互斥; //定义 3 个信号量

类型定义结构{
字符值[BUFFER_SIZE];
int next_in, next_out;
} 缓冲区_t;

buffer_t 缓冲区;


// insertInitial 函数用于向缓冲区添加一个初始值
int insertInitial(char initial, long int id)
{

buffer.value[buffer.next_in] = initial;
buffer.next_in = (buffer.next_in + 1) % BUFFER_SIZE;
printf("producer %ld: produced %c\n", id, initial);

printf("\t\t\t\t\t\t缓冲区:");
for (int i = 0; i < BUFFER_SIZE; i++)
{
```

```
printf("%c", buffer.value[i]);
}
printf("\n");

返回 0;
}

// consumeInitial 函数用于从缓冲区中消费一个初始值
int consumerInitial(char *initial, long int id)
{
     *initial = buffer.value[buffer.next_out];
     buffer.value[buffer.next_out] = '-';
     buffer.next_out = (buffer.next_out + 1) % BUFFER_SIZE;
     printf("\t\t\tconsumer %ld: consumed %c\n", id, *initial);

     printf("\t\t\t\t\t\t缓冲区:");

对于 (int i = 0; i < BUFFER_SIZE; i++) {
printf("%c", buffer.value[i]);
}
printf("\n");

返回 0;
}

// Producer 将迭代 PRODUCER_ITERATIONS 次并调用 insertInitial 函数向缓
冲区插入一个初始值
// Producer argument param 是生产者的整数 id，用于区分多个生产者线程
无效*生产者(无效*参数)
{
字符首字母 [] = "XQL";
int length = strlen(首字母);

初始字符;
long int id = (long int)参数;

int j = PRODUCER_ITERATIONS;
而 (j--)
{
// 等待从 0 到 3 秒的随机时间长度
int randnum = rand() % 4; // 范围在 0 到 3 之间

// 将随机初始值插入缓冲区

int randInitial = rand() % 长度;

sem_wait(&empty);
sem_wait(&mutex);
首字母 = 首字母 [randInitial];
如果 (insertInitial(initial, id))
fprintf(stderr, "插入缓冲区时出错\n");
```

```
sem_post(&mutex);
sem_post(&全);
}

pthread_exit(0);
}

// Consumer 将迭代 CONSUMER_ITERATIONS 次并调用 consumeInitial 函数向
缓冲区插入一个初始值
// Consumer argument param 是消费者的整型 id，用于区分多个消费者线程
无效*消费者(无效*参数)
{
初始字符;
long int id = (long int)参数;

int k = CONSUMER_ITERATIONS;

而 (k--)
{
     睡眠(兰德()%6);

     // 从缓冲区读取
     sem_wait(&满);
     sem_wait(&mutex);
     如果 (consumerInitial(&initial, id))
          fprintf(stderr, "从缓冲区中删除时出错\n");
     sem_post(&mutex);
     sem_post(&empty);
}

pthread_exit(0);
}

主函数()
{
buffer.next_in = 0;
buffer.next_out = 0;

// 用'-'初始化缓冲区

printf("\n 用 '-' 初始化大小为 %d 的缓冲区\n", BUFFER_SIZE);

printf("\t\t\t\t\t\t 缓冲区:");
对于 (int i = 0; i < BUFFER_SIZE; i++) {
buffer.value[i] = '-';
printf("%c", buffer.value[i]);
}
printf("\n");

// 创建生产者和消费者线程

长整数我;
```

```
// 创建消费者线程
printf("\n 正在创建 %d 个消费者和 %d 个生产者\n\n", CONSUMERS, PRODUCERS);

sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&mutex, 0, 1);

// 创建生产者线程
对于 (i = 0; i < 生产者; i++)
如果 (pthread_create(&producerId[i], NULL, 生产者, (void *)i) != 0)
{
错误(“pthread_create”);
中止();
}

for (i = 0; i < 消费者; i++)
如果 (pthread_create(&consumerId[i], NULL, 消费者, (void *)i) != 0)
{
错误(“pthread_create”);
中止();
}

// 等待线程完成

对于 (i = 0; i < 生产者; i++)
如果 (pthread_join(producerId[i], NULL) != 0) {
错误(“pthread_join”);
中止();
}

for (i = 0; i < 消费者; i++)
如果 (pthread_join(consumerId[i], NULL) != 0) {
错误(“pthread_join”);
中止();
}

返回 0;
}
```

随机长度函数

为了生成随机数，使用了标准库函数 RAND() 和 SRAND()。它们在 <stdlib.h> 头文件中定义。

RAND() 的语法是 *int rand(void)*。该函数返回一个整数值，范围从 0 到 RAND_MAX。 RAND_MAX 是定义在 <stdlib.h> 中的符号常量，其取值范围为 0 到 32767。

“种子”是序列的起点。如果他们使用相同的种子，每次运行程序时都会得到相同的数字序列。这就是为什么它是一个“伪随机”函数。

默认情况下，RAND() 函数的种子设置为 1。

RAND() 模 4 设置随机数的范围从 0 到 3。

#include <标准库.h>

无效 getrand（）{

int x = rand() % 4 // 范围在 0 到 3 之间

返回 x

}

睡眠功能

sleep() 函数暂停请求线程的执行，直到参数 seconds 提供的实时秒数已经过去，或者通过调用信号捕获函数或终止进程的动作向调用线程发出信号过去。如果指定的时间段已过，则睡眠函数的返回值应为 0。

线程

可以使用 PTHREAD 在 POSIX 系统中创建线程。需要头文件 <pthread.h>

线程号

每个线程都有一个与之关联的 pthread_t 类型的对象，用于告知其 ID。多个线程不能同时使用同一个 pthread_t 对象。对于多线程，可以创建一个数组，其中每个元素都是一个单独线程的 ID。 *pthread_t id[2]*

创建线程

创建一个线程并使用函数 *pthread_create()*开始。它需要四个参数：

- *pthread_t* 类型的 ID *：对线程 ID 的引用（或指针）
- *pthread_attr_t* *类型的属性：用于设置线程的属性。 NULL 在这个实验中很好
- *void* *类型的起始例程：线程开始执行的函数的名称。如果函数返回类型是 *void* *那么它的名字就简单地写了；否则必须将其类型转换为 void *
- *void* *类型的参数：这是起始例程采用的参数。如果需要多个参数，则使用结构。

起始例程及其参数的返回类型通常设置为 *void* * 。

```
pthread_create(&id[0], NULL, printNumber, &arg);
```

退出线程

*pthread_exit()*用于退出线程。该函数写在启动例程的末尾。由于线程的局部变量在退出时被销毁，因此只返回对全局或动态变量的引用。

等待线程

使用 pthread_join() 使父线程等待子线程。这个函数的两个参数是：

- *pthread_t* 类型的线程 ID：父线程等待的线程 ID。
- Reference to return value *void*** : 退出线程返回的值被这个指针捕获

睡觉

我们必须包含#include <unistd.h>才能在 linux 中使用 sleep()

第二步：编译 C 程序

使用以下命令制作两个程序的可执行版本。

```
$ gcc 生产者消费者.c -o 生产者消费者
```

这将输出 producerconsumer.exe

第五步：运行程序

使用以下命令运行新创建的可执行 .exe 程序。

```
$ ./生产者消费者
```

结果与分析[Linux]

该程序成功地满足了实验的所有要求。它用“-”初始化了一个大小为 4 的缓冲区，创建了 4 个消费者和 3 个生产者。生产者和消费者同步工作，对黄油进行无溢出读写。缓冲区的内容在生产者和消费者线程的每次计算后打印出来。

参考：

- https://www.scaler.com/topics/operating-system/producer-consumer-problem-in-os/
- https://computationstructures.org/lectures/synchronization/synchronization.html
- https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
- https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic
- https://www.educative.io/answers/how-to-create-a-simple-thread-in-c
- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
- https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-rand-generate-random-number
- https://medium.com/@sohamshah456/producer-consumer-programming-with-c-d0d47b8f103f
- https://faculty.cs.niu.edu/~hutchins/csci480/semaphor.htm
- https://learn.microsoft.com/en-gb/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread?redirectedfrom=MSDN
- https://en.wikibooks.org/wiki/Windows_Programming/Handles_and_Data_Types#DWORD,_WORD,_BYTE
- https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/262627d8-3418-4627-9218-4ffe110850b2
- https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea