**操 作 系 统**

**实 验 报 告**

**实验名称：实验四 同步互斥问题**

**姓名：陈亚楠**

**学号：16340041**

实验名称：**同步互斥问题**

1. **实验目的：**

通过两个经典同步问题：生产者－消费者问题、读者－写者问题掌握采用信号量解决同步问题的方法。

**二、实验要求：**

1.生产者－消费者问题：

①设计一个程序来解决有限缓冲问题，其中的生产者与消费者进程如图6.10 与图6.11 所示。

②在6.6.1 小节中，使用了三个信号量: empty (以记录有多少空位)、full (以记录有多少满位)以及mutex (二进制信号量或互斥信号量，以保护对缓冲插入与删除的操作)。对于本项目， empty 与full 将采用标准计数信号量，而mutex 将采用二进制信号量。生产者与消费者作为独立线程，在empty、full、mutex 的同步前提下，对缓冲进行插入与删除。

③本项目，可采用Pthread 。

2.读者－写者问题：

①在Linux环境下，创建一个进程，此进程包含n个线程。用这n个线程来表示n个读者或写者。每个线程按相应测试数据文件(后面有介绍)的要求进行读写操作。用信号量机制分别实现读者优先和写者优先的读者-写者问题。

②读者-写者问题的读写操作限制(仅读者优先或写者优先)：

1)写-写互斥，即不能有两个写者同时进行写操作。

2)读-写互斥，即不能同时有一个线程在读，而另一个线程在写。

3)读-读允许，即可以有一个或多个读者在读。

读者优先的附加限制：如果一个读者申请进行读操作时已有另一个读者正在进行读操作，则该读者可直接开始读操作。

写者优先的附加限制：如果一个读者申请进行读操作时已有另一写者在等待访问共享资源，则该读者必须等到没有写者处于等待状态后才能开始读操作。

③运行结果显示要求：要求在每个线程创建、发出读写操作申请、开始读写操作和结束读写操作时分别显示一行提示信息，以确定所有处理都遵守相应的读写操作限制。

**三、实验过程：**

**１．生产者－消费者问题**

（１）定义缓冲区：

从内部来说,缓冲区是一个元数据类型为buffer\_item (可通过 typedef来定义)的固定大小的数组。而从使用上来说，这个数组可按环形队列来处理。buffer\_item的定义及缓冲区大小可保存在头文件中，如下所示：

*// 缓冲区元数据buffer\_item定义*

typedef int buffer\_item;

*// 缓冲区大小*

#define **BUFFER\_SIZE** 5

缓冲区可通过如下两个函数来实现：inser\_item 与 remove\_item。这两个函数将为生产者和消费者线程所分别使用，其函数结构如下所示：

int **insert\_item**(buffer\_item item)

{

if (count <= BUFFER\_SIZE) {

buffer[rear] = item;

rear = (rear + 1) % BUFFER\_SIZE;

count++;

return 0;

} else {

return -1;

}

}

int **remove\_item**(buffer\_item item)

{

if (count == 0) {

return -1;

} else {

item = buffer[head];

head = (head + 1) % BUFFER\_SIZE;

count--;

return 0;

}

}

（2）声明定义测试数据的结构：

*// 测试数据结构*

typedef struct

{

pthread\_t pthreadId;

int sleepTime;

int keepTime;

buffer\_item productId;

}data;

（3）定义生产者与消费者线程：

①生产者线程：

void \***producer**(void\* param)

{

data\* pthread = (data\*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

buffer\_item productId = pthread->productId;

**free**(pthread);

while(true){

**sleep**(sleepTime);

**sem\_wait**(&empty);

**sem\_wait**(&mutex);

if (**insert\_item**(productId)) {

**printf**("Error, the buffer is full!\n");

**exit**(-1);

} else {

**printf**("Producer pthread %ld produced product %d.\n", pthreadId, productId);

}

**sleep**(keepTime);

**sem\_post**(&mutex);

**sem\_post**(&full);

break;

}

}

②消费者线程：

void \***consumer**(void \*param)

{

data \*pthread = (data \*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

buffer\_item bufferItem;

**free**(pthread);

while(true){

**sleep**(sleepTime);

**sem\_wait**(&full);

**sem\_wait**(&mutex);

if (**remove\_item**(bufferItem))

{

**printf**("Error, The buffer is empty!\n");

**exit**(-1);

}

else

{

**printf**("Consumer pthread %ld consumed product.\n", pthreadId);

}

**sleep**(keepTime);

**sem\_post**(&mutex);

**sem\_post**(&empty);

break;

}

}

（4）定义创建线程函数：

void **createPthread**() {

pthread\_t pthreadId;

char pthreadRole;

for(int i = 0; i < TESTNUMBER; i++)

{

**scanf**("%ld %c ", &pthreadId, &pthreadRole);

data\* pthread = **malloc**(sizeof(data));

pthread->pthreadId = pthreadId;

if(pthreadRole == 'C') {

**scanf**("%d %d", &pthread->sleepTime, &pthread->keepTime);

**pthread\_create**(&pthreadId, &attr, consumer, pthread);

}

else if (pthreadRole == 'P') {

**scanf**("%d %d %d", &pthread->sleepTime, &pthread->keepTime, &pthread->productId);

**pthread\_create**(&pthreadId, &attr, producer, pthread);

} else {

**printf**("Invalid input!\n");

**exit**(-1);

}

}

}

（5）主函数main将初始化缓冲、信号量，创建生产者与消费者线程。在创建完这些线程后，主函数main将睡眠一段时间，并在被唤醒的时候终止应用程序。主函数main的结构如下：

int **main**(int argc, char const \*argv[])

{

*// 初始化信号量*

**sem\_init**(&mutex, 0, 1);

**sem\_init**(&empty, 0, BUFFER\_SIZE);

**sem\_init**(&full, 0, 0);

**pthread\_attr\_init**(&attr);

*//pthread\_t pthreadArray[TESTNUMBER];*

*// 创建进程*

**createPthread**();

**sleep**(60);

**sem\_destroy**(&mutex);

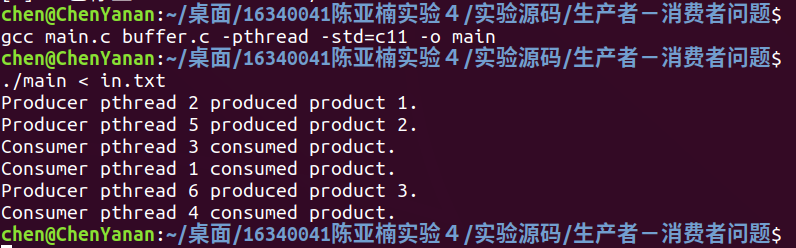
**sem\_destroy**(&empty);

**sem\_destroy**(&full);

return 0;

}

（6）实验结果：



1. **读者－写者问题**
2. 读者优先：

读者优先指的是除非有写者在写文件，否则读者不需要等待。所以可以用一个整型变量read\_count记录当前的读者数目，用于确定是否需要释放正在等待的写者线程(当read\_count=0时，表明所有的读者读完，需要释放写者等待队列中的一个写者)。每一个读者开始读文件时，必须修改read\_count变量。因此需要一个互斥对象mutex来实现对全局变量read\_count修改时的互斥。

另外，为了实现写-写互斥，需要增加一个临界区对象write。当写者发出写请求时，必须申请临界区对象的所有权。通过这种方法，也可以实现读-写互斥，当read\_count=1时(即第一个读者到来时)，读者线程也必须申请临界区对象的所有权。

当读者拥有临界区的所有权时，写者阻塞在临界区对象write上。当写者拥有临界区的所有权时，第一个读者判断完“read\_count==1”后阻塞在write上，其余的读者由于等待对read\_count的判断，阻塞在mutex上。

这里仅强调读者、写者线程的定义：

1. 读者：

void \***reader**(void \*param)

{

data \*pthread = (data \*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

**free**(pthread);

while(true){

**sleep**(sleepTime);

**printf**("Reader pthread %ld wants to read.\n", pthreadId);

**sem\_wait**(&mutex);

readcount++;

if (readcount == 1) {

**sem\_wait**(&wrt);

}

**sem\_post**(&mutex);

**printf**("Reader pthread %ld is reading.\n", pthreadId);

**sleep**(keepTime);

**printf**("Reader pthread %ld finishs to read.\n", pthreadId);

**sem\_wait**(&mutex);

readcount--;

if (readcount == 0) {

**sem\_post**(&wrt);

}

**sem\_post**(&mutex);

break;

}

}

实验结果：



1. 写者：

void \***writer**(void \*param)

{

data \*pthread = (data \*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

**free**(pthread);

while(true){

**sleep**(sleepTime);

**printf**("Writer pthread %ld wants to write.\n", pthreadId);

**sem\_wait**(&wrt);

**printf**("Writer pthread %ld is writing.\n", pthreadId);

**sleep**(keepTime);

**printf**("Writer pthread %ld finishs to write.\n", pthreadId);

**sem\_post**(&wrt);

break;

}

}

1. 写者优先：

写者优先与读者优先类似。不同之处在于一旦一个写者到来，它应该尽快对文件进行写操作，如果有一个写者在等待，则新到来的读者不允许进行读操作。为此应当添加一个整型变量write\_count，用于记录正在等待的写者的数目，当write\_count=0时，才可以释放等待的读者线程队列。

为了对全局变量write\_count实现互斥，必须增加一个互斥对象mutex2。

为了实现写者优先，应当添加一个临界区对象read，当有写者在写文件或等待时，读者必须阻塞在read上。同样，有读者读时，写者必须等待。于是，必须有一个互斥对象RW\_mutex来实现这个互斥。

有写者在写时，写者必须等待。

读者线程要对全局变量read\_count实现操作上的互斥，必须有一个互斥对象命名为mutex1。

这里仅强调读者、写者线程的定义：

1. 读者：

void \***writer**(void \*param)

{

data \*pthread = (data \*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

**free**(pthread);

while (true)

{

**sleep**(sleepTime);

**printf**("Writer pthread %ld wants to write.\n", pthreadId);

**sem\_wait**(&writeMutex);

writecount++;

if(writecount == 1) {

**sem\_wait**(&rd);

}

**sem\_post**(&writeMutex);

**sem\_wait**(&wrt);

**printf**("Writer pthread %ld is writing.\n", pthreadId);

**sleep**(keepTime);

**printf**("Writer pthread %ld finishs to write.\n", pthreadId);

**sem\_post**(&wrt);

**sem\_wait**(&writeMutex);

writecount--;

if (writecount == 0)

{

**sem\_wait**(&rd);

}

**sem\_post**(&writeMutex);

break;

}

}

1. 写者：

void \***reader**(void \*param)

{

data \*pthread = (data \*)param;

pthread\_t pthreadId = pthread->pthreadId;

int sleepTime = pthread->sleepTime;

int keepTime = pthread->keepTime;

**free**(pthread);

while (true)

{

**sleep**(sleepTime);

**printf**("Reader pthread %ld wants to read.\n", pthreadId);

**sem\_wait**(&rd);

**sem\_wait**(&readMutex);

readcount++;

if (readcount == 1)

{

**sem\_wait**(&wrt);

}

**sem\_post**(&readMutex);

**sem\_post**(&rd);

**printf**("Reader pthread %ld is reading.\n", pthreadId);

**sleep**(keepTime);

**printf**("Reader pthread %ld finishs to read.\n", pthreadId);

**sem\_wait**(&readMutex);

readcount--;

if (readcount == 0)

{

**sem\_post**(&wrt);

}

**sem\_post**(&readMutex);

break;

}

}

实验结果：

