|  |  |  |  |
| --- | --- | --- | --- |
| **《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 |  |
| 一、实验题目  生产者—消费者问题  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 Windows API 实现。  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。  如前所述，您可以使用 Pthreads 或 Windows API 解决此问题。在以下部分中，我们将提供有关每个选择的更多信息。 | | | |
| 二、相关原理与知识   1. pthread   Pthread是由POSIX标准(IEEE 1003.1c)为线程创建和同步定义的API。这是线程行为的规范，而不是实现。  所有Pthread程序都需要包括pthread.h头文件。语句pthread\_t tid 声明了所创建线程的标识符。每个线程都有一组属性， 包括栈大小和调度信息。pthread\_attr\_t atr表示线程的属性，通过函数调用pthread\_attr\_init(&atr)来设置这些属性。若没有明确地设置任何属性,则使用提供的默认属性。通过函数调用pthread\_create()来创建一个独立线程。除了传递线程标识符和线程属性外，还要传递函数名称，以便新线程可以开始执行。最后传递特定函数的参数。   1. 互斥锁   互斥锁是一种常用的同步机制，用于控制多个线程访问共享资源的顺序，以避免数据竞争和死锁等并发问题。  互斥锁的基本原理是，在每个共享资源的访问前，请求线程必须先获得该资源的互斥锁，以独占式地进行访问；访问结束后释放锁，使其他线程能够访问该资源。这样保证了只有一个线程访问该资源，从而避免了并发冲突。  在C语言关于互斥锁的编程中，常用的函数如下：  pthread\_mutex\_init：用于初始化互斥锁对象。  函数原型为：int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr)，其中mutex为互斥锁对象指针，attr为互斥锁属性指针。如果attr为NULL，则使用默认属性初始化互斥锁。  pthread\_mutex\_lock：用于加锁互斥锁对象。  函数原型为：int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)，其中mutex为互斥锁对象指针。如果该锁已经被另一个线程锁定，则调用线程将阻塞，直到该锁被释放为止。  pthread\_mutex\_trylock：用于尝试加锁互斥锁对象，如果该锁已经被另一个线程锁定，则返回EBUSY错误。  函数原型为：int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex)，其中mutex为互斥锁对象指针。  pthread\_mutex\_unlock：用于解锁互斥锁对象。  函数原型为：int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)，其中mutex为互斥锁对象指针。如果该锁没有被锁定，则函数将返回EPERM错误。   1. 信号量   信号量是一种同步机制，用于控制多个线程对共享资源的访问，与互斥锁不同的是，信号量可以允许多个线程同时访问资源，但需要根据资源的可用数量来控制并发量。  信号量的基本原理是，对于每个共享资源，设置一个信号量计数器，表示该资源的可用数量，线程在访问该资源前需要获取信号量，如果信号量计数器为正数，则线程可以继续访问资源，并将信号量减1，如果信号量计数器为0，则线程需要等待其他线程释放资源后，才能继续访问。  在C语言关于信号量的编程中，常用函数包括：  sem\_init：用于初始化信号量对象。  函数原型为：int sem\_init(sem\_t \*sem, int pshared, unsigned int value)，其中sem为信号量对象指针，pshared表示信号量是否在进程间共享，value为信号量初始计数值。  sem\_wait：用于等待信号量，如果信号量计数器为0，则线程将阻塞，直到其他线程释放资源。  函数原型为：int sem\_wait(sem\_t \*sem)，其中sem为信号量对象指针。  sem\_post：用于释放信号量，将信号量计数器加1。  函数原型为：int sem\_post(sem\_t \*sem)，其中sem为信号量对象指针。  sem\_destroy：用于销毁信号量。  函数原型为：int sem\_destroy(sem\_t \*sem)，其中 sem 为信号量对象指针。在销毁信号量之前，需要确保所有线程都已经释放了该信号量，否则可能导致未定义的行为。  需要注意的是，信号量需要谨慎使用，特别是在高并发环境下，过度使用信号量可能会导致性能问题，甚至引发死锁等并发问题。 | | | |
| 三、实验过程  程序设计的整体思路为，使用互斥锁和信号量实现producer-consumer问题；故大致操作即创建producer和consumer线程，每个线程的函数使用死循环执行操作，每次执行完所有操作后进行随机时间的休眠，这样可以模拟不同线程、随机时间的访问缓存的冲突和互斥锁、信号量对该问题的解决情况。  程序的具体设计，可分为变量及互斥锁和信号量的定义与初始化、线程函数的创建、线程的创建、互斥锁和信号量的销毁四个部分，代码的具体解释也将按照这四个部分展开。   1. 变量及互斥锁和信号量的定义与初始化   对于producer-consumer问题，其主要是针对缓冲区中数据的produce和consume并使用互斥锁和信号量解决其中的多线程访问有界缓冲区冲突问题。故首先要对模拟的有界缓冲区、互斥锁和信号量进行定义，代码如下：   |  | | --- | | #define BUFFER\_SIZE 10    typedef struct {  int buffer[BUFFER\_SIZE];  int in;  int out;  } BoundedBuffer;    BoundedBuffer bb;  sem\_t empty, full;  pthread\_mutex\_t mutex; |   可以看到，上述代码首先定义了有界缓冲区的最大长度，此处为10；之后使用结构体构造缓冲区，其中包括缓冲区buffer本身，其中的in和out分别指示produce和consume线程操作缓冲区的位置，即若当in的值为5时，此时的produce线程会尝试往buffer[5]中插入产生的数据。  为了防止多线程对缓冲区的访问冲突，此处定义了互斥锁mutex和信号量empty、full。当producer和consumer线程尝试访问缓冲区时，它们必须首先获取互斥锁，以避免竞争条件。然后，它们必须等待相应的信号量，以确保缓冲区中有足够的空间或物品可供访问。  信号量的使用即producer线程使用full，consumer线程使用empty，来判断此时的缓冲区是否为满或空，以判断此时缓冲区中是否还能存放producer产生的数据或是否还存在数据供consumer消费。full信号量表示缓冲区中可用的物品数量，它的初始值为0。当消费者线程从缓冲区中移除一个物品时，它会通过调用sem\_post(&full)增加full信号量的值，表示缓冲区中多了一个可用的物品。如果full信号量的值已经达到缓冲区的最大容量，则生产者线程需要等待，直到有一个物品被消费并释放出空间。empty信号量表示缓冲区中可用的空间数量，它的初始值为BUFFER\_SIZE。当生产者线程向缓冲区中添加一个物品时，它会通过调用sem\_post(&empty)增加empty信号量的值，表示缓冲区中多了一个空闲位置。如果empty信号量的值已经为0，则消费者线程需要等待，直到有一个物品被生产并释放出空间。  互斥锁的使用即在线程访问缓冲区之前，要首先尝试lock该缓冲锁；若此时其他线程正在运行并访问缓冲区，互斥锁会处于locked状态，无法再次加锁，此时该线程会进行等待，当互斥锁被解锁以后再运行。若有多个线程等待互斥锁解锁，此时在linux系统中默认使用Completely Fair Scheduler调度算法来对所有正在等待的线程进行调度，实现程序运行的高效和线程之间的公平。  缓冲区、信号量和互斥锁均需要初始化，此过程在main函数中完成，其代码如下：   |  | | --- | | bb.in = 0;  bb.out = 0;  seminit(&empty, 0, BUFFERSIZE);  seminit(&full, 0, 0);  pthreadmutexinit(&mutex, NULL); |   缓冲区的in和out均置0；empty的信号量初始值为BUFFERSIZE且不在进程间共享；full信号量初始值为0，且不在进程间共享；mutex互斥锁初始化。   1. 线程函数的创建   在之前的实验中已经提到过，创建线程时需要一个参数和返回值均为void\*类型的变量，因此我们需要定义对于producer和consumer线程的线程函数。  对于producer线程，具体代码如下：   |  | | --- | | void\* producer(void\* arg) {  int producer\_id = \*(int\*)arg;  while (1) {  int item = rand() % 1000;    sem\_wait(&empty);  pthread\_mutex\_lock(&mutex);    bb.buffer[bb.in] = item;  bb.in = (bb.in + 1) % BUFFER\_SIZE;    printf("Producer %d produced: %d**\n**", producer\_id, item);  display\_buffer();    pthread\_mutex\_unlock(&mutex);  sem\_post(&full);    sleep(rand() % 5);  }  } |   传入的参数args为producer线程的id，用以在输出时展示是哪个producer线程产生的数据。该函数随后进入死循环，其中随机产生一个0-1000的数字作为等待插入缓冲区的值，当互斥锁和信号量检测均通过后，方可访问缓冲区；代码中实现缓冲区插入item的方法为直接令buffer[in]=item，之后更新in值。随后输出该操作，即XX producer线程产生了item值，并展示当前的缓冲区情况。  对于consumer线程，具体代码如下：   |  | | --- | | void\* consumer(void\* arg) {  int consumer\_id = \*(int\*)arg;  while (1) {  sem\_wait(&full);  pthread\_mutex\_lock(&mutex);    int item = bb.buffer[bb.out];  bb.buffer[bb.out] = 0;  bb.out = (bb.out + 1) % BUFFER\_SIZE;    printf("Consumer %d consumed: %d**\n**", consumer\_id, item);  display\_buffer();    pthread\_mutex\_unlock(&mutex);  sem\_post(&empty);    sleep(rand() % 5);  }  } |   与producer线程类似，传入的参数args为consumer线程的id，用以在输出时展示是哪个producer线程产生的数据。该函数随后进入死循环，当互斥锁和信号量检测均通过后，方可访问缓冲区，删除缓冲区指定位置的值，即将其置为0；代码中实现缓冲区删除元素的方法为直接令buffer[out]=0，之后更新out值。随后输出该操作，即XX consumer线程消费了item值，并展示当前的缓冲区情况。  对于producer线程函数和consumer线程函数中均涉及的展示缓冲区函数，即display\_buffer，其定义如下：   |  | | --- | | void display\_buffer() {  printf("Buffer: ");  for (int i = 0; i < BUFFER\_SIZE; i++) {  printf("%-5d ", bb.buffer[i]);  }  printf("**\n\n**");  } |   主要是遍历当前缓冲区并按照较美观的方式输出。   1. 线程的创建   创建线程的操作及具体解释，在上次实验中已经诠释过，所以此处做简单介绍。  线程创建的代码如下：   |  | | --- | | if(argc != 4)  {  printf("Usage: <sleep\_time> <producer\_number> <consumer\_number>**\n**");  exit(EXIT\_FAILURE);  }    int sleep\_time = atoi(argv[1]);  int producer\_number = atoi(argv[2]);  int consumer\_number = atoi(argv[3]);    pthread\_t producer\_threads[producer\_number], consumer\_threads[consumer\_number];  int producer\_ids[producer\_number], consumer\_ids[consumer\_number];    for (int i = 0; i < producer\_number; i++) {  producer\_ids[i] = i + 1;  pthread\_create(&producer\_threads[i], NULL, producer, &producer\_ids[i]);  }    for (int i = 0; i < consumer\_number; i++) {  consumer\_ids[i] = i + 1;  pthread\_create(&consumer\_threads[i], NULL, consumer, &consumer\_ids[i]);  } |   此处还包括了main函数的传参部分。通过\*argv数组，我们获取程序的运行时间sleep\_time、producer的线程数量producer\_number和consumer的线程数量consumer\_numer。之后创建线程的id和便于展示的producer\_id、consumer\_id，并使用线程id、线程函数和线程参数的结构体（此处只有一个参数故没有创建结构体）完成线程的创建。   1. 互斥锁和信号量的销毁   当进程结束时，需要对互斥锁和信号量完成销毁。但我们定义线程时使用的线程函数为死循环函数，此处为了不让进程一直运行下去，我们选择不使用pthread\_join函数等待线程结束并回收，取而代之的是使用sleep函数让进程休眠一段时间。在休眠期间线程照常运行，一旦休眠结束，我们就对互斥锁和信号量完成销毁。  上述操作的代码如下：   |  | | --- | | sleep(sleep\_time);    pthread\_mutex\_destroy(&mutex);  sem\_destroy(&empty);  sem\_destroy(&full); | | | | |
| 四、实验结果与分析  我们针对上述代码，执行了三次实验操作，其具体如下：    此为producer和consumer线程均为5个，程序运行3秒的结果图，可以观察到，在互斥锁的作用下，不会出现生产者消费者同时修改缓冲区的情况；当没有empty资源后，生产者线程不再进行生产操作，直到消费者线程释放empty资源；当没有full资源后消费者不再消耗，直到生产者释放full资源。  剩下两次实验均运行3秒，但其中一个producer线程为3个，consumer线程为5个；另一个producer线程为4个，consumer线程为3个，具体情况类似，如下图：      可以观察到，在第二个实验producer线程为3个，consumer线程为5个时，产生的操作数目明显减少，且缓冲区中0的个数始终较多，这更可以说明在对缓冲区进行操作时由于empty信号量，consumer大多数时候在等待producer产生数据。这刚好对应了consumer线程数目较多的情况。  而第三个实验，producer线程为4个，consumer线程为3个时，明显观察到操作个数增多，且缓冲区0的个数在各个时段对比第二个实验均较少，这也对应了producer线程数目较多的情况。 | | | |
| 五、问题总结   1. 在使用银河麒麟系统编译 c 文件时，报错undefined reference to pthread\_create。经过查阅资料发现这是由于 pthread 不是 linux 下的默认的库，也就是在链接的时候，无法找到 phread 库中该函数的入口地址，于是链接会失败。   解决:在编译指令结尾加上 -lpthread即可，即使用gccthread.c-othread-lpthread 指令进行编译即可。   1. 问题：误以为wait()、post()等操作可直接使用   解决：使用semaphore.h中的函数int sem\_wait(sem\_t \*sem)和int sem\_post(sem\_t \*sem) | | | |
| 六、源代码  #include <pthread.h>  #include <semaphore.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #define BUFFER\_SIZE 10  typedef struct {  int buffer[BUFFER\_SIZE];  int in;  int out;  } BoundedBuffer;  BoundedBuffer bb;  sem\_t empty, full;  pthread\_mutex\_t mutex;  void display\_buffer() {  printf("Buffer: ");  for (int i = 0; i < BUFFER\_SIZE; i++) {  printf("%-5d ", bb.buffer[i]);  }  printf("\n\n");  }  void\* producer(void\* arg) {  int producer\_id = \*(int\*)arg;  while (1) {  int item = rand() % 1000;  sem\_wait(&empty);  pthread\_mutex\_lock(&mutex);  bb.buffer[bb.in] = item;  bb.in = (bb.in + 1) % BUFFER\_SIZE;  printf("Producer %d produced: %d\n", producer\_id, item);  display\_buffer();  pthread\_mutex\_unlock(&mutex);  sem\_post(&full);  sleep(rand() % 5);  }  }  void\* consumer(void\* arg) {  int consumer\_id = \*(int\*)arg;  while (1) {  sem\_wait(&full);  pthread\_mutex\_lock(&mutex);  int item = bb.buffer[bb.out];  bb.buffer[bb.out] = 0;  bb.out = (bb.out + 1) % BUFFER\_SIZE;  printf("Consumer %d consumed: %d\n", consumer\_id, item);  display\_buffer();  pthread\_mutex\_unlock(&mutex);  sem\_post(&empty);  sleep(rand() % 5);  }  }  int main(int argc, char \*argv[]) {    if(argc != 4)  {  printf("Usage: <sleep\_time> <producer\_number> <consumer\_number>\n");  exit(EXIT\_FAILURE);  }  int sleep\_time = atoi(argv[1]);  int producer\_number = atoi(argv[2]);  int consumer\_number = atoi(argv[3]);  pthread\_t producer\_threads[producer\_number], consumer\_threads[consumer\_number];  int producer\_ids[producer\_number], consumer\_ids[consumer\_number];  bb.in = 0;  bb.out = 0;  sem\_init(&empty, 0, BUFFER\_SIZE);  sem\_init(&full, 0, 0);  pthread\_mutex\_init(&mutex, NULL);  for (int i = 0; i < producer\_number; i++) {  producer\_ids[i] = i + 1;  pthread\_create(&producer\_threads[i], NULL, producer, &producer\_ids[i]);  }  for (int i = 0; i < consumer\_number; i++) {  consumer\_ids[i] = i + 1;  pthread\_create(&consumer\_threads[i], NULL, consumer, &consumer\_ids[i]);  }  sleep(sleep\_time);  pthread\_mutex\_destroy(&mutex);  sem\_destroy(&empty);  sem\_destroy(&full);  return 0;  } | | | |