作业 8

8.1 银行有 n 个柜员,每个顾客进入银行后先取一个号,并且等着叫号,当一个柜员空闲后,就叫下一个号.

请使用 PV 操作分别实现:

//顾客取号操作 Customer_Service

//柜员服务操作 Teller_Service

重温一下互斥和同步的概念：

互斥：是指散布在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行。最基本的场景就是：一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。

同步：是指散布在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。最基本的场景就是：两个或两个以上的进程或线程在运行过程中协同步调，按预定的先后次序运行。比如 A 任务的运行依赖于 B 任务产生的数据。

也就是说互斥是两个任务之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程。因此互斥具有唯一性和排它性，但互斥并不限制任务的运行顺序，即任务是无序的，而同步的任务之间则有顺序关系。

而对于该题，则是一个典型的生产者消费者问题，生产者消费者问题，是多线程同步的一个经典问题。这个问题描述的场景是对于一个有固定大小的缓冲区，同时共享给两个线程去使用。

这两个线程会分为两个角色：

- 一个负责往这个缓冲区里放入一定的数据，我们叫他生产者。
- 另一个负责从缓冲区里取数据，我们叫他消费者。

这里就会有两个问题：

- 第一个问题是生产者不可能无限制的放数据去缓冲区，因为缓冲区是有大小的，当缓冲区满的时候，生产者就必须停止生产。
- 第二个问题亦然，消费者也不可能无限制的从缓冲区去取数据，取数据的前提是缓冲区里有数据，所以当缓冲区空的时候，消费者就必须停止生产。

这个问题涉及到两类独立操作的同步：一类是生产（顾客取号），另一类是消费（柜员服务顾客）。

在该场景中，顾客扮演了“生产者”的角色，他们“生产”号码。柜员扮演了“消费者”的角色，他们“消费”这些号码。这两个操作不能同时进行，因此需要进行同步。当没有顾客（即没有号码）时，柜员需要等待。当所有柜员都在服务顾客时，新来的顾客需要等待。

可以利用一段 C 语言代码实现上述功能：

```c
    #define __USE_GNU
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t customer; // 顾客信号量
sem_t teller;   // 柜员信号量
int number = 0; // 号码

void* Customer_Service(void* data) {
    while (1) {
        number++; // 顾客取号
        printf("Customer took number %d\n", number);
        sem_post(&customer); // 增加一个顾客信号量
    }
    return NULL;
}

void* Teller_Service(void* data) {
    while (1) {
        sem_wait(&customer); // 等待顾客
        printf("Teller is serving customer %d\n", number);
        number--; // 服务完毕，减少一个号码
        sem_post(&teller); // 增加一个柜员信号量
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    sem_init(&customer, 0, 0);
    sem_init(&teller, 0, 0);

    pthread_create(&thread1, NULL, Customer_Service, NULL);
    pthread_create(&thread2, NULL, Teller_Service, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

sem 系列的函数是用于操作信号量的。sem_post 和 sem_wait 函数在这个程序中被用于同步顾客和柜员的行为。

- `sem_post(&semaphore)`：此函数会增加信号量的值。如果有其他线程在等待这个信号量，那么其中一个线程会被唤醒。在这个程序中，`sem_post(&customer)`表示有新的顾客取号，`sem_post(&teller)`表示柜员已经服务完一个顾客。

- `sem_wait(&semaphore)`：此函数会减少信号量的值。如果信号量的值为 `0`，那么这个函数会阻塞，直到信号量的值大于 `0`。在这个程序中，`sem_wait(&customer)`表示柜员正在等待新的顾客。

`sem_t` 是一个信号量数据类型，用于控制多个线程之间的访问。信号量是一种同步原语，用于协调并发线程的执行。
其运行结果如下：

```
Customer took number 9064
Customer took number 9065
Customer took number 9066
Customer took number 9067
Customer took number 9068
Customer took number 9069
Customer took number 9070
Customer took number 9071
Customer took number 9072
Customer took number 9073
Customer took number 9074
Customer took number 9075
Customer took number 9076
Customer took number 9077
Teller is serving customer 9063
Teller is serving customer 9077
Teller is serving customer 9076
Teller is serving customer 9075
Teller is serving customer 9074
Teller is serving customer 9073
Teller is serving customer 9072
Teller is serving customer 9071
Teller is serving customer 9070
Teller is serving customer 9069
Teller is serving customer 9068
```

可以看到生产者和消费者的操作是互斥的 ，只有当生产者取号后，消费者才能开始服务。但是取号数越来越多，但消费者却没有服务到所有顾客，这是因 `Customer_Service` 函数和 `Teller_Service` 函数是在不同的线程中运行的，它们的执行速度会不同。如果顾客取号的速度快于柜员服务的速度，那么 `number` 的值就会持续增加，因为新的顾客在取号，但柜员还没有来得及服务他们。

另外上述代码中其实并未考虑到柜员空闲的情况，也就是说顾客只会一直取号，等待被消费，并不考虑是否存在消费者的情况。这是假设顾客不需要等待任何资源，但是柜员不能没有顾客，不过真实的情况下，顾客不能无限制的生产资源（号数），因此需要加上限制。同时还存在一个问题，这是因为在两个线程中，都在访问和修改全局变量 `number`。这可能会导致不一致的结果，因为一个线程可能在另一个线程完成其操作之前开始执行其操作。因此还需要引入一个互斥锁（`mutex`），以确保在同一时间只有一个线程可以访问和修改 `number`。

最后为了保证两个线程执行速度大致相同，可以用一个简单的办法，即可以在每个线程的循环中添加一个延时。这样可以模拟现实生活中的情况，即顾客需要一些时间来取号，柜员也需要一些时间来服务顾客。我们可以使用 usleep 函数来实现这个延时。

代码修改如下：

```
#define __USE_GNU
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define NUM_TELLERS 3 // 柜员数量
sem_t customer; // 顾客信号量
sem_t teller;   // 柜员信号量
int number = 0; // 号码
pthread_mutex_t lock; // 定义互斥锁
void* Customer_Service(void* data) {
    while (1) {
        sem_wait(&teller); // 等待柜员
        usleep(1000000); // 延时 1 秒
        pthread_mutex_lock(&lock); // 获取互斥锁
        number++; // 顾客取号
        printf("Customer took number %d\n", number);
        pthread_mutex_unlock(&lock); // 释放互斥锁
        sem_post(&customer); // 增加一个顾客信号量
    }
    return NULL;
}

void* Teller_Service(void* data) {
    int teller_number = *(int*)data; // 获取柜员编号
    while (1) {
        sem_wait(&customer); // 等待顾客
        usleep(1000000); // 延时 1 秒
        pthread_mutex_lock(&lock); // 获取互斥锁
        printf("Teller %d is serving customer %d\n", teller_number,
number);
        number--; // 服务完毕，减少一个号码
        pthread_mutex_unlock(&lock); // 释放互斥锁
        sem_post(&teller); // 增加一个柜员信号量
    }
    return NULL;
}
int main() {
    pthread_t customers;
    pthread_t tellers[NUM_TELLERS];
```

```
    int teller_numbers[NUM_TELLERS]; // 存储柜员编号的数组
    sem_init(&customer, 0, 0);
    sem_init(&teller, 0, NUM_TELLERS); // 初始时所有的柜员都是空闲的
    pthread_mutex_init(&lock, NULL);

    pthread_create(&customers, NULL, Customer_Service, NULL);
    // 创建柜员线程
    for (int i = 0; i < NUM_TELLERS; i++) {
        teller_numbers[i] = i + 1; // 初始化柜员编号
        pthread_create(&tellers[i], NULL, Teller_Service,
&teller_numbers[i]); // 创建柜员线程，传递柜员编号
    }

    pthread_join(customers, NULL);
    for (int i = 0; i < NUM_TELLERS; i++) {
        pthread_join(tellers[i], NULL);
    }

    return 0;
}
```

执行结果如下：

```
Teller 1 is serving customer 1
Customer took number 1
Customer took number 2
Teller 2 is serving customer 2
Teller 2 is serving customer 1
Customer took number 1
Customer took number 2
Teller 1 is serving customer 2
Customer took number 2
Teller 3 is serving customer 2
Teller 3 is serving customer 1
Customer took number 1
Customer took number 2
Teller 1 is serving customer 2
```

8.2 多个线程的规约(Reduce)操作是把每个线程的结果按照某种运算(符合交换律和结合律) 两两合并直到得到最终结果的过程。

试设计管程 monitor 实现一个 8 线程规约的过程，随机初始化 16 个整数，每个线程通过调用 monitor.getTask 获得 2 个数，相加后，返回一个数 monitor.putResult ，然后再 getTask( ) 直到全部完成退出，最后打印归约过程和结果。

要求：为了模拟不均衡性，每个加法操作要加上随机的时间扰动，变动区间 1~10ms。

提示：使用 pthread_系列的 cond_wait, cond_signal, mutex 实现管程；使用 rand( ) 函数产生随机数，和随机执行时间。

相关概念：

规约：

在多线程编程中，“规约”（Reduction）是一种基础的并行算法。简单来说，我们有N个输入数据，使用一个符合结合律的二元操作符作用其上，最终生成1个结果。这个二元操作符可以是求和、取最大、取最小、平方、逻辑与或等等。

对于该问题，规约的过程是多个线程通过执行加法操作（这是一个二元操作符），将16个整数两两相加，最终得到一个结果。这个过程涉及到线程间的同步，因为每个线程需要等待获取两个数，然后执行加法操作，再将结果返回，然后再获取下一对数，如此循环，直到所有的数都被加完。

管程：

管程（Monitor）是一种用于管理并发程序中的共享资源的高级同步机制。它可以看作是一个程序结构，该结构内的多个子程序（对象或模块）形成的多个工作线程互斥访问共享资源。共享资源一般是硬件设备或一群变量。管程的主要特性是在一个时间点，最多只有一个线程在执行管程的某个子程序。这就保证了对共享资源的互斥访问。

管程（Monitor）的语义主要涉及到以下几个方面：

1. 互斥访问：在一个时间点，最多只有一个线程在执行管程的某个子程序123。这就保证了对共享资源的互斥访问。
2. 同步：管程提供了一种机制，线程可以临时放弃互斥访问，等待某些条件得到满足后，重新获得执行权恢复它的互斥访问。这就涉及到了条件变量的概念，每个条件变量都对应有一个等待队列。通过条件通知去唤醒等待队列的线程竞争锁资源。
3. 封装：管程是定义了一个数据结构和能为并发所执行的一组操作，这组操作能够进行同步和改变管程中的数据。这相当于对临界资源的同步操作都集中进行管理，凡是要访问临界资源的进程或线程，都必须先通过管程，由管程的这套机制来实现多进程或线程对同一个临界资源的互斥访问和使用。

C语言中，可以使用`pthread`库来实现管程。以下是一些与管程相关的API：

1. pthread_mutex_init：初始化一个互斥锁。
2. pthread_mutex_destroy：销毁一个互斥锁。
3. pthread_mutex_lock：锁定一个互斥锁。
4. pthread_mutex_unlock：解锁一个互斥锁。
5. pthread_cond_init：初始化一个条件变量。
6. pthread_cond_destroy：销毁一个条件变量。
7. pthread_cond_wait：等待一个条件变量。

8. pthread_cond_signal：激活一个等待该条件的线程，存在多个等待线程时按入队顺序激活其中一个；

10. pthread_cond_broadcast 释放了所有堵在 cond 上的进程。

C 语言代码如下：

```c
#define __USE_GNU
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 8
#define NUM_INTS 16

typedef struct {
    int tasks[NUM_INTS];
    int task_index;
    int results[NUM_INTS];
    int result_index;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} monitor_t;

void monitor_init(monitor_t* monitor) {
    for (int i = 0; i < NUM_INTS; i++) {
        monitor->tasks[i] = rand() % 100;
    }
    monitor->task_index = NUM_INTS;
    monitor->result_index = 0;
    pthread_mutex_init(&monitor->mutex, NULL);
    pthread_cond_init(&monitor->cond, NULL);
}

void monitor_destroy(monitor_t* monitor) {
    pthread_mutex_destroy(&monitor->mutex);
    pthread_cond_destroy(&monitor->cond);
}

void monitor_getTask(monitor_t* monitor, int* num1, int* num2) {
    pthread_mutex_lock(&monitor->mutex);
    while (monitor->task_index < 2 && monitor->result_index < 2) {
        pthread_cond_wait(&monitor->cond, &monitor->mutex);
    }
    if (monitor->task_index >= 2) {
```

```
        *num1 = monitor->tasks[--monitor->task_index];
        *num2 = monitor->tasks[--monitor->task_index];
    } else if (monitor->result_index >= 2) {
        *num1 = monitor->results[--monitor->result_index];
        *num2 = monitor->results[--monitor->result_index];
    } else {
        *num1 = *num2 = 0;
    }
    pthread_mutex_unlock(&monitor->mutex);
}

void monitor_putResult(monitor_t* monitor, int result) {
    pthread_mutex_lock(&monitor->mutex);
    monitor->results[monitor->result_index++] = result;
    pthread_cond_broadcast(&monitor->cond);
    pthread_mutex_unlock(&monitor->mutex);
}

void* thread_func(void* arg) {
    monitor_t* monitor = (monitor_t*)arg;
    while (1) {
        int num1, num2;
        monitor_getTask(monitor, &num1, &num2);
        if (num1 == 0 && num2 == 0) {
            break;
        }
        usleep((rand() % 10 + 1) * 1000); // 随机时间扰动
        int sum = num1 + num2;
        printf("Thread %ld added %d and %d to get %d\n",
(long)pthread_self(), num1, num2, sum);
        monitor_putResult(monitor, sum);
    }
    return NULL;
}

int main() {
    srand(time(NULL));
    pthread_t threads[NUM_THREADS];
    monitor_t monitor;

    monitor_init(&monitor);

    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_func, &monitor);
```

```
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final result: %d\n", monitor.results[0]);

    monitor_destroy(&monitor);

    return 0;
}
```

比较重要的是 monitor_getTask 的函数，该函数接受指向 monitor_t 结构体的指针，以及两个名为 num1 和 num2 的整型指针作为参数。在函数内部，使用 pthread_mutex_lock 函数锁定了 monitor 结构体中的互斥锁。然后，使用 while 循环来等待任务队列中有足够的任务，或者结果队列中已经有了足够的结果。在等待期间，使用 pthread_cond_wait 函数来阻塞线程并等待条件变量 cond 的信号。

如果结果队列中已经有了足够的结果（这里是指 result 都生成足够了，满足最后只有一个 result 时），则将 num1 和 num2 的值都设置为 0，并使用 pthread_mutex_unlock 函数解锁互斥锁并返回。否则，从任务队列中取出两个任务，并将它们分别赋值给 num1 和 num2。最后，使用 pthread_mutex_unlock 函数解锁互斥锁并返回。

运行结果如下：

```
Thread 140737351636736 added 80 and 25 to get 105
Thread 140737343244032 added 77 and 96 to get 173
Thread 140737334851328 added 5 and 13 to get 18
Thread 140737351636736 added 44 and 20 to get 64
Thread 140737343244032 added 53 and 30 to get 83
Thread 140737326458624 added 4 and 26 to get 30
Thread 140737351636736 added 46 and 47 to get 93
Thread 140737334851328 added 0 and 83 to get 83
```

但是这串代码只实现了八个线程互斥地加，其在八个结果生成后没有继续规约地进行加法操作，无法终止程序。这是因为，每个线程在完成所有任务后会检查是否还有任务可以执行。如果没有，它们就会退出。然而，由于我们没有一个明确的信号来告诉线程所有的规约操作都已经完成，所以线程可能会 monitor_getTask 函数中无限期地等待新的任务。

因此可以在 monitor_getTask 函数中添加了一个检查，以确定是否所有的规约操作都已经完成。如果所有的规约操作都已经完成，我们就让线程退出。同时添加了指示，以表明参与加法的操作数是原本的初值还是后面进行加法的产生的结果。

修改后的代码如下：

```c
#define __USE_GNU
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 8
#define NUM_INTS 16

/**
 * @brief 定义了一个监视器类型 monitor_t，包含了任务数组、任务索引、结果数组、结果索引、互斥锁和条件变量。
 *
 */
typedef struct {
    int tasks[NUM_INTS];
    int task_index;
    int results[NUM_INTS];
    int result_index;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} monitor_t;

/**
 * @brief 初始化监视器
 *
 * @param monitor 监视器指针
 */
void monitor_init(monitor_t* monitor) {
    for (int i = 0; i < NUM_INTS; i++) {
        monitor->tasks[i] = rand() % 100;
    }
    monitor->task_index = NUM_INTS;
    monitor->result_index = 0;
    pthread_mutex_init(&monitor->mutex, NULL);
    pthread_cond_init(&monitor->cond, NULL);
}

/**
 * @brief 销毁监视器
 *
 * @param monitor 监视器指针
 */
void monitor_destroy(monitor_t* monitor) {
```

```
    pthread_mutex_destroy(&monitor->mutex);
    pthread_cond_destroy(&monitor->cond);
}

/**
 * 从监视器中获取任务
 * @param monitor 监视器
 * @param num1 存储第一个数字的指针
 * @param num2 存储第二个数字的指针
 * @param is_result 表明操作数是否为初值的指针
 */
void monitor_getTask(monitor_t* monitor, int* num1, int* num2, int*
is_result) {
    pthread_mutex_lock(&monitor->mutex);
    while (monitor->task_index < 2 && monitor->result_index < 2) {
        if (monitor->task_index == 0 && monitor->result_index == 1) {
            *num1 = *num2 = 0;
            pthread_mutex_unlock(&monitor->mutex);
            return;
        }
        pthread_cond_wait(&monitor->cond, &monitor->mutex);
    }
    if (monitor->task_index >= 2) {
        *num1 = monitor->tasks[--monitor->task_index];
        *num2 = monitor->tasks[--monitor->task_index];
        *is_result = 0;
    } else if (monitor->result_index >= 2) {
        *num1 = monitor->results[--monitor->result_index];
        *num2 = monitor->results[--monitor->result_index];
        *is_result = 1;
    } else {
        *num1 = *num2 = 0;
    }
    pthread_mutex_unlock(&monitor->mutex);
}

/**
 * 将结果放入监视器的结果数组中，并唤醒所有等待的线程。
 * @param monitor 监视器指针
 * @param result 要放入结果数组的结果
 */
void monitor_putResult(monitor_t* monitor, int result) {
    pthread_mutex_lock(&monitor->mutex);
    monitor->results[monitor->result_index++] = result;
```

```
    pthread_cond_broadcast(&monitor->cond);
    pthread_mutex_unlock(&monitor->mutex);
}

/**
 * @brief 线程函数，用于执行加法任务并将结果放入共享缓冲区中
 *
 * @param arg 监视器指针
 * @return void*
 */
void* thread_func(void* arg) {
    monitor_t* monitor = (monitor_t*)arg;
    while (1) {
        int num1, num2, is_result;
        monitor_getTask(monitor, &num1, &num2, &is_result);
        if (num1 == 0 && num2 == 0) {
            break;
        }
        usleep((rand() % 10 + 1) * 1000); // 随机时间扰动
        int sum = num1 + num2;
        /**
         * pthread_self is  using the pthread_self function to get the
thread ID of the current thread.
         */
        printf("Thread %ld added %d and %d to get %d. The numbers
were %s.\n", (long)pthread_self(), num1, num2, sum, is_result ?
"results from previous additions" : "original numbers");

        monitor_putResult(monitor, sum);
    }
    return NULL;
}

int main() {
    srand(time(NULL));
    pthread_t threads[NUM_THREADS];
    monitor_t monitor;

    monitor_init(&monitor);

    for (long i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, thread_func, &monitor);
    }
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final result: %d\n", monitor.results[0]);

    monitor_destroy(&monitor);

    return 0;
}
```

（注：以上代码使用了 gpt 添加了注释）

运行结果如下：

```
Thread 139684659074816 added 27 and 97 to get 124. The numbers were original numbers.
Thread 139684709431040 added 51 and 86 to get 137. The numbers were original numbers.
Thread 139684692645632 added 51 and 55 to get 106. The numbers were original numbers.
Thread 139684675860224 added 71 and 42 to get 113. The numbers were original numbers.
Thread 139684701038336 added 9 and 54 to get 63. The numbers were original numbers.
Thread 139684667467520 added 20 and 72 to get 92. The numbers were original numbers.
Thread 139684717823744 added 69 and 68 to get 137. The numbers were original numbers.
Thread 139684675860224 added 113 and 106 to get 219. The numbers were results from previous additions.
Thread 139684684252928 added 83 and 30 to get 113. The numbers were original numbers.
Thread 139684709431040 added 137 and 124 to get 261. The numbers were results from previous additions.
Thread 139684709431040 added 261 and 113 to get 374. The numbers were results from previous additions.
Thread 139684667467520 added 92 and 63 to get 155. The numbers were results from previous additions.
Thread 139684675860224 added 219 and 137 to get 356. The numbers were results from previous additions.
Thread 139684667467520 added 155 and 374 to get 529. The numbers were results from previous additions.
Thread 139684667467520 added 529 and 356 to get 885. The numbers were results from previous additions.
Final result: 885
```

可以看到其产生了正确的规约结果！