作业8

毕定钧 2021K8009906014

本次作业包含:

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:

1) 顾客取号操作 $Customer\_Service$

2) 柜员服务操作 $Teller\_Service$

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 $1 \sim 10ms$。

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。

8.1

代码及其运行结果如下：

```c
void *Customer_Service(void *arg) {
    int customer_number = *((int *)arg);
    printf("Customer %c arrives at the counter\n", 'A' + customer_number);
    sem_wait(&sem_customers[customer_number]); // Customer takes a number
    printf("Customer %c gets a number\n", 'A' + customer_number);
    sem_post(&sem_clerks[0]); // Wake up clerk 1
    sem_post(&sem_clerks[1]); // Wake up clerk 2
    return NULL;
}

void *Teller_Service(void *arg) {
    int clerk_number = *((int *)arg);
    for (int i = 0; i < 10; i++) {
        sem_wait(&sem_clerks[clerk_number]); // Wait for a customer to take a number
        int service_time = rand() % 3 + 1; // Generate a random service time (1~3 seconds)
        printf("Clerk %d starts serving a customer, service time %d seconds\n", clerk_number, service_time);
        sleep(service_time);
        printf("Clerk %d finishes the service\n", clerk_number);
        sem_post(&sem_customers[i]); // Finish the service
    }
    return NULL;
}
```

```
Customer A arrives at the counter
Customer B arrives at the counter
Customer D arrives at the counter
Customer E arrives at the counter
Customer C arrives at the counter
Customer F arrives at the counter
Customer G arrives at the counter
Customer J arrives at the counter
Customer I arrives at the counter
Clerk 0 starts serving a customer, service time 2 seconds
Customer H arrives at the counter
Clerk 1 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Customer A gets a number
Clerk 1 starts serving a customer, service time 1 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Customer B gets a number
Clerk 1 starts serving a customer, service time 3 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Clerk 0 finishes the service
Customer C gets a number
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 starts serving a customer, service time 1 seconds
Clerk 1 finishes the service
Customer D gets a number
Clerk 1 starts serving a customer, service time 1 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Customer E gets a number
Clerk 1 starts serving a customer, service time 3 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Customer F gets a number
Clerk 1 starts serving a customer, service time 3 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 0 finishes the service
Clerk 1 finishes the service
Customer G gets a number
Clerk 0 starts serving a customer, service time 3 seconds
Clerk 1 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
Customer H gets a number
Clerk 1 starts serving a customer, service time 1 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 1 seconds
Clerk 1 finishes the service
Customer I gets a number
Clerk 1 starts serving a customer, service time 2 seconds
Clerk 0 finishes the service
Clerk 0 starts serving a customer, service time 2 seconds
Clerk 1 finishes the service
```

8.2

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

#define THREAD_COUNT 8
#define TASK_COUNT 16

pthread_mutex_t mutex;
pthread_cond_t taskAvailable;
pthread_cond_t resultsAvailable;
int tasks[TASK_COUNT];
int taskIndex = 0;
int results[THREAD_COUNT];
int tasksRemaining = TASK_COUNT;
int tasksRemaining_tmp = TASK_COUNT;

// Function to generate random integers between 1 and 10
int getRandomNumber() {
    return (rand() % 100) + 1;
}

// Function for each thread to perform the addition and put the result
void* threadFunction(void* arg) {
    int threadId = *(int*)arg;
    while (1) {
        pthread_mutex_lock(&mutex);
        while (taskIndex >= tasksRemaining_tmp) {
            if (tasksRemaining <= 0) {
                pthread_mutex_unlock(&mutex);
                pthread_exit(NULL);
            }
            pthread_cond_wait(&taskAvailable, &mutex);
        }
        int task1 = tasks[taskIndex++];
        int task2 = tasks[taskIndex++];
        tasksRemaining -= 2;
        pthread_cond_signal(&taskAvailable);
        pthread_mutex_unlock(&mutex);

        // Simulate random time disturbance
        int disturbance = getRandomNumber();
        usleep(disturbance * 1000); // Sleep in milliseconds

        int sum = task1 + task2;
        printf("Thread %d: Adding %d + %d = %d\n", threadId, task1, task2, sum);
        pthread_mutex_lock(&mutex);
        results[threadId] = sum;
        pthread_cond_signal(&resultsAvailable);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
int main() {
    srand(time(NULL));
    pthread_t threads[15];
    int threadIds[15];
    int remainingTasks = TASK_COUNT;
    int remainingThreads = THREAD_COUNT;
    int threadindex = 0;

    // Initialize tasks with random integers
    for (int i = 0; i < TASK_COUNT; i++) {
        tasks[i] = getRandomNumber();
    }

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&taskAvailable, NULL);
    pthread_cond_init(&resultsAvailable, NULL);

    while (remainingTasks > 1) {
        // Create and start threads
        for (int i = 0; i < remainingThreads; i++) {
            threadIds[i + threadindex] = i;
            pthread_create(&threads[i + threadindex], NULL, threadFunction, &threadIds[i + threadindex]);
        }

        // Wait for all threads to finish
        for (int i = 0; i < remainingThreads; i++) {
            pthread_join(threads[i + threadindex], NULL);
        }

        for (int i = 0; i < remainingTasks; i++) {
            pthread_mutex_lock(&mutex);
            while (results[i] == 0) {
                pthread_cond_wait(&resultsAvailable, &mutex);
            }
            pthread_mutex_unlock(&mutex);
        }

        remainingTasks /= 2;
        threadindex += remainingThreads;
        remainingThreads /= 2;
        printf("Reducing %d results to %d\n", remainingTasks * 2, remainingTasks);
        for (int i = 0; i < remainingTasks; i++) {
            tasks[i] = results[i];
        }
        taskIndex = 0;
        tasksRemaining_tmp /= 2;
    }

    int finalResult = tasks[0];
    printf("Final result: %d\n", finalResult);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&taskAvailable);
    pthread_cond_destroy(&resultsAvailable);

    return 0;
}
```

```
Thread 5: Adding 59 + 28 = 87
Thread 0: Adding 49 + 2 = 51
Thread 1: Adding 88 + 78 = 166
Thread 7: Adding 39 + 17 = 56
Thread 2: Adding 36 + 10 = 46
Thread 4: Adding 56 + 31 = 87
Thread 3: Adding 21 + 11 = 32
Thread 6: Adding 80 + 56 = 136
Reducing 16 results to 8
Thread 1: Adding 46 + 32 = 78
Thread 3: Adding 87 + 87 = 174
Thread 0: Adding 51 + 166 = 217
Thread 2: Adding 136 + 56 = 192
Reducing 8 results to 4
Thread 1: Adding 192 + 174 = 366
Thread 0: Adding 217 + 78 = 295
Reducing 4 results to 2
Thread 0: Adding 295 + 366 = 661
Reducing 2 results to 1
Final result: 661
```

根据题目要求，实现了功能，其中我定义了一个大小为 $8+4+2+1$ 的线程组用来存放各个线程，为了节约空间，将操作数直接存回覆盖原有操作数。其中，定义了 $tasksRemaining$ 并使用互斥锁保护，每次执行加法后将其减 2，直到其为 0 或小于 0 则退出线程，一个线程可以不仅仅使用一次，而是只要空闲就可以使用，这样可以保证程序不仅仅用于处理 16 个数的相加，也可以用于处理更大的数目，当然要想完全实现任意个数相加还需要在宏定义、数组大小等方面再进行优化。