# 作业8

贾城昊

2021K8009929010

**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**

调研：

本次作业代码种会使用**sem\_wait**和**sem\_post**函数**，sem\_wait**和**sem\_post**是信号量的两个主要操作，用于实现同步和互斥。它们通常用于多线程和多进程编程中来控制共享资源的访问。

1. **sem\_wait**（等待）：
   * **sem\_wait**函数用于请求获得一个信号量。如果信号量的计数值大于0，表示资源可用，**sem\_wait**将减少信号量的计数，并立即返回，允许线程或进程访问共享资源。
   * 如果信号量的计数值为0，表示资源不可用，**sem\_wait**将阻塞线程或进程，直到信号量的计数值变为大于0。这使得线程或进程等待其他线程或进程释放资源。
   * 一旦成功获得信号量，**sem\_wait**将继续执行，并允许访问共享资源。如果多个线程或进程同时尝试获得信号量，只有一个会成功，其余将被阻塞。
2. **sem\_post**（释放）：
   * **sem\_post**函数用于释放信号量，增加其计数值。这表示共享资源现在可用于其他线程或进程。
   * 当一个线程或进程完成了对共享资源的使用，应该调用**sem\_post**来释放信号量，以允许其他线程或进程继续访问资源。
   * 如果有线程或进程在等待信号量，它们中的一个将被唤醒，以获得对资源的访问。

所以**sem\_wait**和**sem\_post**分别相当于P操作和V操作

1. C程序如下：

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define NUM\_CUSTOMERS 10 // 顾客数量

#define NUM\_TELLERS 3 // 柜员数量

sem\_t customers; // 顾客信号量，初始值为0

sem\_t tellers; // 柜员信号量，初始值为NUM\_TELLERS

void \*Customer\_Service(void \*customer\_number) {

// 获取顾客号码

int customer\_id = \*((int \*)customer\_number);

// 顾客取号操作

printf("Customer %d takes a number and waits.\n", customer\_id);

sem\_post(&customers); // 通知柜员有新顾客

// 等待柜员服务

sem\_wait(&tellers);

printf("Customer %d is being served.\n", customer\_id);

// 顾客被柜员服务

pthread\_exit(NULL);

}

void \*Teller\_Service(void \*teller\_id) {

// 获取柜员号码

int teller\_number = \*((int \*)teller\_id);

while (1) {

// 等待顾客

sem\_wait(&customers);

// 调用顾客的号码

printf("Teller %d calls the next customer.\n", teller\_number);

// 为顾客服务

//...

// 通知柜员空闲

sem\_post(&tellers);

}

}

int main() {

pthread\_t customers\_threads[NUM\_CUSTOMERS];

pthread\_t tellers\_threads[NUM\_TELLERS];

sem\_init(&customers, 0, 0);

sem\_init(&tellers, 0, NUM\_TELLERS);

int customer\_numbers[NUM\_CUSTOMERS];

int teller\_numbers[NUM\_TELLERS];

// 创建顾客线程

for (int i = 0; i < NUM\_CUSTOMERS; i++) {

customer\_numbers[i] = i;

pthread\_create(&customers\_threads[i], NULL, Customer\_Service, &customer\_numbers[i]);

}

// 创建柜员线程

for (int i = 0; i < NUM\_TELLERS; i++) {

teller\_numbers[i] = i;

pthread\_create(&tellers\_threads[i], NULL, Teller\_Service, &teller\_numbers[i]);

}

// 等待顾客线程和柜员线程完成

for (int i = 0; i < NUM\_CUSTOMERS; i++) {

pthread\_join(customers\_threads[i], NULL);

}

for (int i = 0; i < NUM\_TELLERS; i++) {

pthread\_join(tellers\_threads[i], NULL);

}

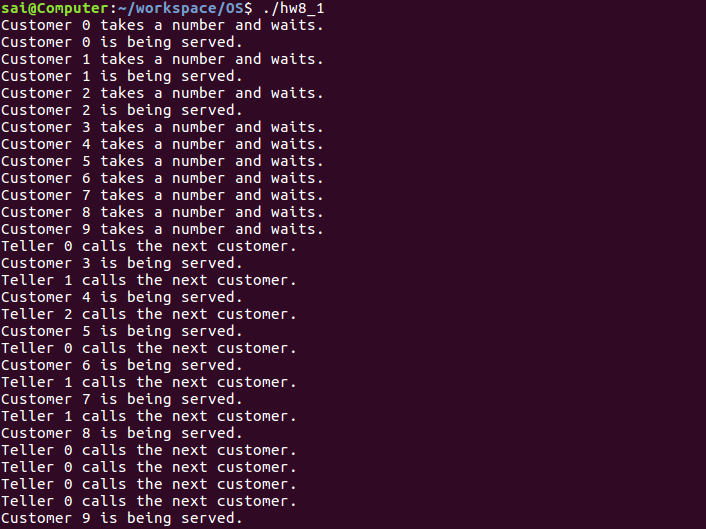
sem\_destroy(&customers);

sem\_destroy(&tellers);

return 0;

}

1. 运行结果：



可以看出，结果是正确的，满足题目要求。

**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。

1. C程序如下：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <time.h>

#define THREADS 8

#define ARRAY\_SIZE 16

// Monitor structure

typedef struct {

pthread\_mutex\_t mutex;

pthread\_cond\_t task\_available;

pthread\_cond\_t result\_available;

int task\_index;

int result\_index;

int array[ARRAY\_SIZE];

int result[ARRAY\_SIZE];

int sum;

} Monitor;

Monitor monitor;

void monitor\_init(Monitor \*m) {

pthread\_mutex\_init(&m->mutex, NULL);

pthread\_cond\_init(&m->task\_available, NULL);

pthread\_cond\_init(&m->result\_available, NULL);

m->task\_index = ARRAY\_SIZE;

m->result\_index = 0;

m->sum = 0;

for (int i = 0; i < ARRAY\_SIZE; i++) {

m->array[i] = rand() % 100; // Initialize the array with random integers

}

}

int get\_random\_delay() {

return (rand() % 10 + 1);

}

void monitor\_getTask(int \*task1, int \*task2) {

pthread\_mutex\_lock(&monitor.mutex);

while (monitor.task\_index + monitor.result\_index < 2) {

if(monitor.task\_index == 0 && monitor.result\_index == 1){

pthread\_mutex\_unlock(&monitor.mutex);

\*task1 = \*task2 = -1;

return;

}

pthread\_cond\_wait(&monitor.task\_available, &monitor.mutex);

}

if(monitor.task\_index >= 2){

\*task1 = monitor.array[monitor.task\_index - 1];

\*task2 = monitor.array[monitor.task\_index - 2];

monitor.task\_index -= 2;

}else if (monitor.result\_index >= 2){

\*task1 = monitor.result[monitor.result\_index - 1];

\*task2 = monitor.result[monitor.result\_index - 2];

monitor.result\_index -= 2;

}else{

\*task1 = monitor.array[monitor.task\_index - 1];

\*task2 = monitor.result[monitor.result\_index - 1];

monitor.task\_index -= 1;

monitor.result\_index -= 1;

}

pthread\_mutex\_unlock(&monitor.mutex);

}

void monitor\_putResult(int local\_sum) {

pthread\_mutex\_lock(&monitor.mutex);

monitor.sum += local\_sum;

monitor.result[monitor.result\_index++] = local\_sum;

pthread\_cond\_broadcast(&monitor.result\_available);

pthread\_mutex\_unlock(&monitor.mutex);

}

void \*thread\_routine(void \*arg) {

long thread\_id = (long)arg;

int local\_sum = 0;

int task1, task2;

while (1) {

monitor\_getTask(&task1, &task2);

if(task1 == -1 && task2 == -1){

break;

}

usleep(get\_random\_delay() \* 1000); // Simulate random delay

local\_sum = task1 + task2;

printf("Thread %ld added %d and %d to get %d\n", thread\_id, task1, task2, local\_sum);

monitor\_putResult(local\_sum);

}

pthread\_exit(NULL);

}

void monitor\_destroy(Monitor monitor) {

pthread\_mutex\_destroy(&monitor.mutex);

pthread\_cond\_destroy(&monitor.task\_available);

pthread\_cond\_destroy(&monitor.result\_available);

}

int main() {

srand(time(NULL));

pthread\_t threads[THREADS];

monitor\_init(&monitor);

for (long i = 0; i < THREADS; i++) {

pthread\_create(&threads[i], NULL, thread\_routine, (void \*)i);

}

for (int i = 0; i < THREADS; i++) {

pthread\_join(threads[i], NULL);

}

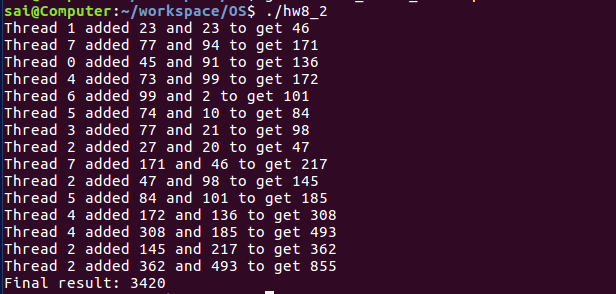
monitor\_destroy(monitor);

printf("Final result: %d\n", monitor.sum);

return 0;

}

1. 运行结果：

****

可以看到其产生了正确的结果