**作业 7**

**唐嘉良**

**2020K8009907032**

1. **设有两个优先级相同的进程 T1，T2 如下。令信号量 S1，S2 的初值为 0，已知 z=2，试 问 T1，T2 并发运⾏结束后 x=? y=? z=?**

**线程 T1 线程 T2**

**y:=1; x:=1;**

**y:=y+2; x:=x+1;**

**V(S1); p(S1);**

**z:=y+1; x:=x+y;**

**P(S2); V(S2);**

**y:=z+y; z:=x+z;**

**注:请分析所有可能的情况，并给出结果与相应执⾏顺序。**

答：T2第三行须在T1第三行之后执行，且T2第五行须在T1第五行之前执行。

根据这一原则，有不同的执行顺序。观察语句，发现可能存在相关的语句只有z:=y+1、z:=x+z、y:=z+y这三条。首先x:=x+y肯定在T1第三行和第五行之间，且它与同样在T1第三行和第五行之间的z:=y+1并无相关。而z:=y+1和y:=z+y的相对位置已经确定，所以只需要讨论z:=x+z的位置。

**1)**如果该语句位于z:=y+1之前，则三条相关语句的顺序（由先到后）为z:=x+z、z:=y+1、y:=z+y，运行结束后(x,y,z)=(5,7,4);

**2)**如果该语句位于z:=y+1之后且位于y:=z+y之前，则三条相关语句的顺序（由先到后）为z:=y+1、z:=x+z、y:=z+y，运行结束后(x,y,z)=(5,12,9);

**3)**如果该语句位于y:=z+y之后，则三条相关语句的顺序（由先到后）为z:=y+1、y:=z+y、z:=x+z，运行结束后(x,y,z)=(5,7,9);

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**

**答**：C语言风格伪代码设计如下：

int counter = n; //信号量：空闲柜员数

int customer = 0; //信号量：要服务的顾客数

int Customer\_Service(){

    P(counter);//取号，如果柜员有空就直接叫号，否则进入等待队列

    call();//叫号，相当于顾客到柜台（进buffer）

    V(customer);

}

int Teller\_Service(){

    P(customer);

    serve();//服务,相当于顾客离开柜台（出buffer）

    V(counter);

}

1. **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。**

**答**：设计代码如下：

---------------------------------------------------------------------

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>

#include <syscall.h>

#include <stdbool.h>

#include <sys/time.h>

typedef struct {

int a, b;

} Pair;

typedef struct Node {

int num;

struct Node \* next;

} Node;

typedef struct {

int cnt;

Node \* head;

Node \* nail;

} Queue;

void Queue\_init(Queue \* que) {

que->cnt = 0;

que->head = NULL;

que->nail = NULL;

}

void push(Queue \* que, int num) {

Node \* new\_node = malloc(sizeof(Node));

new\_node->num = num;

if (que->head == NULL) {

que->head = new\_node;

que->nail = new\_node;

new\_node->next = NULL;

} else {

que->nail->next = new\_node;

que->nail = new\_node;

}

que->cnt += 1;

}

int pop(Queue \* que) {

Node \* old\_node = que->head;

int res = old\_node->num;

if (old\_node->next == NULL) {

que->head = NULL;

que->nail = NULL;

} else {

que->head = old\_node->next;

}

que->cnt -= 1;

free(old\_node);

return res;

}

int Queue\_print(Queue \* que) {

for (Node \* p = que->head; p != NULL; p = p->next) {

printf("%d ", p->num);

}

printf("\n");

}

typedef struct {

Queue que;

pthread\_mutex\_t lock;

pthread\_cond\_t can\_reduce;

int remind\_task;

} Monitor;

Monitor task\_monitor;

void Monitor\_init(Monitor \* p) {

p->remind\_task = 15;//////////////

Queue\_init(&p->que);

for (int i = 0; i < 16; i++) {////////

push(&p->que, rand()%100 );

}

pthread\_mutex\_init(&p->lock, NULL);

}

bool Monitor\_getTask(Monitor \* p, Pair \* result) {

pthread\_mutex\_lock(&p->lock);

if (p->remind\_task <= 0) {

pthread\_mutex\_unlock(&p->lock);

return false;

}//no task to do

p->remind\_task -= 1;

while (p->que.cnt < 2) {

pthread\_cond\_wait(&p->can\_reduce, &p->lock);

}//still have tasks to do, but nums in queue are not enough

//wait until other threads put res into queue

result->a = pop(&p->que);

result->b = pop(&p->que);

printf("get task: %d, %d\n", result->a, result->b);

pthread\_mutex\_unlock(&p->lock);

return true;

}

void Monitor\_putResult(Monitor \* p, int res) {

pthread\_mutex\_lock(&p->lock);

push(&p->que, res);

printf("put result: %d\n", res);

printf("The queue now is ");

Queue\_print(&task\_monitor.que);

pthread\_cond\_signal(&p->can\_reduce);//tell other threads

pthread\_mutex\_unlock(&p->lock);

}

void \* func\_thread\_reduce(void \* arg) {

Pair task;

while (1) {

bool get\_task = Monitor\_getTask(&task\_monitor, &task);

if (!get\_task) {

printf("A thread finished!\n");

return NULL;

}

sleep((rand()%10)/1000); //random time delay: 1~10ms

int res = task.a + task.b;

Monitor\_putResult(&task\_monitor, res);

}

return NULL;

}

int main() {

Monitor\_init(&task\_monitor);

printf("The initial 16 integers are:\n");

Queue\_print(&task\_monitor.que);

printf("Processing...\n");

pthread\_t thread[8];

for (int i = 0; i < 8; i++) {

pthread\_create(thread + i, NULL, func\_thread\_reduce, NULL);

}

for (int i = 0; i < 8; i++) {

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

}

printf("\nPass\nThe final result is: ");

Queue\_print(&task\_monitor.que);

return 0;

}

运行结果如下：（初始化的随机数范围为0~100，每次push两数之和入队之后都会打印一遍当前队列）

