1. **实验描述及实验目的**

多线程是现代操作系统的核心概念，是能够保障操作系统正常运转的基础。Windows、Linux等操作系统向用户提供了多线程、信号量等函数接口，使用户可以通过C/C++、JAVA等主流编程语言来访问调用，以实现用户级线程的创建、运行、管理等功能。本次实验的目标如下：

1. 掌握Windows、Linux系统中多线程、信号量等的相关C函数使用；初步掌握多线程编程。
2. 理解多线程的运行和调度原理；学会设计临界区/信号量来实现多个线程之间的互斥。
3. **实验报告要求**

(1) 实验报告排版请注意字体、字号、段间距、样式等格式的统一。模板中原有的内容保留“楷体\_GB2312g”格式，自行输入的内容可以采用“宋体”+“Times New Roman”小四号字体。可以在模板的基础上自行拓展大标题、小标题等格式，无强制要求，注意美观易辨识即可。

(2) 请按照实验内容题目提供核心代码、注释、必要的说明与实验结果，其余的设计、分析、思考、总结可以自行拓展。请注意完成实验内容题目的全部要求。请注意截图的清晰程度。报告中的代码，请注意其排版，如有 IDE 的高亮请保留高亮。

1. **实验内容题目**

**（1）题目一：**Mary和Sally是亲姐妹，她们有一个共同的银行账户，她们可以分别到ATM机取款；爸爸，妈妈，奶奶，爷爷，舅舅也可以分别到ATM机给银行账户存钱。设账户的初始余额为10元。爸爸，妈妈，奶奶，爷爷，舅舅每次分别存入10，20，30，40，50元，每个人分别存款2次。Mary和Sally每次分别取50和100元，每个人分别取款2次。存款和取款的顺序是随机的。假设Mary和Sally的银行账户是信用卡，即当余额少于取款额时，仍旧能够取款成功。利用临界区(Windows系统)或者mutex信号量(Linux系统)编制程序来模拟上述存取款过程，在主程序结束时将账户的最后余额输出，并人工验证一下是否正确。

**（2）题目二：**本题的要求跟题目一的要求一样，唯一区别是：在本题中假设Mary和Sally的银行账户不是信用卡，即当余额少于取款额时，不能取款，取款线程需要阻塞等待直到账户有足够的钱。

1. **实验过程与代码实现**
2. 假设Mary和Sally的银行账户是信用卡，即当余额少于取款额时，仍旧能够取款成功。利用临界区(Windows系统)或者mutex信号量(Linux系统)编制程序来模拟上述存取款过程，在主程序结束时将账户的最后余额输出，并人工验证一下是否正确。

**实验代码：**

#include <stdlib.h> //包含随机数产生函数

#include <time.h> //与时间有关的函数头文件

#include <windows.h> //针对Windows操作系统

#include <stdio.h>

int bank\_balance = 10;

CRITICAL\_SECTION cs;

typedef struct {

char name[20];

int num;

int thread\_id;

int is\_deposit; // 1表示存钱，0表示取钱

}Transaction;

//存钱线程函数

DWORD WINAPI Deposit(LPVOID lpParam)

{

Transaction\* info = (Transaction\*)lpParam;

// 随机延迟，模拟操作耗时

Sleep(rand() % 100);

EnterCriticalSection(&cs);

bank\_balance += info->num;

printf("thread\_id:%d %s存了%d元，当前余额：%d元\n", info->thread\_id, info->name,info->num,bank\_balance);

LeaveCriticalSection(&cs);

free(info);

return 0;

}

//信用卡取钱线程函数

DWORD WINAPI Withdraw(LPVOID lpParam)

{

Transaction\* info = (Transaction\*)lpParam;

int output = info->num;

// 随机延迟，模拟操作耗时

Sleep(rand() % 100);

EnterCriticalSection(&cs);

bank\_balance -= output;

printf("thread\_id:%d %s取了%d元，当前余额：%d元\n", info->thread\_id, info->name, output, bank\_balance);

LeaveCriticalSection(&cs);

free(info);

return 0;

}

int main()

{

printf("初始账户余额：%d 元\n", bank\_balance);

srand((unsigned int)time(NULL));

InitializeCriticalSection(&cs);

HANDLE threads[14]; // 14个线程

int thread\_idx = 0;

Transaction all\_transactions[14];

struct {

char name[20];

int num;

}depositors[] = {

{"爸爸",10},{"妈妈",20},{"奶奶",30},{"爷爷",40},{"舅舅",50}

};

int dep\_num = sizeof(depositors) / sizeof(depositors[0]);

struct {

char name[20];

int num;

}withdrawers[] = {

{"Mary",50},{"Sally",100}

};

int with\_num = sizeof(withdrawers) / sizeof(withdrawers[0]);

// 存款交易

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 2; j++) {

strcpy\_s(all\_transactions[thread\_idx].name, sizeof(all\_transactions[thread\_idx].name), depositors[i].name);

all\_transactions[thread\_idx].num = depositors[i].num;

all\_transactions[thread\_idx].is\_deposit = 1;

thread\_idx++;

}

}

// 取款交易

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

strcpy\_s(all\_transactions[thread\_idx].name, sizeof(all\_transactions[thread\_idx].name), withdrawers[i].name);

all\_transactions[thread\_idx].num = withdrawers[i].num;

all\_transactions[thread\_idx].is\_deposit = 0;

thread\_idx++;

}

}

// 随机打乱交易顺序

for (int i = thread\_idx - 1; i > 0; i--)

{

int j = rand() % (i + 1);

Transaction temp = all\_transactions[i];

all\_transactions[i] = all\_transactions[j];

all\_transactions[j] = temp;

}

// 按随机顺序创建线程

for (int i = 0; i < 14; i++)

{

Transaction\* trans = (Transaction\*)malloc(sizeof(Transaction));

memcpy(trans, &all\_transactions[i], sizeof(Transaction));

trans->thread\_id = i + 1;

if (trans->is\_deposit) {

threads[i] = CreateThread(NULL, 0, Deposit, trans, 0, NULL);

}

else {

threads[i] = CreateThread(NULL, 0, Withdraw, trans, 0, NULL);

}

if (threads[i] == NULL) {

printf("创建线程失败！\n");

free(trans);

}

}

// 等待所有线程完成

WaitForMultipleObjects(14, threads, TRUE, INFINITE);

// 清理资源

for (int i = 0; i < 14; i++) {

if (threads[i] != NULL) {

CloseHandle(threads[i]);

}

}

DeleteCriticalSection(&cs);

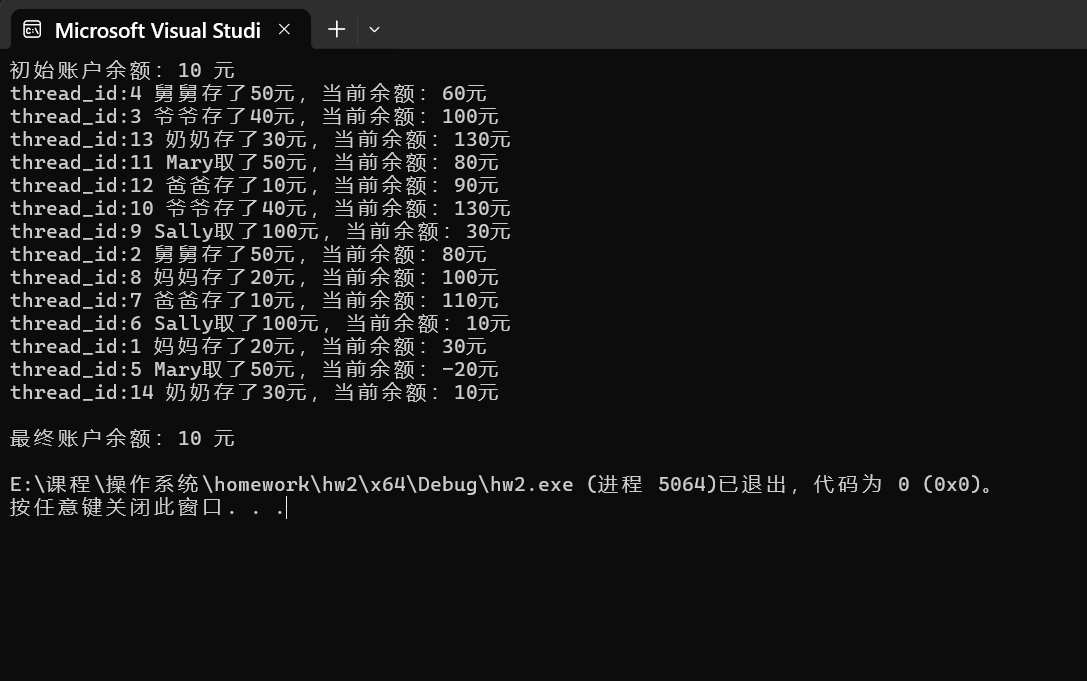
// 输出最终余额

printf("\n最终账户余额：%d 元\n", bank\_balance);

return 0;

}

**实验结果：**



**（2）**在本题中假设Mary和Sally的银行账户不是信用卡，即当余额少于取款额时，不能取款，取款线程需要阻塞等待直到账户有足够的钱。

**实验代码：**

#include <stdlib.h> //包含随机数产生函数

#include <time.h> //与时间有关的函数头文件

#include <windows.h> //针对Windows操作系统

#include <stdio.h>

int bank\_balance = 10;

CRITICAL\_SECTION cs;

HANDLE can\_withdraw = CreateSemaphore(NULL, 0, LONG\_MAX, NULL); // 信号量，初始值为0

typedef struct {

char name[20];

int num;

int thread\_id;

int is\_deposit; // 1表示存钱，0表示取钱

}Transaction;

//存钱线程函数

DWORD WINAPI Deposit(LPVOID lpParam)

{

Transaction\* info = (Transaction\*)lpParam;

// 随机延迟，模拟操作耗时

Sleep(rand() % 100);

EnterCriticalSection(&cs);

bank\_balance += info->num;

printf("thread\_id:%d %s存了%d元，当前余额：%d元\n", info->thread\_id, info->name,info->num,bank\_balance);

ReleaseSemaphore(can\_withdraw, 1, NULL);// 存钱后发送信号量

LeaveCriticalSection(&cs);

free(info);

return 0;

}

DWORD WINAPI Withdraw(LPVOID lpParam)

{

Transaction\* info = (Transaction\*)lpParam;

Sleep(rand() % 100);

int can\_proceed = 0;

while (!can\_proceed)

{

EnterCriticalSection(&cs);

if (bank\_balance >= info->num)

{

// 余额足够，执行取款

bank\_balance -= info->num;

printf("thread\_id:%d %s取出%d元，当前余额：%d元\n",

info->thread\_id, info->name, info->num, bank\_balance);

can\_proceed = 1;

LeaveCriticalSection(&cs);

}

else

{

// 余额不足，等待信号量"

printf("thread\_id:%d %s想取%d元，当前余额：%d元,余额不足，等待中\n",

info->thread\_id, info->name, info->num, bank\_balance);

LeaveCriticalSection(&cs);

WaitForSingleObject(can\_withdraw, INFINITE);

}

}

free(info);

return 0;

}

int main()

{

printf("初始账户余额：%d 元\n", bank\_balance);

srand((unsigned int)time(NULL));

InitializeCriticalSection(&cs);

HANDLE threads[14]; // 14个线程

int thread\_idx = 0;

Transaction all\_transactions[14];

struct {

char name[20];

int num;

}depositors[] = {

{"爸爸",10},{"妈妈",20},{"奶奶",30},{"爷爷",40},{"舅舅",50}

};

int dep\_num = sizeof(depositors) / sizeof(depositors[0]);

struct {

char name[20];

int num;

}withdrawers[] = {

{"Mary",50},{"Sally",100}

};

int with\_num = sizeof(withdrawers) / sizeof(withdrawers[0]);

// 存款交易

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 2; j++) {

strcpy\_s(all\_transactions[thread\_idx].name, sizeof(all\_transactions[thread\_idx].name), depositors[i].name);

all\_transactions[thread\_idx].num = depositors[i].num;

all\_transactions[thread\_idx].is\_deposit = 1;

thread\_idx++;

}

}

// 取款交易

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

strcpy\_s(all\_transactions[thread\_idx].name, sizeof(all\_transactions[thread\_idx].name), withdrawers[i].name);

all\_transactions[thread\_idx].num = withdrawers[i].num;

all\_transactions[thread\_idx].is\_deposit = 0;

thread\_idx++;

}

}

// 随机打乱交易顺序

for (int i = thread\_idx - 1; i > 0; i--)

{

int j = rand() % (i + 1);

Transaction temp = all\_transactions[i];

all\_transactions[i] = all\_transactions[j];

all\_transactions[j] = temp;

}

// 按随机顺序创建线程

for (int i = 0; i < 14; i++)

{

Transaction\* trans = (Transaction\*)malloc(sizeof(Transaction));

memcpy(trans, &all\_transactions[i], sizeof(Transaction));

trans->thread\_id = i + 1;

if (trans->is\_deposit) {

threads[i] = CreateThread(NULL, 0, Deposit, trans, 0, NULL);

}

else {

threads[i] = CreateThread(NULL, 0, Withdraw, trans, 0, NULL);

}

if (threads[i] == NULL) {

printf("创建线程失败！\n");

free(trans);

}

}

// 等待所有线程完成

WaitForMultipleObjects(14, threads, TRUE, INFINITE);

// 清理资源

for (int i = 0; i < 14; i++) {

if (threads[i] != NULL) {

CloseHandle(threads[i]);

}

}

DeleteCriticalSection(&cs);

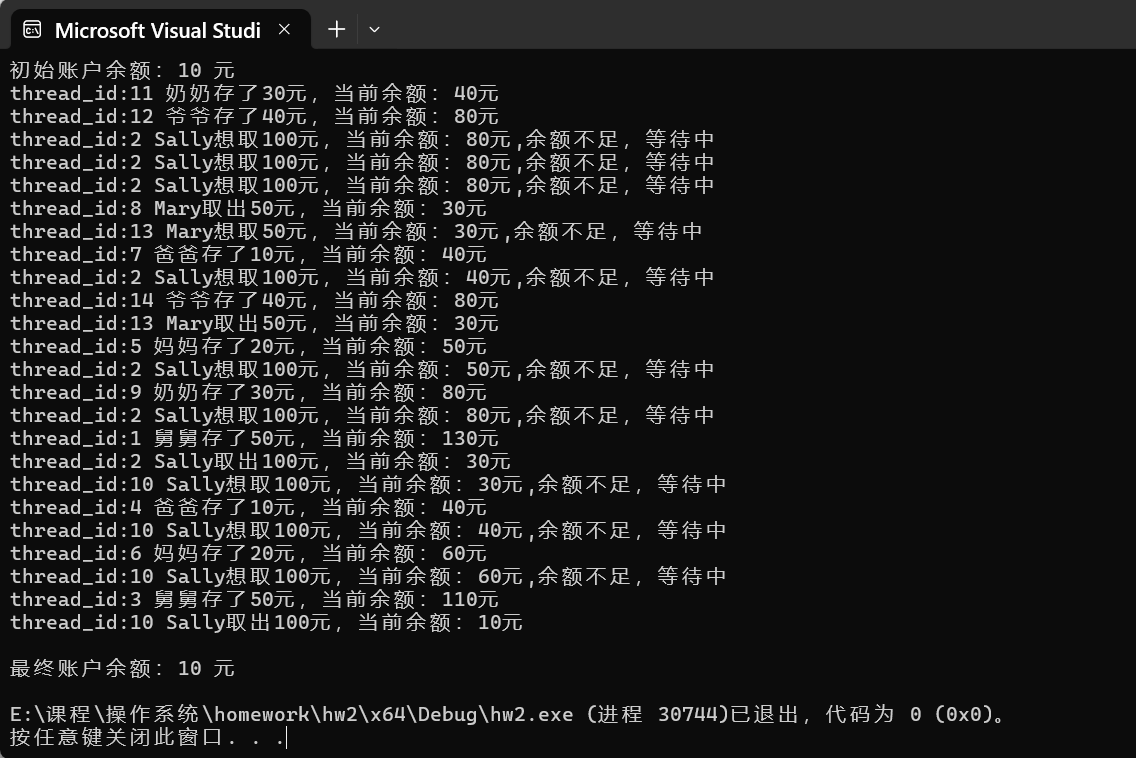
// 输出最终余额

printf("\n最终账户余额：%d 元\n", bank\_balance);

return 0;

}

**实验结果：**



1. **实验总结**

第二题相比于第一题，增加了余额应当大于等于扣款的限制，如此以来，就要增加在取款余额不足时的等待机制。我们借助信号量来实现：余额不够的情况下，信号量经过一次循环后置零，取款线程拥塞直至接收到存款线程发出的置一信号量才重新判断。