北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **操作系统并发特征验证性实验**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年10月8日**

目录

1. 开发运行环境和工具

操作系统	Linux内核版本	处理器	GCC版本
Deepin 20.9	5.18.17-amd64-desktop-hwe	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (8核，非大小核)	gcc (Uos 8.3.0.3-3+rebuild) 8.3.0

- **其他工具**:
    - **strace**：version 4.26
    - **编辑**：VSCode (version 1.82.2)

- **编译和运行**：Terminal

2. 实验过程

2.1. 阅读算法

算法逻辑写在以下的代码注释中。

头文件。三份源码都相同。

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// 引用线程库、标准输入输出库、标准库和时间库
```

宏定义操作、定义循环变量和共享全局变量。三份源码都相同。

```
typedef char _bool;
#define _true (char)1
#define _false (char)0
#define LOG printf("author: 柯劲帆\n")
// 宏定义一些操作
const int LOOP_NUM = 10000000;
// 定义循环次数
volatile int nAccount1, nAccount2, turn;
// 创建三个变量用于模拟两个账户和线程并发执行的控制，是临界资源
// volatile关键字表示这些变量有可能在本程序以外被修改，每次访问这些变量都需要到内存中读取。
```

主程序。三份源码都相同。

```
int main(int argc, char* argv[]) {
    clock_t start, end;
    start = clock(); // 记录程序开始时间
    pthread_t th1, th2;
    pthread_create(&th1, NULL, transA, NULL);
    pthread_create(&th2, NULL, transB, NULL);
    // 创建两个线程，分别执行transA和transB
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    // 等待两个线程完成
    end = clock(); // 获取程序结束时间
    printf("program running time: %lf\n", ((double)(end - start)) / CLOCKS_PER_SEC);
    // 打印程序运行时间，这个时间包括了：1.两个线程实际运行的时间之和；2.被其他CPU任务阻塞的时间
    LOG;
    // 输出结束信息
}
```

none_sync.cpp的tranA / transB函数。

```
void* transA(void* arg) {
    int nLoop = 0; // 定义循环变量
    int nTemp1, nTemp2, nRandom; // 定义过程记录变量和随机变量
    do {
        nRandom = rand() % 10000; // 为随机变量赋值
        // === 临界区开始 ===
        nTemp1 = nAccount1;
        nTemp2 = nAccount2;
        nAccount1 = nTemp1 + nRandom;
        nAccount2 = nTemp2 - nRandom;
        // 对两个账户进行转账操作
        // === 临界区结束 ===
        nLoop += 1;
        if (nAccount1 + nAccount2 != 0) {
            printf("error\n");
            return NULL;
        } // 如果两个账户之和不为0，即程序出错，退出程序
    } while (nLoop <= LOOP_NUM);
    return arg;
}
```

linux_sync.cpp的tranA / transB函数。

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// 创建一个互斥锁，这个锁将用于保护共享资源的访问，以防止多线程竞争条件
void* transA(void* arg)
{
    int nLoop = 0;
    int nTemp1, nTemp2, nRandom;
    do {
        pthread_mutex_lock(&mutex);
        // 锁定互斥锁以保护临界区
        nRandom = rand() % 10000;
        // === 临界区开始 ===
        nTemp1 = nAccount1;
        nTemp2 = nAccount2;
        nAccount1 = nTemp1 + nRandom;
        nAccount2 = nTemp2 - nRandom;
        // === 临界区结束 ===
        nLoop += 1;
        if (nAccount1 + nAccount2 != 0) {
            printf("error\n");
            return NULL;
        } // 如果两个账户之和不为0，即程序出错，退出程序
        pthread_mutex_unlock(&mutex);
        // 解锁互斥锁
    } while (nLoop <= LOOP_NUM);
    return arg;
}
```

peterson.cpp的tranA / transB函数。

```
volatile _bool flag[2];
// 设置旗标变量
```

```cpp
void* transA(void* arg)
{
    int nLoop = 0;
    int nTemp1, nTemp2, nRandom;
    do {
        flag[0] = _true; // 标志本线程已准备好进入临界区
        turn = 1; // 将turn设置为另一个线程的标号
        while (turn == 1 && flag[1] == _true);
        // 进入临界区前，检查另一个线程的flag是否为1，且turn不等于自己的标号
        // 如果另一个线程也正在等待，且轮到它，则循环等待
        nRandom = rand() % 10000;
        // === 临界区开始 ===
        nTemp1 = nAccount1;
        nTemp2 = nAccount2;
        nAccount1 = nTemp1 + nRandom;
        nAccount2 = nTemp2 - nRandom;
        // === 临界区结束 ===
        nLoop += 1;
        if (nAccount1 + nAccount2 != 0) {
            printf("error\n");
            return NULL;
        }
        flag[0] = _false;
        // 当线程执行完临界区代码后，将自己的flag置为0，表示已经离开了临界区
    } while (nLoop <= LOOP_NUM);
    return arg;
}
```

2.2. 编译运行

在命令行中分别编译3份代码。命令行提示信息、命令如下。均无输出。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./none_sync.cpp -o ./none_sync -pthread
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./linux_sync.cpp -o ./linux_sync -pthread
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./peterson.cpp -o ./peterson -pthread
```

运行第1份程序，命令行提示信息、命令和输出如下。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ ./none_sync
error
error
program running time: 0.000212
author: 柯劲帆
```

运行第2份程序，命令行提示信息、命令和输出如下。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ ./linux_sync
program running time: 4.596819
author: 柯劲帆
```

运行第3份程序，命令行提示信息、命令和输出如下。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ ./peterson
program running time: 3.736581
author: 柯劲帆
```

2.3. 修改并运行none_sync.cpp代码

修改代码，打印各个变量以观察变量的修改状况。

```
do {
    nRandom = rand() % 10000;
    nTemp1 = nAccount1;
    printf("[transA:%d] nTemp1 = %d\n", nLoop, nTemp1);
    nTemp2 = nAccount2;
    printf("[transA:%d] nTemp2 = %d\n", nLoop, nTemp2);
    nAccount1 = nTemp1 + nRandom;
    printf("[transA:%d] nAccount1 = %d + %d = %d\n", nLoop, nTemp1, nRandom, nTemp1 + nRandom);
    nAccount2 = nTemp2 - nRandom;
    printf("[transA:%d] nAccount2 = %d - %d = %d\n", nLoop, nTemp2, nRandom, nTemp2 - nRandom);
    nLoop += 1;
    if (nAccount1 + nAccount2 != 0) {
        printf("error\n");
        printf("[transA] nTemp1=%d, nTemp2=%d, nRandom=%d, nTemp1+nRandom=%d, nTemp2-nRandom=%d, nAccount1=%d, nAccount2=%d\n", nTemp1, nTemp2, nRandom, nTemp1 + nRandom, nTemp2 - nRandom, nAccount1, nAccount2);
        return NULL;
    }
} while (nLoop <= LOOP_NUM);
```

编译并运行。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./none_sync.cpp -o ./none_sync -pthread && ./none_sync
```

截取error部分和main部分的输出如下：

```
error
[transA] nTemp1=41171, nTemp2=-21, nRandom=59, nTemp1+nRandom=41230, nTemp2-nRandom=-80, nAccount1=41230, nAccount2=-80
error
[transB] nTemp1=16, nTemp2=-16, nRandom=5, nTemp1+nRandom=21, nTemp2-nRandom=-21, nAccount1=41230, nAccount2=-80
program running time: 0.000237
author: 柯劲帆
```

完整输出见附录"7.1. none_sync_1程序的完整输出"。

2.4. 修改并运行linux_sync.cpp和peterson.cpp代码

将linux_sync.cpp代码的do-while循环部分修改如下。

```
do {
    pthread_mutex_lock(&mutex);
    nRandom = rand() % 10;
    nTemp1 = nAccount1;
    nTemp2 = nAccount2;
    nAccount1 = nTemp1 + nRandom;
    nAccount2 = nTemp2 - nRandom;
    pthread_mutex_unlock(&mutex);
} while ((nAccount1 + nAccount2) == 0);
```

编译并运行。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./linux_sync_1.cpp -o ./none_sync_1 -pthread && ./linux_sync_1
```

程序一直运行不停止，直至在命令行输入`ctrl+C`才退出。

将peterson.cpp代码的transA函数do-while循环部分修改如下。

```
do {
    flag[0] = _true;
    turn = 1;
    while (turn == 1 && flag[1] == _true);
    nRandom = rand() % 10000;
    nTemp1 = nAccount1;
    nTemp2 = nAccount2;
    nAccount1 = nTemp1 + nRandom;
    nAccount2 = nTemp2 - nRandom;
    flag[0] = _false;
} while ((nAccount1 + nAccount2) == 0);
```

transB函数作类似更改，不同之处在于上述代码第3行`turn = 0;`和第4行`while (turn == 0 && flag[0] == _true);`。

编译并运行。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./peterson_1.cpp -o ./peterson_1 -pthread && ./peterson_1
```

程序一直运行不停止，直至在命令行输入`ctrl+C`才退出。

2.5. 分别多次运行linux_sync程序和peterson程序比较运行时间

多次在命令行中运行linux_sync程序和peterson程序，将输出的运行时间记录。

所有的运行结果均没有error提示信息打印出来，说明以下的运行时间均为do-while循环内临界区运行了一共2×10000000次的时间。

程序	第1次运行用时/s	第2次运行用时/s	第3次运行用时/s	第4次运行用时/s	第5次运行用时/s	第6次运行用时/s	平均运行用时/s
linux_sync	5.829870	3.351867	4.729514	3.930167	5.229939	4.624409	**4.615961**
peterson	3.757013	3.428822	3.420579	2.483765	5.445688	4.487762	**3.837272**

2.6. 使用strace工具观察linux_sync程序和peterson程序的系统调用情况

在终端中调用strace工具分析linux_sync程序和peterson程序的系统调用情况。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ strace -c ./linux_sync
program running time: 4.172697
author: 柯劲帆
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99    1.971464     1971464         1           futex
  0.00    0.000060           2        29           mmap
  0.00    0.000030           2        12           mprotect
  0.00    0.000022          11         2           clone
  0.00    0.000013           2         6           openat
  0.00    0.000008           8         1           munmap
  0.00    0.000006           1         5           read
  0.00    0.000006           1         6           close
  0.00    0.000006           0         7           fstat
  0.00    0.000005           2         2           write
  0.00    0.000004           2         2           clock_gettime
  0.00    0.000003           1         3           brk
  0.00    0.000003           3         1         1 access
  0.00    0.000002           1         2           rt_sigaction
  0.00    0.000002           2         1           execve
  0.00    0.000001           1         1           rt_sigprocmask
  0.00    0.000001           1         1           arch_prctl
  0.00    0.000001           1         1           set_tid_address
  0.00    0.000001           1         1           set_robust_list
  0.00    0.000001           1         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
100.00    1.971639                    85         1 total
```

```
(base) kejingfan@KJF-Huawei-PC:~/code$ strace -c ./peterson
program running time: 4.891933
author: 柯劲帆
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         5           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         6           close
  0.00    0.000000           0         7           fstat
  0.00    0.000000           0        29           mmap
  0.00    0.000000           0        12           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           rt_sigaction
```

```
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         2           clone
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         2           clock_gettime
  0.00    0.000000           0         6           openat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    85         1 total
```

使用taskset命令，将linux_sync程序的线程绑定到7号CPU上运行，并使用strace工具进行监视系统调用情况。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ strace -c taskset -c 7 ./linux_sync
program running time: 0.451617
author: 柯劲帆
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 81.46    0.002184        1092         2           futex
  9.85    0.000264         132         2           clone
  3.25    0.000087           2        37           mmap
  1.49    0.000040           2        16           mprotect
  0.82    0.000022           2         9           openat
  0.63    0.000017          17         1           sched_setaffinity
  0.48    0.000013           6         2           munmap
  0.34    0.000009           1         6           read
  0.34    0.000009           0        10           fstat
  0.30    0.000008           0         9           close
  0.30    0.000008           1         6           brk
  0.22    0.000006           3         2         2 access
  0.11    0.000003           1         2           execve
  0.07    0.000002           1         2           rt_sigaction
  0.07    0.000002           1         2           arch_prctl
  0.07    0.000002           2         1           sched_getaffinity
  0.04    0.000001           1         1           rt_sigprocmask
  0.04    0.000001           1         1           set_tid_address
  0.04    0.000001           0         2           clock_gettime
  0.04    0.000001           1         1           set_robust_list
  0.04    0.000001           1         1           prlimit64
  0.00    0.000000           0         2           write
------ ----------- ----------- --------- --------- ----------------
100.00    0.002681                   117         2 total
```

发现linux_sync程序运行速度非常快，用时接近无绑定CPU运行时间的10%。比较strace显示的系统调用用时，发现futex用时减少显著。

说明在futex调度多个CPU的过程中产生了大量的时间开销。

同理使用taskset命令，将peterson程序的线程绑定到7号CPU上运行，并使用strace工具进行监视系统调用情况。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ strace -c taskset -c 7 ./peterson
^Cstrace: Process 26667 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         6           read
  0.00    0.000000           0         9           close
  0.00    0.000000           0         9           fstat
  0.00    0.000000           0        37           mmap
  0.00    0.000000           0        16           mprotect
  0.00    0.000000           0         2           munmap
  0.00    0.000000           0         6           brk
  0.00    0.000000           0         2           rt_sigaction
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         2           clone
  0.00    0.000000           0         2           execve
  0.00    0.000000           0         2           arch_prctl
  0.00    0.000000           0         1         1 futex
  0.00    0.000000           0         1           sched_setaffinity
  0.00    0.000000           0         1           sched_getaffinity
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           clock_gettime
  0.00    0.000000           0         9           openat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                   112         3 total
```

发现在单颗CPU上运行的peterson程序运行非常缓慢；futex只被调用1次且用时忽略不计。

将该程序提前结束。

减小`LOOP_NUM`的大小，观察导致单核中peterson程序运行缓慢的原因。

在源代码中修改`LOOP_NUM`的大小。

```
// const int LOOP_NUM = 10000000;
const int LOOP_NUM = 1000;
```

编译并运行。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./peterson.cpp -o ./peterson -pthread && taskset -c 7 ./peterson
program running time: 0.000046
author: 柯劲帆
```

程序正常地快速运行完毕了。

不断调整循环次数`LOOP_NUM`的大小，发现当`LOOP_NUM`为10^3的数量级或以下时，程序正常快速运行并结束；当`LOOP_NUM`为10^5的数量级以上时，程序运行极为缓慢；`LOOP_NUM`在10^3和10^5的数量级之间时，程序有时快速运行并结束，有时运行非常慢，且受到是否使用strace工具监视的影响。

上述实验结果记录如下：

LOOP_NUM	10^2	10^3	10^4	10^5	10^6	10^7
直接运行程序用时	0.000027秒	0.000032秒	0.000081秒	0.001459秒	超过1分钟	超过1分钟
使用strace工具监视程序用时	0.000037秒	0.000051秒	0.000040秒	超过1分钟	超过1分钟	超过1分钟

可以看出，循环次数的数量级的增加导致了运行时间的大量非线性增加，由此猜测是线程运行时间的长短导致了操作系统采用了不同的并发执行策略，导致了总的程序执行时间产生了非线性变化。

因此我又做了如下实验：

修改代码，在peterson程序中每个临界区结束时，打印输出线程编号和`nLoop`值。分别设置`LOOP_NUM`为10^3和10^7，编译和运行peterson程序。（限于篇幅，此处省略程序输出的展示，仅用文字说明结果）

观察发现，当`LOOP_NUM`为10^3时，两个线程并不是交错执行的，而是当transA线程循环结束时，transB线程才开始循环，相当于串行执行；但是当`LOOP_NUM`为10^7时，首先transA线程循环了1329次后，transB线程才开始循环，接下来两个线程交错执行，如下是截取的部分输出：

（每行两个数。第一个数为1表示该行由transA输出，第一个数为2表示该行由transB输出；第二个数表示该行是在对应线程的第几个循环输出的。）

```
1 1326
1 1327
1 1328
1 1329
2 1
1 1330
2 2
1 1331
2 3
1 1332
2 4
1 1333
```

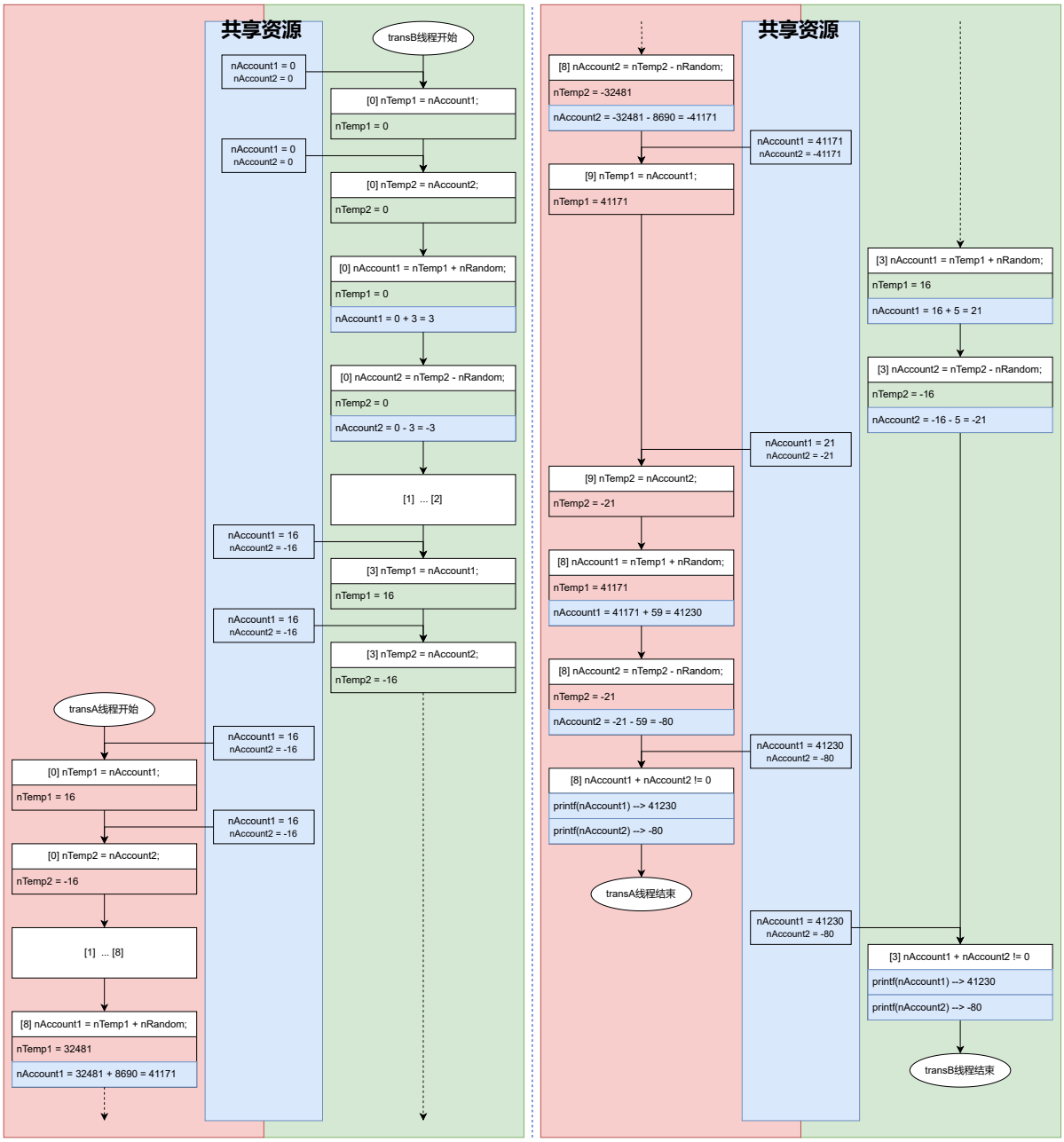
说明影响单颗CPU上peterson程序运行时间大幅度变化的原因是：

- 单个线程运行时间较短时（如`LOOP_NUM`为10^3），操作系统倾向于让线程串行计算，减少了切换开销；
- 当单个线程独占CPU的运行时间达到一定长度时，操作系统会将另一个线程插入，让两个线程并发运行。因此当线程运行时间较长时（如`LOOP_NUM`为10^7），在第一个线程运行一段时间后，另一个线程也开始运行，两个线程开始频繁争抢CPU时间片，切换开销变大，导致运行速度急剧下降；
- strace工具会增加计算开销，影响操作系统的决策。在`LOOP_NUM`为10^4到10^5的数量级时，当该CPU上的运行任务较多，串行运行的线程可能提前开始与其他线程并行运行，导致运行速度急剧下降。

3. 实验分析

3.1. none_sync算法错误原因分析

分析修改了输出信息的none_sync_1.cpp编译的程序，作出时序执行图：



出错的原因是：

在无同步控制的情况下,当两个线程同时执行转账操作时,会出现数据竞争的问题。

具体来说，transA线程读取了账户余额，此时transB线程也读取了相同的账户余额，然后两线程基于读取的余额做计算，最后写入结果。这会导致计算错误，总额不再相等。

3.2. 后两种算法正确的原因分析

3.2.1. Linux操作系统同步机制正确的原因

语句`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`创建的互斥锁保护了共享资源的访问，防止多线程竞争。

当transA线程即将进入临界区时，`pthread_mutex_lock(&mutex);`语句调用系统接口锁定互斥锁，保护临界区，直至临界区执行完毕。在这个过程中，transB线程都无法进入临界区，从而保护了共享资源nAccount1和nAccount2变量在transA线程运行的过程中不会被transB修改。同理，transB线程锁定互斥锁后，transA无法进入临界区。

3.2.2. Peterson算法正确的原因

将**进入临界区前**的并行执行情况枚举如下（假设当前没有线程执行临界区，flag都为false）：

步骤1	步骤2	步骤3	步骤4	步骤5	步骤6	步骤7	进入
A flag[0] = _true	A turn = 1	A while() --> false					transA
A flag[0] = _true	A turn = 1	B flag[1] = _true	B turn = 0	B while() -> true	A while() -> false		transA
A flag[0] = _true	B flag[1] = _true	B turn = 0	B while() -> true	A turn = 1	A while() -> true	B while() -> false	transB
A flag[0] = _true	B flag[1] = _true	A turn = 1	A while() -> true	B turn = 0	B while() -> true	A while() -> false	transA
A flag[0] = _true	B turn = 0	B while() -> true	A turn = 1	A while() -> true	B while() -> false		transB
A flag[0] = _true	B while() -> false						transB

以上是transA先执行第1行do-while代码的大部分情况枚举（由对称性，transB先执行第1行do-while代码的情况同理）。

当任意一个线程（如transA）**在临界区内执行时**，该线程（如transA）对应的flag都为true，另外一个线程（如transB）在判断是否要等待的while语句前，turn都会被赋值为正在执行临界区的线程（如transA）的编号。因此未进入临界区的线程（如transB）会陷入while循环等待。

显然，peterson算法的逻辑可以使得在同一时刻执行临界区的线程数不超过1个，且不会发生所有线程都在while等待的情况。

3.3. Peterson 算法或Linux操作系统同步机制出现问题的原因分析

Peterson 算法或Linux操作系统同步机制的代码不会导致$\mathrm{nAccount1} + \mathrm{nAccount2} \neq 0$，因此不加循环变量的判别不会跳出循环，导致程序死循环。

因此将循环设置为`nLoop < 1000000`，并用`if (nAccount1 + nAccount2 != 0) break;`语句实现原循环条件中的判别功能。

3.4. 两种同步机制的效率比较

从**运行机制**上来看，Linux操作系统同步机制比Peterson算法相对效率更高。

- Peterson算法通过旗标变量和轮转变量实现互斥，需要线程自己判断是否获得访问权限，需要更多的用户态运算来实现同步。
- 操作系统同步机制通过内核级锁实现互斥，由操作系统直接管理同步访问，线程直接获取和释放访问权限，减少了用户态和内核态之间的切换，也不需要线程自己实现同步逻辑，效率更高。

但是从实验中的**实际运行**时间上看，Peterson算法相对Linux操作系统同步机制效率更高。

下面通过分析实验过程的现象比较两种同步机制的效率：

1. 在使用strace工具观察linux_sync程序和peterson程序的系统调用情况的过程中，分析发现耗时最多的系统调用是futex。linux_sync程序调用了两次futex，总共用时较多，而peterson程序只调用了一次futex，用时忽略不计。

   查阅资料[1]可知，futex用于等待给定地址的值发生更改，并为唤醒在特定地址上等待的任何线程提供了一个方法，用于实现共享内存中锁的争用情况。

   在linux_sync程序中，第二个线程执行前访问了futex变量，检测到第一个线程被锁定了，因此进入等待状态，并在第一个线程完成后被唤醒。这样的机制使得linux_sync程序实现内核级锁实现互斥，减少用户态和内核态之间的切换，效率更高。而peterson程序只调用了一次futex，在该项系统调用内并行完成了两个线程。

2. 在将线程绑定到单个CPU上运行的实验中，发现linux_sync程序运行速度非常快，用时接近无绑定CPU运行时间的10%；但是peterson程序运行非常缓慢。
   - linux_sync程序：在多核并行运行时，linux_sync程序需要通过系统调用futex影响CPU的调度，这个调度过程消耗了大量的时间，但是当绑定单个CPU之后这个调度过程被简化了，因此运行时间大大减少。
   - peterson程序：当循环次数到10^3量级以上时，操作系统会让两个线程交替执行，线程切换开销较大，导致绑定单个CPU之后，peterson程序运行非常缓慢；但是不绑定CPU，即使有CPU调度的开销，多核并发的贡献能抵消调度开销，大大减少peterson程序运行的时间。

3. 直接在命令行运行linux_sync程序和peterson程序，多次运行，通过比较运行时间反映两种算法或机制的运行效率。从实际运行时间上看，peterson算法相对linux操作系统同步机制效率更高。

根据以上现象和资料，总结出以下的原因导致Peterson算法程序的运行时间比linux_sync程序的运行时间短：

- peterson程序中没有使用系统调用，核心运算也非常简单，所以系统调度更具灵活性，有助于多核并发，弥补了CPU调度开销和线程切换开销；
- linux_sync程序进行了系统调用，存在大量CPU的调度时间开销。

5. 问题与讨论

5.1. 编译问题

使用g++编译3份代码时，需要加`-pthread`参数，如下。

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./none_sync.cpp -o ./none_sync -pthread
```

否则，

```
(base) kejingfan@KJF-Huawei-PC:~/code$ g++ ./none_sync.cpp -o ./none_sync
/usr/bin/ld: /tmp/ccrfomkX.o: in function `main':
none_sync.cpp:(.text+0x18e): undefined reference to `pthread_create'
/usr/bin/ld: none_sync.cpp:(.text+0x1a9): undefined reference to
`pthread_create'
/usr/bin/ld: none_sync.cpp:(.text+0x1ba): undefined reference to
`pthread_join'
/usr/bin/ld: none_sync.cpp:(.text+0x1cb): undefined reference to
`pthread_join'
collect2: error: ld returned 1 exit status
```

这可能是由于操作系统所带的g++ (Uos 8.3.0.3-3+rebuild) 8.3.0版本特性导致的。我在g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0版本上并未发现不加`-pthread`参数会报错。

`-pthread` 参数是告诉g++编译器链接POSIX线程库的选项。POSIX线程库是一种用于多线程编程的标准库，也被称为pthreads（POSIX Threads）。它提供了创建、管理和同步线程的函数和数据结构[2]。当在代码中使用多线程功能，如 `pthread_create` 和 `pthread_join` 时，需要链接 pthreads 库，以便编译器能够找到这些函数的定义并正确连接它们到可执行文件中。如果不加 `-pthread` 参数，编译器可能会产生错误消息，因为它无法找到对应的线程函数定义。

6. 实验结论

- 不加线程同步机制的并行线程在访问共享资源时容易产生数据竞争，会导致运算错误等一系列问题。开发者需要在程序开发中注意线程并行同步机制的问题，注意避免数据竞争，保护共享资源。
- 多线程并发过程中的主要额外时间开销是进程切换开销和CPU调度开销，开发者需要将这两个开销与多线程并行计算和多核并行计算的效率优势进行权衡。
- 不同的线程同步机制在实现上有区别，如Peterson算法主要依靠程序逻辑实现，而Linux内核则是通过系统调用实现互斥机制。同一同步机制在不同条件下（如不同程序、不同系统）效率也可能有所不同，因为这些条件既会影响线程切换和CPU切换的开销，也会影响计算速度等别的因素。选择合适的机制可以提高效率。

7. 附录

7.1. none_sync_1程序的完整输出

```
(base) kejingfan@KJF-Huawei-PC:~/code$ ./linux_sync
[transB:0] nTemp1 = 0
[transB:0] nTemp2 = 0
[transB:0] nAccount1 = 0 + 3 = 3
[transB:0] nAccount2 = 0 - 3 = -3
[transB:1] nTemp1 = 3
[transB:1] nTemp2 = -3
[transB:1] nAccount1 = 3 + 6 = 9
[transB:1] nAccount2 = -3 - 6 = -9
[transB:2] nTemp1 = 9
[transB:2] nTemp2 = -9
[transB:2] nAccount1 = 9 + 7 = 16
```

```
[transB:2] nAccount2 = -9 - 7 = -16
[transB:3] nTemp1 = 16
[transB:3] nTemp2 = -16
[transA:0] nTemp1 = 16
[transA:0] nTemp2 = -16
[transA:0] nAccount1 = 16 + 7793 = 7809
[transA:0] nAccount2 = -16 - 7793 = -7809
[transA:1] nTemp1 = 7809
[transA:1] nTemp2 = -7809
[transA:1] nAccount1 = 7809 + 8335 = 16144
[transA:1] nAccount2 = -7809 - 8335 = -16144
[transA:2] nTemp1 = 16144
[transA:2] nTemp2 = -16144
[transA:2] nAccount1 = 16144 + 5386 = 21530
[transA:2] nAccount2 = -16144 - 5386 = -21530
[transA:3] nTemp1 = 21530
[transA:3] nTemp2 = -21530
[transA:3] nAccount1 = 21530 + 492 = 22022
[transA:3] nAccount2 = -21530 - 492 = -22022
[transA:4] nTemp1 = 22022
[transA:4] nTemp2 = -22022
[transA:4] nAccount1 = 22022 + 6649 = 28671
[transA:4] nAccount2 = -22022 - 6649 = -28671
[transA:5] nTemp1 = 28671
[transA:5] nTemp2 = -28671
[transA:5] nAccount1 = 28671 + 1421 = 30092
[transA:5] nAccount2 = -28671 - 1421 = -30092
[transA:6] nTemp1 = 30092
[transA:6] nTemp2 = -30092
[transA:6] nAccount1 = 30092 + 2362 = 32454
[transA:6] nAccount2 = -30092 - 2362 = -32454
[transA:7] nTemp1 = 32454
[transA:7] nTemp2 = -32454
[transA:7] nAccount1 = 32454 + 27 = 32481
[transA:7] nAccount2 = -32454 - 27 = -32481
[transA:8] nTemp1 = 32481
[transA:8] nTemp2 = -32481
[transA:8] nAccount1 = 32481 + 8690 = 41171
[transA:8] nAccount2 = -32481 - 8690 = -41171
[transB:3] nAccount1 = 16 + 5 = 21
[transB:3] nAccount2 = -16 - 5 = -21
[transA:9] nTemp1 = 41171
[transA:9] nTemp2 = -21
[transA:9] nAccount1 = 41171 + 59 = 41230
[transA:9] nAccount2 = -21 - 59 = -80
error
[transA] nTemp1=41171, nTemp2=-21, nRandom=59, nTemp1+nRandom=41230, nTemp2-
nRandom=-80, nAccount1=41230, nAccount2=-80
error
[transB] nTemp1=16, nTemp2=-16, nRandom=5, nTemp1+nRandom=21, nTemp2-
nRandom=-21, nAccount1=41230, nAccount2=-80
program running time: 0.000237
author: 柯劲帆
```

7.2. 参考资料

[1] The futex() system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address (while the addresses for the same memory in separate processes may not be equal, the kernel maps them internally so the same memory mapped in different locations will correspond for futex() calls). This system call is typically used to implement the contended case of a lock in shared memory.

[2] In computing, POSIX Threads, commonly known as pthreads, is an execution model that exists independently from a programming language, as well as a parallel execution model. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.