南京農業大學

操作系统选修实验

实践设计报告

题　　目: 利用 Win Api 函数获取机器时间并保存

姓　　名:

学　　院: 信息科技学院

专　　业: 计算机科学与技术

班　　级:

学　　号:

指导教师: 姜海燕　职称: 教授

2019 年 10 月 6 日

一、 实验目的

掌握 Win32 Api 函数的使用方法，体验操作系统程序接口。

二、 Win32 API 函数原型及说明

1) GetModuleFileNameEx 函数

说明	
获取一个已装载模板的完整路径名称	
返回值	
Long，如执行成功，返回复制到 lpFileName 的实际字符数量；零表示失败。会设置 GetLastError	
参数表	
参数	类型及说明
hModule	Long，一个模块的句柄。可以是一个 DLL 模块，或者是一个应用程序的实例句柄
lpFileName	String，指定一个字串缓冲区，要在其中容纳文件的用 NULL 字符中止的路径名，hModule 模块就是从这个文件装载进来的
nSize	Long，装载到缓冲区 lpFileName 的最大字符数量
注解	
在 Windows 95 下，函数会核查应用程序的内部版本号是否为 4.0 或更大的一个数字。如果是，就返回一个长文件名，否则返回短文件名	

2) CreateFile 函数

① 函数原型：

```
HANDLE WINAPI CreateFile(
_In_ LPCTSTR lpFileName,
_In_ DWORD dwDesiredAccess,
_In_ DWORD dwShareMode,
_In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
_In_ DWORD dwCreationDisposition,
_In_ DWORD dwFlagsAndAttributes,
_In_opt_ HANDLE hTemplateFile
```

```
);
```

② 返回值：

Long，如执行成功，则返回文件句柄。INVALID_HANDLE_VALUE 表示出错，会设置 GetLastError。即使函数成功，但若文件存在，且指定了 CREATE_ALWAYS 或 OPEN_ALWAYS，GetLastError 也会设为 ERROR_ALREADY_EXISTS

③ 函数声明：

```
HANDLE CreateFile(LPCTSTR lpFileName, //普通文件名或者设备文件名
DWORD dwDesiredAccess, //访问模式（写/读）
DWORD dwShareMode, //共享模式
LPSECURITY_ATTRIBUTES lpSecurityAttributes, //指向安全属性的指针
DWORD dwCreationDisposition, //如何创建
DWORD dwFlagsAndAttributes, //文件属性
HANDLE hTemplateFile //用于复制文件句柄
);
```

3) Rdtsc 函数

① 函数原型：

```
#define RDTSC(qp) \
        do { \
            unsigned long lowPart, highPart; \
            __asm__ __volatile__("rdtsc" : "=a" (lowPart), "=d" (highPart)); \
            qp = (((unsigned long long) highPart) << 32) | lowPart; \
        } while (0);
```

② 说明：

RDTSC 指令不是序列化指令。（这就造成了后文中提到的乱序问题）这样，在读取计数器之前，它没有必要等到前面的所有指令都已执行。类似地，在执行读取操作之前，后面的指令也可以开始执行。

4) QueryPerformanceFrequency 函数

① 功能：

检索性能计数器的频率。

② 说明：

性能计数器的频率是一致的。因此，只需在应用程序初始化时查询频率，可以缓存结果。

③ 原型：

BOOL QueryPerformanceFrequency(LARGE_INTEGER *lpFrequency);

④ 返回值：

非零，硬件支持高精度计数器；零，硬件不支持，读取失败。

5) WriteFile 函数

① 原型：

```
BOOL WriteFile(
HANDLE  hFile,//文件句柄
LPCVOID lpBuffer,//数据缓存区指针
DWORD   nNumberOfBytesToWrite,//你要写的字节数
LPDWORD lpNumberOfBytesWritten,//用于保存实际写入字节数的存储区域的指针
LPOVERLAPPED lpOverlapped//OVERLAPPED 结构体指针
);
```

② 参数 类型及说明：

hFile Long，一个文件的句柄
lpBuffer Any，参数类型:指针,指向将写入文件的 数据缓冲区
nNumberOfBytesToWrite Long，要写入数据的字节数量。如写入零字节，表示什么都不写入，但会更新文件的“上一次修改时间”。针对位于远程系统的命名管道，限制在 65535 个字节以内
lpNumberOfBytesWritten Long，实际写入文件的字节数量（此变量是用来返回的 ）
lpOverlapped OVERLAPPED，倘若在指定 FILE_FLAG_OVERLAPPED 的前提下打开文件，这个参数就必须引用一个特殊的结构。那个结构定义了一次异步写操作。否则，该参数应置为空（将声明变为 ByVal As Long，并传递零值）

6) DeleteFile 函数

① 用法

DeleteFile 方法删除指定文件。
object.DeleteFile (filespec[, force]);

② 参数

Object 必选项。 应为 FileSystemObject 的名称。
Filespec 必选项。 要删除的文件的名称。 *filespec* 可以在最后的路径成分中包含通配字符。
Force 可选项。 Boolean 值，如果要删除设置了只读属性的文件，则为 true ；如果不删除则为 false （默认）。

三、 实验关键函数实现

7) 程序介绍

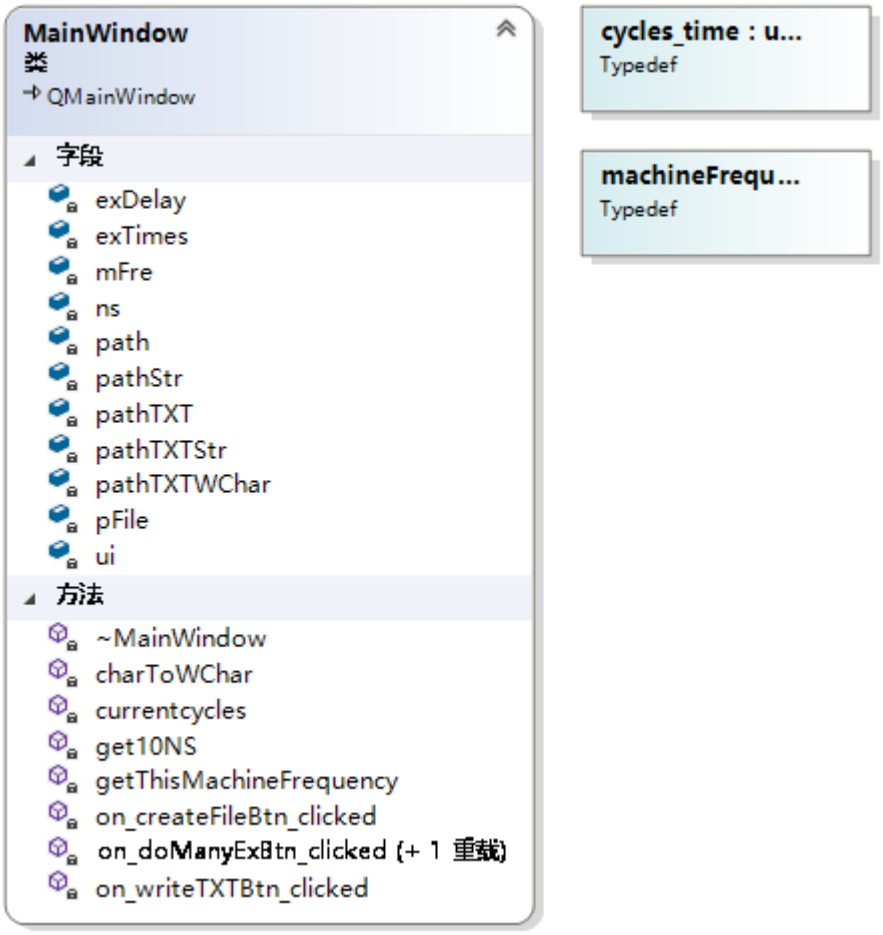


Figure 三-1 软件类图

如上图，软件依据面向对象的编程思想，设置了若干成员对象和方法。

① 成员对象介绍

```
typedef unsigned long long cycles_time;
//用来存放 64 位的时间戳计数器里面的内容
typedef unsigned long long machineFrequency;
//用来存放机器主频

char path[MAX_PATH];//当前程序地址
QString pathStr;//当前程序地址--QString 形式

char pathTXT[MAX_PATH];//TXT 文件的位置
QString pathTXTStr;//当前 TXT 文件地址--QString 形式
WCHAR pathTXTWChar[MAX_PATH];//当前 TXT 文件地址--WCHAR 形式
```

```
HANDLE pFile;//TXT 文件指针

machineFrequency mFre;//本机主频
cycles_time ns[10];//存放 10 次获取 rdtsc 的数组

inline cycles_time currentcycles();//获取当前的 rdtsc
inline void get10NS();//连续获取 10 个 rdtsc
inline machineFrequency getThisMachineFrequency();//获取本机主频

int exTimes=0;//已经进行的实验次数
int exDelay=0;//所有实验产生的间隔（每次实验产生 9 次间隔）
```

② 成员函数介绍[1]

➢ 生成 EXE 文件和 TXT 文件的地址的各种编码（MainWindow 构造函数）

函数原型：

```
explicit MainWindow(QWidget *parent = nullptr);
```

函数实现：

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

DWORD size=GetModuleFileNameA(nullptr,path,MAX_PATH);
//MAX_PATH 在 minwindef.h 的第 33 行有定义：#define MAX_PATH 260
    pathStr=path;
    if (size != 0){
        int lenPath=strlen(path);
        while(path[lenPath]!='\\'){
            path[lenPath--]='\0';//给地址结尾添上字符串结束标志
        }
        strcpy(pathTXT,path);//文件所在地址
        strcat(pathTXT,TXT_FILE_NAME);//加上文件名
        pathTXTStr=pathTXT;
    }else{
        exit(0); //若获取失败，则直接退出程序
```

[1] 该部分代码中以 ui 开头的语句可以忽略。它们和界面有关。

```
        }

        srand(time(0));//随机数的种子

        ui->exTimesLcdNum->setStyleSheet("border: 1px solid green;
color: green; background: silver;");
        ui->exTimesLcdNum->setDigitCount(8);
        ui->doManyExLineEdit->setValidator(new
QIntValidator(1,100000,this));
    }
```

➢ 检查并删除现有文件

函数原型：

```
void on_delExistBtn_clicked(bool checked);
```

函数实现：

```
void MainWindow::on_delExistBtn_clicked(bool checked)
{
    int fileStatusValue;

    fileStatusValue=DeleteFile(pathTXTWChar);
    if(fileStatusValue==ERROR_FILE_NOT_FOUND){
        ui->delFileLab->setText("没有找到文件。可以直接进行程序。
");
    }else if(fileStatusValue==ERROR_ACCESS_DENIED){
        ui->delFileLab->setText("有文件，但是被设置成了只读文件。
");
    }else {
        ui->delFileLab->setText("已经找到文件并删除。现在没有文件
了。");
    }
    exTimes=0;
    exDelay=0;

ui->delayLab->setText(QString::number(exDelay*1.0/(exTimes*10),'g',6)
);
    ui->exTimesLcdNum->display(exTimes);//改变记分板上的试验次数
}
```

➢ Char 字符数组向 WCHAR 字符数组转换

函数原型：

```
bool charToWChar(char* src,WCHAR *dest);
```

函数实现：

```
bool MainWindow::charToWChar(char* src,WCHAR *dest){
    memset(dest,0,sizeof(dest));


MultiByteToWideChar(CP_ACP,0,src,strlen(src)+1,dest,sizeof(dest)/size
of(dest[0]));//MultiByteToWideChar 函数是将多字节转换为宽字节的一个
API 函数 char* 转 LPCWSTR

    return true;
}
```

➢ 创建一个 TXT 文件并写入文件表头

函数原型：

```
void on_createFileBtn_clicked(bool checked);
```

函数实现：

```
void MainWindow::on_createFileBtn_clicked(bool checked)
{
    pFile=CreateFile(
                pathTXTStr.toStdWString().c_str(),//文件地址
                GENERIC_READ|GENERIC_WRITE,//文件可读可写
                FILE_SHARE_READ,//保护期间，可读不可写
                NULL,//设为 NULL，子进程不能继承本句柄
                CREATE_ALWAYS,//创建文件，会改写前一个文件。虽然之
前 delete 过，但出于鲁棒性还是这么写吧。
                FILE_ATTRIBUTE_NORMAL,//文件属性设置为默认
                NULL//不设置文件句柄
                );
    if (pFile == INVALID_HANDLE_VALUE)
    {
        ui->createFileLab->setText("失败！文件未创建。");
        CloseHandle(pFile);
    }else {
        ui->createFileLab->setText("成功！文件已创建。");
        char FileHead[] =
"id\t\tname\t\tinputTime\t\tns\t\truntimes\r\n";//写文件头
        WriteFile(pFile, FileHead, strlen(FileHead), NULL, NULL);//
写入 TXT 文件
    }
}
```

➢ 连续获取 10 次 rdtsc

函数原型：

```
inline void MainWindow::get10NS()
```

函数实现：

```
inline void MainWindow::get10NS(){
    Sleep(0);//释放当前线程所剩余的时间片（如果有剩余的话），尽量让下面 20 条语句享有完整的时间片

    /*
     * 1 - 不设置一个不断自增的变量是为了节省运行时的时间。
     * 2 - 每句写 2 遍是因为取一次需要 10ns，而题目刚好要求 20ns。
     */
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[0]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[0]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[1]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[1]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[2]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[2]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[3]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[3]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[4]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[4]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[5]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[5]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[6]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[6]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[7]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[7]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[8]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[8]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[9]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[9]));
}
```

➢ 获取本机主频

函数原型：

```
inline machineFrequency MainWindow::getThisMachineFrequency()
```

函数实现：

```
inline machineFrequency MainWindow::getThisMachineFrequency(){
    LARGE_INTEGER fre;
    QueryPerformanceFrequency(&fre);
    return (machineFrequency)fre.QuadPart;
}
```

➢ 向 TXT 文件写入一组（10 条）记录

函数原型：

```
void on_writeTXTBtn_clicked(bool checked);
```

函数实现：

```
void MainWindow::on_writeTXTBtn_clicked(bool checked)
{
    //===设置变量
    char write_content[100] = { 0 }; //写入内容的数组
    QTime current_time;//用于获得时分秒
    int rand_num[10];//随机数

    //===生成随机数
    for (int i=0;i<10;i++) {
        rand_num[i]=rand()%50+1;
    }

    //===获取主频
    mFre=getThisMachineFrequency();

    //===获取 10 个 rdtsc
    get10NS();

    //===实验组号，方便多次实验时观察记录
    strcat(write_content,"第");

strcat(write_content,QString::number(++exTimes).toStdString().data())
;
    strcat(write_content,"次实验（插入 10 条记录）");
    strcat(write_content, "\r\n"); //换行
```

```
        WriteFile(pFile, write_content, strlen(write_content), NULL,
NULL);

        //===写记录，循环写入 10 条
        for(int id=0;id<10;id++){
            memset(write_content,0,100);//清空 write_content 数组

            //===接上 id===

strcat(write_content,QString::number(id).toStdString().data());//写入
记录的编号

            //===接上 name===
            strcat(write_content, "\t\t"); //接上间隔
            strcat(write_content, "像素"); //接上名字

            //===接上 input===
            strcat(write_content, "\t\t"); //接上间隔
            strcat(write_content,
QTime::currentTime().toString("hh:mm:ss").toStdString().data());//接
上当前的时分秒

            //===接上 ns===
            strcat(write_content, "\t\t"); //接上间隔

strcat(write_content,QString::number((DWORD)(((double(ns[id]))/mFre)*
1000000)).toStdString().data()); //接上 ns
            //===接上随机数===
            strcat(write_content, "\t"); //接上间隔
            strcat(write_content,
QString::number(rand_num[id]).toStdString().data()); //写入随机数
            strcat(write_content, "\r\n"); //换行

            //===写文件===
            WriteFile(pFile, write_content, strlen(write_content),
NULL, NULL);//向 TXT 文件写入
        }

        //将本组间隔时间加入累计
        for(int i=0;i<9;i++){
            exDelay+=(int)(((double(ns[i+1]-ns[i]))/mFre)*1000000);
        }
```

```
ui->delayLab->setText(QString::number(exDelay*1.0/(exTimes*9),'g',6))
;//更新平均间隔时间
        ui->exTimesLcdNum->display(exTimes);//更新记分板上的试验次数
    }
```

➢ 进行指定次数的实验

函数原型：

```
void on_doManyExBtn_clicked();
```

函数实现：

```
void MainWindow::on_doManyExBtn_clicked()
{
    int exTimes=0;//接收用户指定的实验次数
    exTimes=ui->doManyExLineEdit->text().toInt();
    for(int i=0;i<exTimes;i++){
        on_writeTXTBtn_clicked(true);//一次实验
    }
}
```

③ 程序流程介绍

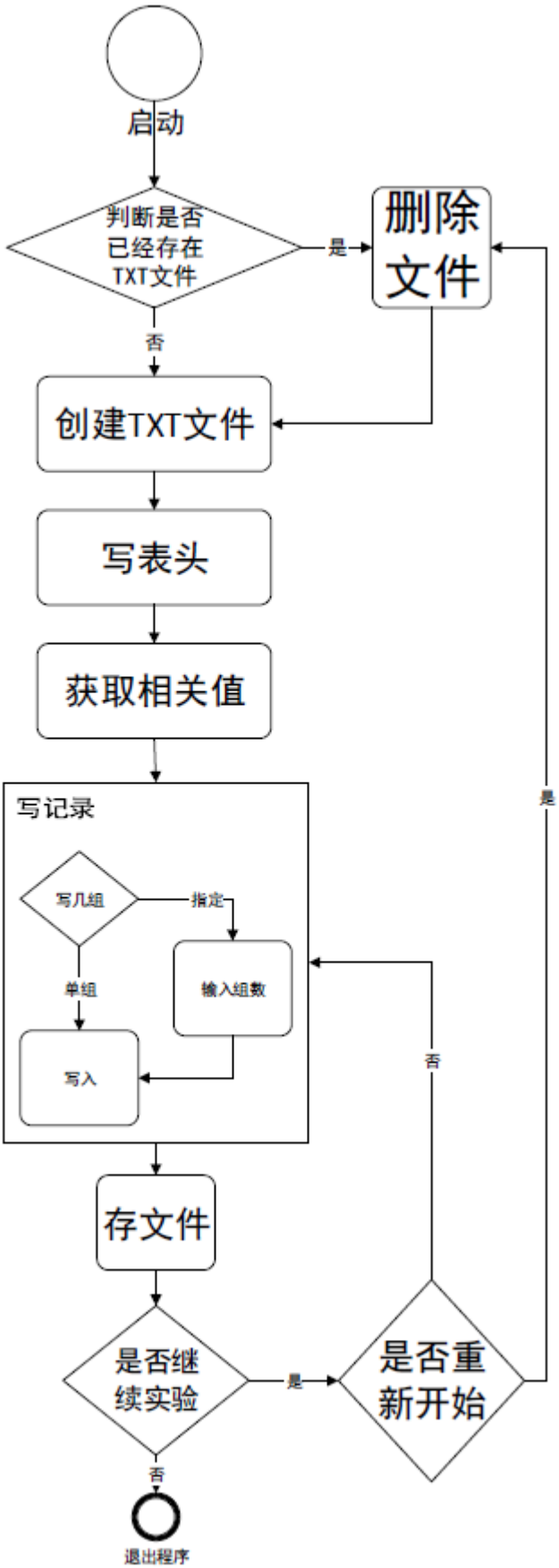

启动
判断是否已经存在TXT文件
是
删除文件
否
创建TXT文件
写表头
获取相关值
写记录
写几组
指定
单组
输入组数
写入
存文件
是否继续实验
是
是否重新开始
否
是
否
退出程序

Figure 三-2 软件流程图

四、 实验测试

1) 运行环境

① 软件环境

软件在 64 位 Windows 10 企业版下使用 Qt Creator 4.9.2 编写、运行。

② 硬件环境

处理器	Intel Core i5-7200U CPU @ 2.50GHz 2.70GHz（注：编写时插电工作）
内存	8.00GB

Table 四-1 运行环境

1) 输入说明

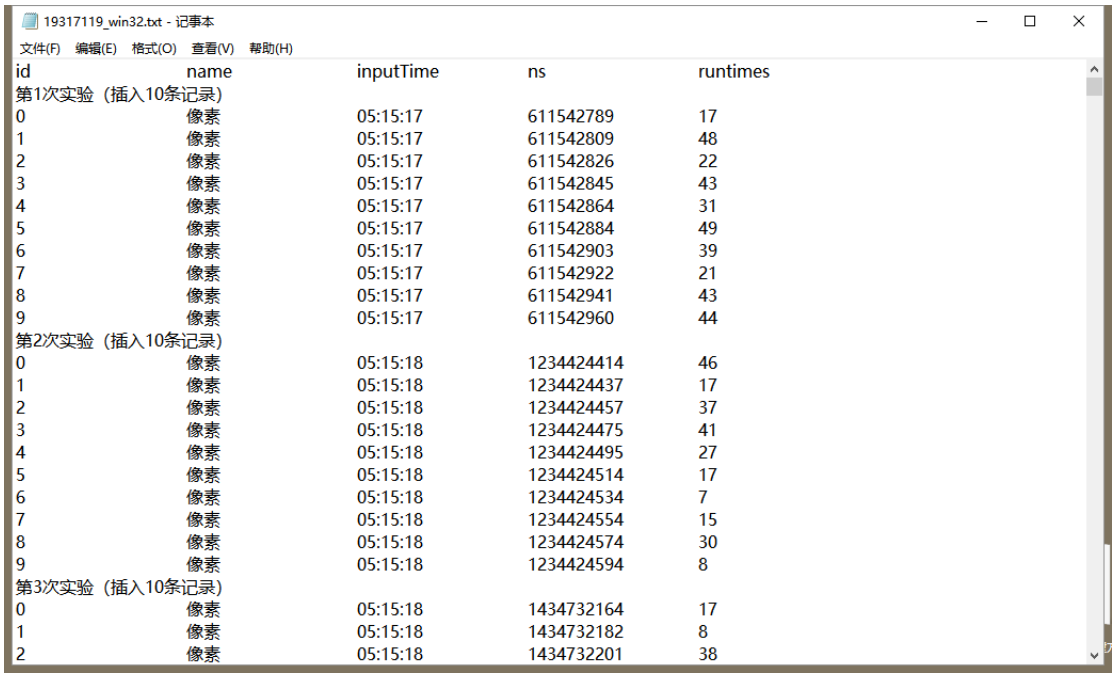
Figure 四-1 输入

如图所示，可以写一组或指定组数写入。

2) 输出说明

Figure 四-2 输出-软件显示

如图四-2，软件会对记录写入情况进行统计。



19317119_win32.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

id	name	inputTime	ns	runtimes
第1次实验（插入10条记录）				
0	像素	05:15:17	611542789	17
1	像素	05:15:17	611542809	48
2	像素	05:15:17	611542826	22
3	像素	05:15:17	611542845	43
4	像素	05:15:17	611542864	31
5	像素	05:15:17	611542884	49
6	像素	05:15:17	611542903	39
7	像素	05:15:17	611542922	21
8	像素	05:15:17	611542941	43
9	像素	05:15:17	611542960	44
第2次实验（插入10条记录）				
0	像素	05:15:18	1234424414	46
1	像素	05:15:18	1234424437	17
2	像素	05:15:18	1234424457	37
3	像素	05:15:18	1234424475	41
4	像素	05:15:18	1234424495	27
5	像素	05:15:18	1234424514	17
6	像素	05:15:18	1234424534	7
7	像素	05:15:18	1234424554	15
8	像素	05:15:18	1234424574	30
9	像素	05:15:18	1234424594	8
第3次实验（插入10条记录）				
0	像素	05:15:18	1434732164	17
1	像素	05:15:18	1434732182	8
2	像素	05:15:18	1434732201	38

Figure 四-3 输出-文件输出

如图四-3，也可以打开 19317119_win32.txt 查看记录（文件体积过大时可能打不开）。

文件内容格式为一个记录表格，表头是：

Id name inputTime ns runtimes

其中，id 为当前记录序号，name 为名字，inputTime 为输入时间，ns 为当前机内时间，精确到纳秒，runtimes 保存[1-50]内随机整数。

当多次实验时，组数可能上万，因此每组有一个标题：“第 XX 次实验（插入 10 条记录）”

3) 程序测试

① 目录获取

通过 Qt 自带的 qdebug.h 类，可以见到程序正确获得了地址：

```
05:13:12: Starting F:\Code\OS\build-fileManagementOS-Desktop_Qt_5_9_1_MinGW_32bit-Debug\debug\fileManagementOS.exe ...
"程序当前目录：F:\\Code\\OS\\build-fileManagementOS-Desktop_Qt_5_9_1_MinGW_32bit-Debug\\debug\\fileManagementOS.exe"
"文本文件地址：F:\\Code\\OS\\build-fileManagementOS-Desktop_Qt_5_9_1_MinGW_32bit-Debug\\debug\\19317119_win32.txt"
```

Figure 四-4 获取地址

② 创建文件

点击创建文件，显然文件被创建了。

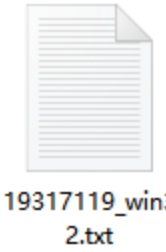


Figure 四-5 创建文件

③ 写入表头

打开文件，显然表头也写入了。

Figure 四-6 写入表头

④ 写文件内容

点击写入记录，显然记录也写入了。

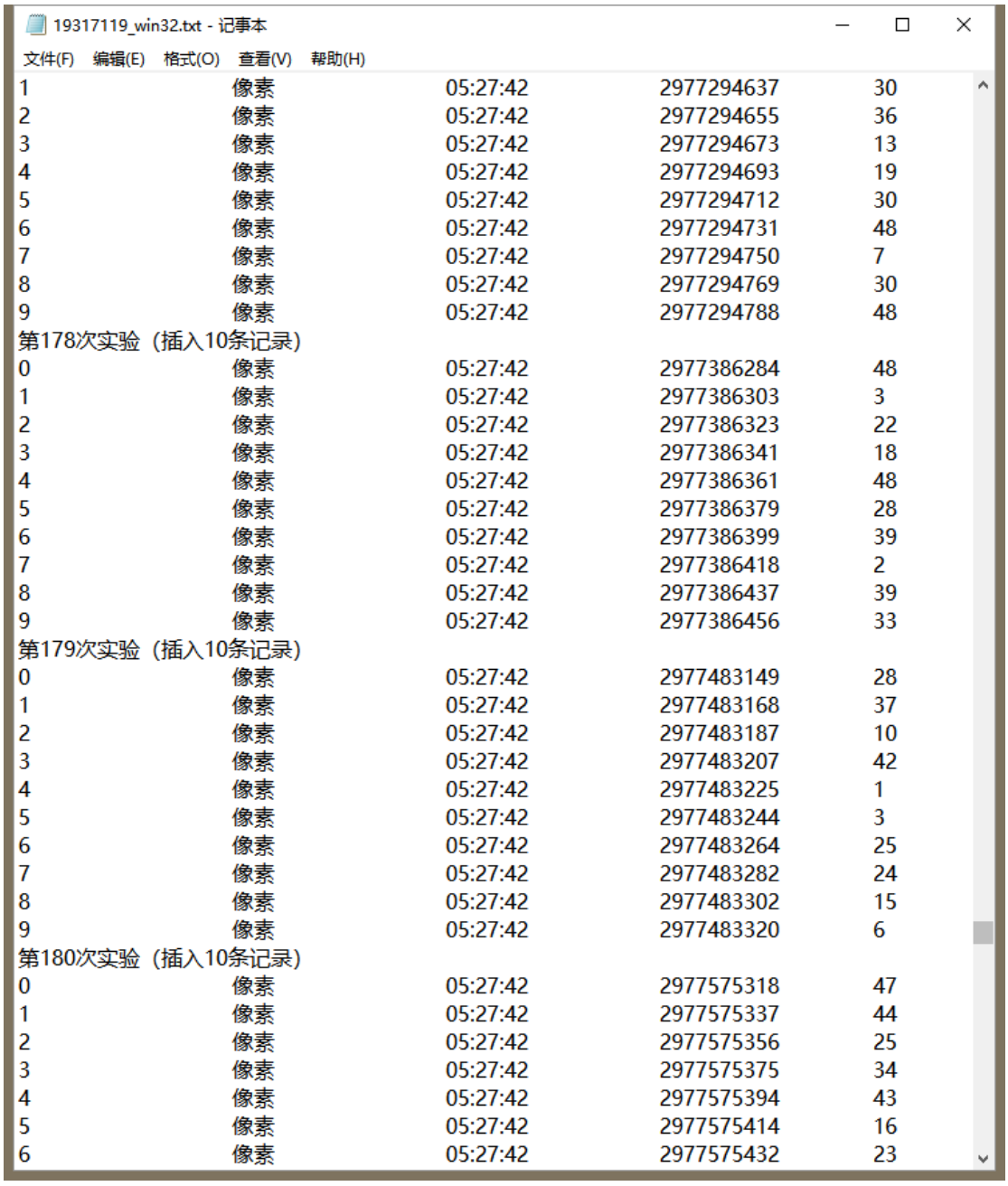


Figure 四-7 写入记录

⑤ 统计信息

看看软件，信息也正确地统计了出来。比较接近２０秒。

Figure 四-8 统计信息

五、 技术问题解决方案

1) RDTSC 相关问题

RDTSC 是一条由 Intel Pentium 添加的，用于在及其微观的时间单位里管理计算机[1]的指令。它是一个用于时间戳计数器的 64 位的寄存器，在每个时钟信号到来时加一。通过它可以计算 CPU 的主频，比如：如果微处理器的主频是 1MHZ 的话，那么 TSC 就会在 1 秒内增加 1000000。它的使用出现了各种问题……

① 不能保证同一块主板上每个核的 TSC 是同步的

这个问题本来我是想研究 http://lwn.net/Articles/211051/来解决的，但是又看到 Intel 的处理器手册里[2]说 Intel 的 Nehalem 以及之后的处理器（在 AMD 那边，是 Bacelona 之后的），可以大胆使用 rdtsc，不用担心变频，不用担心多核，time stamp counter 以恒定速率增加。所以决定不做处理。

2) QueryPerformanceFrequency 相关问题

① CPU 的时钟频率可能变化

比如，CPU 的降频模式。这个问题在 Ubuntu 之类系统下很好解决，改一些配置文件即可[3]。在 Windows 下，有些电脑可以提供了便捷选项，没有便捷选项的电脑通过软件也可以解决这个问题[4]。

3) 线程相关问题

① 程序被分到多个时间片

核心的获取 RDTSC 的代码一共 20 行，如果被分到多个时间片，则间隔会远超 20ns。为了避免这个问题，可以通过 Sleep(0)来放弃这个线程剩下的时间碎片[5]，从一个新的线程开始执行这组代码。

```
//连续获取10次rdtsc
inline void MainWindow::get10NS(){
    Sleep(0);//释放当前线程所剩余的时间片（如果有剩余的话），尽量让下面20条语句享有完整的时间片

    /*
     * 1 - 不设置一个不断自增的变量是为了节省运行时的时间。
     * 2 - 每句写2遍是因为取一次需要10ns，而题目刚好要求20ns。
     */
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[0]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[0]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[1]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[1]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[2]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[2]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[3]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[3]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[4]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[4]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[5]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[5]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[6]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[6]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[7]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[7]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[8]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[8]));

    __asm__ __volatile__ ("rdtsc" : "=A" (ns[9]));
    __asm__ __volatile__ ("rdtsc" : "=A" (ns[9]));
}
```

Figure 五-1Sleep(0)的使用

4) 代码被“优化”相关问题

① out-of-order execution 问题

奔腾® II 处理器和奔腾® Pro 处理器都支持 out-of-order execution，导致指令可能“被优化”执行顺序，导致 rdtsc 指令执行的时间和预期的不同。这可以用 CPUID 指令来强制执行顺序。

比如一段汇编程序是这样的：

```
rdtsc              ; read time stamp
mov time, eax      ; move counter into variable
fdiv               ; floating-point divide
rdtsc              ; read time stamp
sub eax, time      ; find the difference
```

则 RDTSC 指令有可能在 f d i v 指令之前执行。

加上 CPUID 之后：

```
cpuid                   ; force all previous instructions to comple
rdtsc                   ; read time stamp counter
mov     time, eax       ; move counter into variable
fdiv                    ; floating-point divide
cpuid                   ; wait for FDIV to complete before RDTSC
rdtsc                   ; read time stamp counter
sub     eax, time       ; find the difference
```

Figure 五-2 通过 CPUID 指令来强制顺序执行的代码

就可以保证执行顺序。

② 嵌入式汇编代码被优化问题

在 C++里面插入汇编代码，可能会遭到优化。为了避免这个情况，可以在调用时加上 __volatile__限定符，表示不要对嵌入式汇编进行优化。[6]

```
//连续获取10次rdtsc
inline void MainWindow::get10NS(){
    Sleep(0);//释放当前线程所剩余的时间片（如果有剩余的话），尽量让下面20条语句享有完整的时间片

    /*
     * 1 - 不设置一个不断自增的变量是为了节省运行时的时间。
     * 2 - 每句写2遍是因为取一次需要10ns，而题目刚好要求20ns。
     */
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[0]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[0]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[1]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[1]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[2]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[2]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[3]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[3]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[4]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[4]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[5]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[5]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[6]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[6]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[7]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[7]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[8]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[8]));

    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[9]));
    __asm__ __volatile__ ( "rdtsc" : "=A" (ns[9]));
}
```

Figure 五-3 避免嵌入式汇编被优化

六、 实验心得

拿到题目的时候，我感觉这是要在商业系统里面干一件对于特殊系统很轻松的事情。因为，在“装 Windows 的普通 PC 机”这种对性能要求远大于极致控制的环境，从硬件到软件都有太多为了性能而做出的设计，比如：cache、流水线、多发射、乱序执行、CPU 变频、多核、多线程等等……这些设计在这个题目里面全是导致不确定性的罪魁祸首。对于一些极端的环境，可能压根就是几条强制性指令的事情，比如导弹发射系统（我猜的）。我能做的，就是把这么多不确定因素中的某些给确定下来，并且加以控制。

于是我就开始一个一个问题地去搜索。但是正如上面所说，这个问题牵扯到的东西太多，仅仅几天的工夫是做不完的。为了方便实验，我索性将软件完善了下，做了统计和批量实验的功能。解决了外围工作之后，继续实验各种小技巧。

我理想中的思路，应该是有一个预备程序，先测出这台电脑执行各种指令的速度，然后像搭积木一样，通过选取不同长短（执行时间）的积木（指令）来搭建到一个设定好的高度（ 2

０ｎｓ）。

但是，增加了批量添加记录之后，明显可以看到：添加记录的频率越高、次数越多，间隔的平均值就越低。这是在所有的情境下都复现的一个现象。所以我想，如果真的要再严格一点保证这个２０ｎｓ，那还得是个动态调整的过程……太难了。仿佛是一个人要在剧烈晃动的甲板上保持平稳。

虽然看了很多，但并没有机会全部实践下去，所以我只是保证在我的电脑上基本达到２０ｎｓ，在其他设备上并不能保证。

在解决这个问题的过程中，无意间了解了从硬件到软件的许多小知识。比如多核处理器是怎么通过 CDCM6208 时钟分配芯片来同步时钟、通过仲裁电路来协调进程；多线程时代，false sharing 其实会使得在单核跑得快的程序在多核变慢；处理器是怎么因为多核放弃了 RDTSC 然后在多代之后做好了同步又把它拾回来继续用……这些知识没啥用，但今日格一物，明日格一物，不就是致知之道吗？

七、 参考文献

（要求：按照《南京农业大学自然科学版》文献要求格式给出，可以是网文）

[1] http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM

[2]http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[3] https://blog.csdn.net/u013597671/article/details/56679121

[4] https://jingyan.baidu.com/article/3065b3b6bee580becff8a40a.html

[5] https://www.cnblogs.com/wwkk/p/10386045.html

[6] https://blog.csdn.net/zdjxinyu/article/details/91480386