Windows 操作系统

C/C++ 程序实验

姓名：_____陈展博_____

学号：_____1221001003__

班级：_____计科 1 班____

院系：_____信工_____

_____2024_____年__12___月__4___日

实验九　Windows　快速文件系统

一、背景知识

二、实验目的

三、工具/准备工作

四、实验内容与步骤

1. 快速文件系统

步骤 1：登录进入 Windows 。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio Code”。

步骤 3：新建项目名为“9-1”，并且新建项“9-1.cpp”。

步骤 4：将项目所需 txt 文件复制到项目文件夹下。

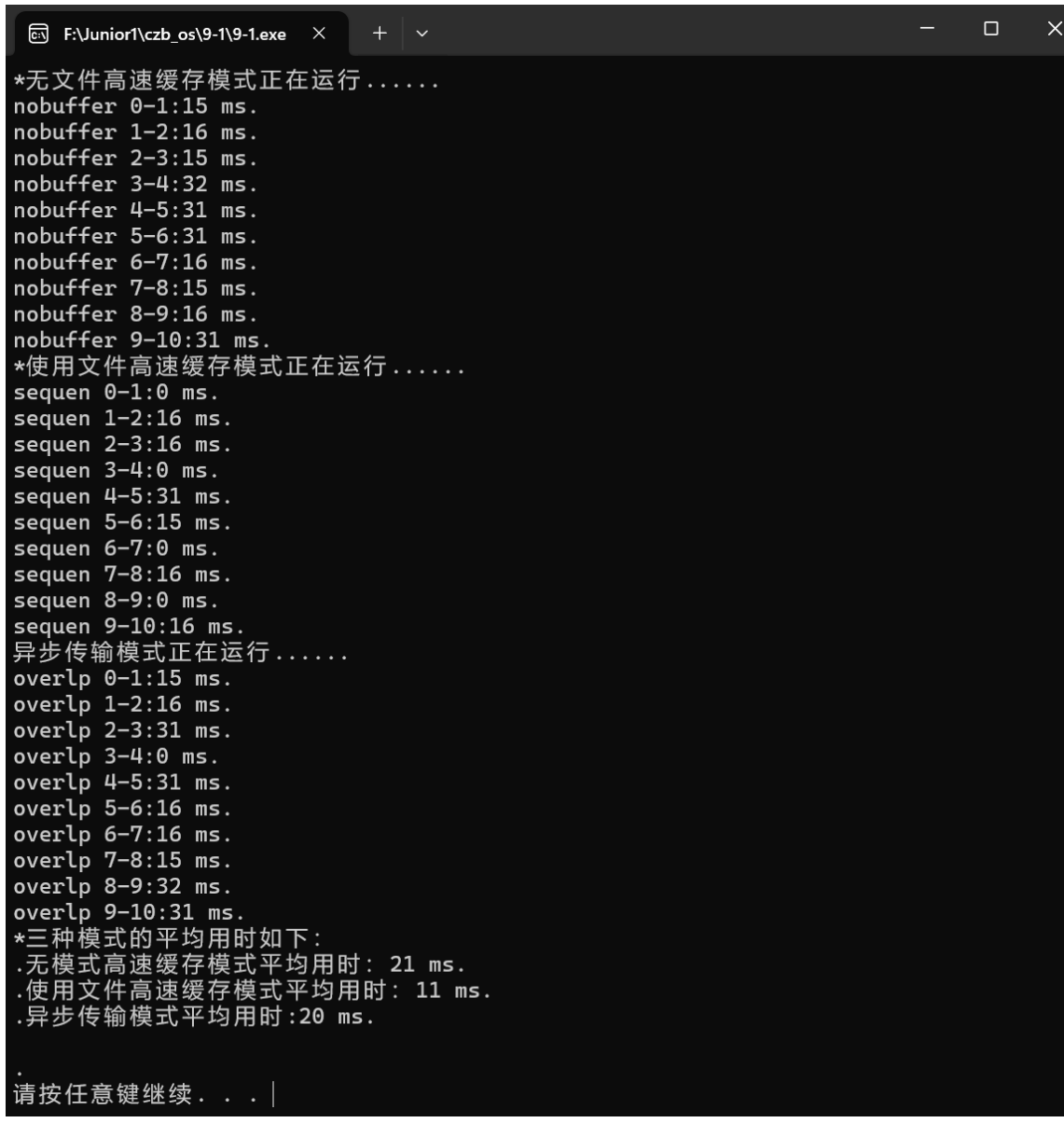
步骤 5：按“F5”开始调试，注意路径里不要含有中文。

步骤 6：按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。

操作能否正常进行？如果不行，则可能的原因是什么？

操作能够正常进行，如果不能正常进行，可能的原因是因为文件路径中包含中文，或者代码中的字符集有错误。

运行结果是：



图片 1 运行结果截图

结果分析：

使用文件高速缓存模式对文件进行操作的平均用时会比使用无模式高速缓存模式和异步传输模式的平均用时要快很多。这里异步传输模式和无高速缓存模式的速度差不多是因为这里主要调用文件操作的函数，若能增加更多与文件操作无关的操作，异步传输模式会更有优势。

选作：探究如何设计程序可以更好发挥出异步传输的性能，并尝试实现。

参考提示：设计一个函数 int filter(char source，char *sink，int f)，其中：

1、source：源文件，即从哪个文件读。

2、sink：目标文件，即写到哪个文件。

3、f：一个与文件内容无关的操作（比如空循环）。

a)仍然设计三种模式的 filter 函数：无缓存模式、文件高速缓存模式和异步模式。

b)给出 10 个大小相同内容不同的源文件，分别调用三种模式的 filter 函数 10 次，读出文件的内容，并写到 10 个不同的目标文件中去。

c)在调用 filter 函数的过程中，加入一些与文件内容无关的操作，在此情况下观察三种模式之间的用时区别（与文件无关的操作耗时不能太短，否则效果将不明显）。

应该注意的是：在调用 filter 函数时，要加入与文件内容无关的 f 操作；由于用了 10 个不同的源文件，对每种传输模式，可以考虑创建 10 个线程来并发地执行文件传输。

d)记录每次函数调用的时间，以及 10 次操作的总时间，分析实习结果，从中体会异步传输模式的优越性，写出心得体会。

请描述你所做的工作：

```
#define BUFFER_SIZE 2048    //定义缓冲区的大小，这里设为2048字节
char*   buffer;             //这里的缓冲区被定义成char型

// 添加与文件内容无关的操作
void perform_unrelated_work() {
    const int count = 10e9; // 模拟耗时操作
    volatile int dummy = 0;
    for (int i = 0; i < count; ++i) {
        dummy += i % 7;
    }
}

void process_files_nobuffer(void (*filter_func)(char*, char*, void (*)(char*)),
                    const std::vector<std::pair<std::string, std::string>>& file_pairs,
                    void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}
```

图片 2 编写 nobuffer 多线程处理文件函数，将每次对文件处理单独作为一个线程处理

```cpp
void process_files_sequen(void (*filter_func)(char*, char*, void (*)(char*)),
                          const std::vector<std::pair<std::string, std::string>>& file_pairs,
                          void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}

void process_files_overlp(void (*filter_func)(char*, char*, void (*)(char*)),
                          const std::vector<std::pair<std::string, std::string>>& file_pairs,
                          void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}
```

图片 3 编写 sequen 和 overlp 多线程处理文件函数，将每次对文件处理单独作为一个线程处理

```
void f1(char* addr) {
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            sum +=i;
        }
        //  *addr = (unsigned char)*addr + 1;
    }
void f2(char* addr) {

    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            sum +=i;
        }
        //*addr = (unsigned char)*addr - 1;
        }
void f3(char* addr) {
    *addr = (unsigned char)*addr * 1;
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}
void f4(char* addr) {
    // *addr = (unsigned char)*addr >> 1;
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}
```

图片 4 对 f1234 函数做修改，减少文件操作比重

```
void f5(char* addr) {
    // *addr = (unsigned char)*addr << 1;
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}
```

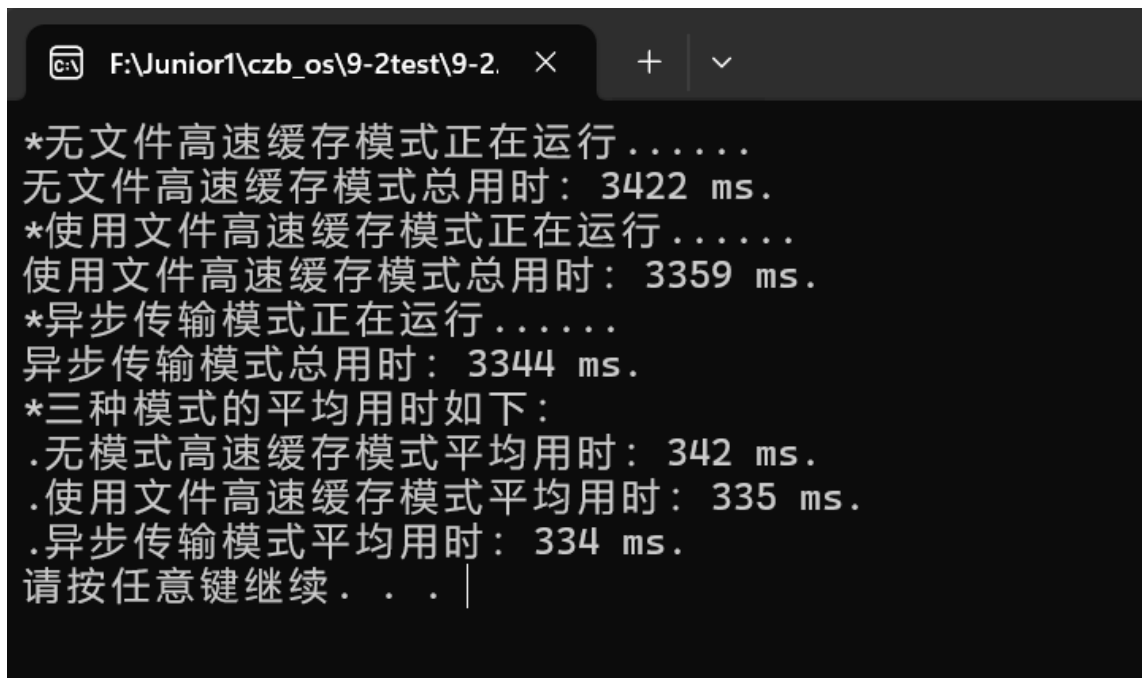
图片 5 对 f5 函数修改，减少文件操作比重

```
void main()
{
    //分配缓冲区
    buffer = new char[BUFFER_SIZE];
    //记录执行filter函数的开始时间
    DWORD tick;

    //用于三种模式各自的平均时间
    DWORD nobuffer_average_time = 0;
    DWORD sequen_average_time = 0;
    DWORD overlp_average_time = 0;

    // 定义文件对和函数指针
    vector<pair<string, string>> file_pairs_nobuffer = {
        {"source.txt", "nobuffer_1.txt"}, {"source_1.txt", "nobuffer_2.txt"},
        {"source_2.txt", "nobuffer_3.txt"}, {"source_3.txt", "nobuffer_4.txt"},
        {"source_4.txt", "nobuffer_5.txt"}, {"source_5.txt", "nobuffer_6.txt"},
        {"source_6.txt", "nobuffer_7.txt"}, {"source_7.txt", "nobuffer_8.txt"},
        {"source_8.txt", "nobuffer_9.txt"}, {"source_9.txt", "nobuffer_10.txt"}};

    vector<pair<string, string>> file_pairs_sequen = {
        {"source.txt", "sequen_1.txt"}, {"source_1.txt", "sequen_2.txt"},
        {"source_2.txt", "sequen_3.txt"}, {"source_3.txt", "sequen_4.txt"},
        {"source_4.txt", "sequen_5.txt"}, {"source_5.txt", "sequen_6.txt"},
        {"source_6.txt", "sequen_7.txt"}, {"source_7.txt", "sequen_8.txt"},
        {"source_8.txt", "sequen_9.txt"}, {"source_9.txt", "sequen_10.txt"}};

    vector<pair<string, string>> file_pairs_overlp = {
        {"source.txt", "overlp_1.txt"}, {"source_1.txt", "overlp_2.txt"},
        {"source_2.txt", "overlp_3.txt"}, {"source_3.txt", "overlp_4.txt"},
        {"source_4.txt", "overlp_5.txt"}, {"source_5.txt", "overlp_6.txt"},
        {"source_6.txt", "overlp_7.txt"}, {"source_7.txt", "overlp_8.txt"},
        {"source_8.txt", "overlp_9.txt"}, {"source_9.txt", "overlp_10.txt"}};

    void (*funcs[])(char*) = {f1, f2, f3, f4, f5, f1, f2, f3, f4, f5};
```

图片 6 主函数写明输入输出文件对

```cpp
// 记录时间并运行无缓存模式
cout << "*无文件高速缓存模式正在运行......" << endl;
DWORD start_time = GetTickCount();
process_files_nobuffer(filter_nobuffer, file_pairs_nobuffer, funcs);
DWORD nobuffer_time = GetTickCount() - start_time;
cout << "无文件高速缓存模式总用时：" << nobuffer_time << " ms." << endl;

// 记录时间并运行文件缓存模式
cout << "*使用文件高速缓存模式正在运行......" << endl;
start_time = GetTickCount();
process_files_sequen(filter_sequen, file_pairs_sequen, funcs);
DWORD sequen_time = GetTickCount() - start_time;
cout << "使用文件高速缓存模式总用时：" << sequen_time << " ms." << endl;

    // 记录时间并运行异步模式
cout << "*异步传输模式正在运行......" << endl;
 start_time = GetTickCount();
process_files_overlp(filter_overlp, file_pairs_overlp, funcs);
DWORD overlp_time = GetTickCount() - start_time;
cout << "异步传输模式总用时：" << overlp_time << " ms." << endl;

// 输出平均时间对比
cout << "*三种模式的平均用时如下：" << endl;
cout << ".无模式高速缓存模式平均用时：" << nobuffer_time / 10 << " ms." << endl;
cout << ".使用文件高速缓存模式平均用时：" << sequen_time / 10 << " ms." << endl;
cout << ".异步传输模式平均用时：" << overlp_time / 10 << " ms." << endl;

system("pause");
return;
```

图片 7 主函数调用文件操作函数，并记录时间



图片 8 运行结果

名称	修改日期	类型	大小
sequen_4.txt	2024/12/4 16:09	文本文档	486 KB
sequen_5.txt	2024/12/4 16:09	文本文档	486 KB
sequen_6.txt	2024/12/4 16:09	文本文档	486 KB
sequen_7.txt	2024/12/4 16:09	文本文档	486 KB
sequen_8.txt	2024/12/4 16:09	文本文档	486 KB
sequen_9.txt	2024/12/4 16:09	文本文档	486 KB
sequen_10.txt	2024/12/4 16:09	文本文档	486 KB
source.txt	2024/12/4 14:46	文本文档	486 KB
source_1.txt	2024/12/4 14:46	文本文档	485 KB
source_2.txt	2024/12/4 14:46	文本文档	486 KB
source_3.txt	2024/12/4 14:46	文本文档	486 KB
source_4.txt	2024/12/4 14:46	文本文档	486 KB
source_5.txt	2024/12/4 14:47	文本文档	486 KB
source_6.txt	2024/12/4 14:47	文本文档	486 KB
source_7.txt	2024/12/4 14:47	文本文档	486 KB
source_8.txt	2024/12/4 14:47	文本文档	486 KB
source_9.txt	2024/12/4 14:47	文本文档	486 KB

图片 9 源文件一览

实验分析与心得体会

无缓存模式

直接操作硬盘，文件读写速度慢。由于增加了耗时操作，这种模式的性能可能下降更多。

顺序扫描模式

缓存优化后，性能有所提升。适合大文件处理，但仍然受到同步机制的限制。

异步模式

异步操作减少了等待时间，与耗时操作并发运行，表现出显著优势。

附源代码:

```
/*****************************************************************
/*
/*****************************************************************
/*---------------------------------------------------------
/*三种模式
/*  1. FILE_FLAG_NOBUFFER
/*  2. FILE_FLAG_SEQUENTIAL_SCAN
/*  3. FILE_FLAG_BUFFERING|FILE_FLAG_OVERLAPPED
/*---------------------------------------------------------
/*五种操作
```

```
/*   1.   charactor    +1
/*   2.   charactor    -1
/*   3.   charactor    -32
/*   4.   charactor    +32
/*   5.   charactor    *1
/*-------------------------------------------------------------
*/
#include<iostream>
#include<windows.h>
#include <thread>
#include <vector>
#include <string>
using namespace std;
//三种模式
void filter_nobuffer(char* source, char* sink, void(*func)(char* addr));
void filter_sequen(char* source, char* sink, void(*func)(char* addr));
//Overlap 用于异步重叠操作
void filter_overlp(char* source, char* sink, void(*func)(char* addr));
//五种不同功能的操作
void f1(char* addr);
void f2(char* addr);
void f3(char* addr);
void f4(char* addr);
void f5(char* addr);

#define BUFFER_SIZE 2048     //定义缓冲区的大小，这里设为 2048 字节
char*     buffer;                //这里的缓冲区被定义成 char 型

// 添加与文件内容无关的操作
void perform_unrelated_work() {
    const int count = 10e9; // 模拟耗时操作
    volatile int dummy = 0;
    for (int i = 0; i < count; ++i) {
        dummy += i % 7;
    }
}

void process_files_nobuffer(void (*filter_func)(char*, char*, void (*)(char*)),
                    const std::vector<std::pair<std::string, std::string>>& file_pairs,
                    void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
```

```
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}

void process_files_sequen(void (*filter_func)(char*, char*, void (*)(char*)),
                    const std::vector<std::pair<std::string, std::string>>& file_pairs,
                    void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}

void process_files_overlp(void (*filter_func)(char*, char*, void (*)(char*)),
                    const std::vector<std::pair<std::string, std::string>>& file_pairs,
                    void (*funcs[])(char*)) {
    std::vector<std::thread> threads;
    std::vector<std::thread> threads_func1;
    std::vector<std::thread> threads_func2;
    for (size_t i = 0; i < file_pairs.size(); ++i) {
        threads.emplace_back([filter_func, &file_pairs, funcs, i]() {
            filter_func(const_cast<char*>(file_pairs[i].first.c_str()),
```

```
                        const_cast<char*>(file_pairs[i].second.c_str()), funcs[i]);
                        perform_unrelated_work();
        });
    }
    for (auto& t : threads) {
        t.join();
    }
}
/*-----------------------------------------------------------------*/
//对文件内容进行的 5 种操作
//f1   +1
//f2   -1
//f3   *1
//f4   >>
//f5   <<

void f1(char* addr) {
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            sum +=i;
        }
        //   *addr = (unsigned char)*addr + 1;
    }
void f2(char* addr) {

    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            sum +=i;
        }
        //*addr = (unsigned char)*addr - 1;
        }
void f3(char* addr) {
    *addr = (unsigned char)*addr * 1;
    int count = 888;
    int sum = 0;
```

```
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}
void f4(char* addr) {
    // *addr = (unsigned char)*addr >> 1;
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}
void f5(char* addr) {
    // *addr = (unsigned char)*addr << 1;
    int count = 888;
    int sum = 0;
        for (size_t i = 0; i < count; i++)
        {
            /* code */
            // sum +=i;
        }}

/*-------------------------------------------------------------*/
//没有文件高速缓存的 filter 函数
void filter_nobuffer(char* source, char* sink, void(*func)(char* addr))
{
    HANDLE handle_src, handle_dst;  //定义原文件与目标文件的句柄
    BOOL cycle;                     //用来判断一个缓冲区是否被写满
    DWORD NumberOfBytesRead, NumberOfBytesWrite, index; //读的字节数、写的字节数
                                                        //打开原文件
                                                        //因为是 OPEN_EXISTING 所以开始时得新建
source.txt
handle_src = CreateFile(source, GENERIC_READ, NULL, NULL, OPEN_EXISTING,
FILE_FLAG_NO_BUFFERING, NULL);

    //创建目标文件
handle_dst = CreateFile(sink, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL,
NULL);
```

```
    if(handle_src == INVALID_HANDLE_VALUE || handle_dst == INVALID_HANDLE_VALUE)
    {
        cout << "CreatFile Invocation Error!" << endl;
        exit(1);
    }
    cycle = TRUE;

    //用来 cycle 判断文件什么时候读完
    while (cycle)
    {
        //从原文件读数据送入缓冲区
if (ReadFile(handle_src, buffer, BUFFER_SIZE, &NumberOfBytesRead, NULL) == FALSE)
        {
            cout << "ReadFile Error!" << endl;
            exit(1);
        }

        //当读不满一个缓冲区时，说明达到文件末尾，结束循环
        if (NumberOfBytesRead < BUFFER_SIZE)
            cycle = FALSE;
        //对文件内容进行的操作
        for (index = 0; index < NumberOfBytesRead; index++)
            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件
if (WriteFile(handle_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)
        {
            cout << "WriteFile Error!" << endl;
            exit(1);
        }
    }

    //关闭文件句柄
    CloseHandle(handle_src);
    CloseHandle(handle_dst);
}


/*------------------------------------------------------------*/

void filter_sequen(char* source, char* sink, void(*func)(char* addr))
```

```
{
    HANDLE handle_src, handle_dst;  //定义原文件与目标文件的句柄
    BOOL cycle;                     //用来判断一个缓冲区是否被写满
    DWORD NumberOfBytesRead, NumberOfBytesWrite, index; //读的字节数、写的字节数
                                                        //打开原文件
handle_src = CreateFile(source, GENERIC_READ, NULL, NULL, OPEN_EXISTING,
FILE_FLAG_SEQUENTIAL_SCAN, NULL);

    //创建目标文件
handle_dst = CreateFile(sink, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS,
FILE_FLAG_SEQUENTIAL_SCAN, NULL);

if(handle_src == INVALID_HANDLE_VALUE || handle_dst == INVALID_HANDLE_VALUE)
    {
        cout << "CreatFile Invocation Error!" << endl;
        exit(1);
    }
    cycle = TRUE;

    //用来 cycle 判断文件什么时候读完
    while (cycle)
    {
        //从原文件读数据送入缓冲区
if (ReadFile(handle_src, buffer, BUFFER_SIZE, &NumberOfBytesRead, NULL) == FALSE)
        {
            cout << "ReadFile Error!" << endl;
            exit(1);
        }

        //当读不满一个缓冲区时，说明达到文件末尾，结束循环
        if (NumberOfBytesRead < BUFFER_SIZE)
            cycle = FALSE;
        //对文件内容进行的操作
        for (index = 0; index < NumberOfBytesRead; index++)
            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件
if (WriteFile(handle_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)
        {
            cout << "WriteFile Error!" << endl;
            exit(1);
```

```
        }
    }

    //关闭文件句柄
    CloseHandle(handle_src);
    CloseHandle(handle_dst);
}

/*------------------------------------------------------------*/

void filter_overlp(char* source, char* sink, void(*func)(char* addr))
{
    HANDLE handle_src, handle_dst;  //定义原文件与目标文件的句柄
    BOOL cycle;                     //用来判断一个缓冲区是否被写满
    DWORD NumberOfBytesRead, NumberOfBytesWrite, index, dwError;    //读的字节数、写的字节数
    OVERLAPPED overlapped;          //overlapped 结构
                                    //打开原文件
handle_src = CreateFile(source, GENERIC_READ, NULL, NULL, OPEN_EXISTING, FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);

    //创建目标文件
handle_dst = CreateFile(sink, GENERIC_WRITE, NULL, NULL, CREATE_ALWAYS, NULL, NULL);

    if(handle_src == INVALID_HANDLE_VALUE || handle_dst == INVALID_HANDLE_VALUE)
    {
        cout << "CreatFile Invocation Error!" << endl;
        exit(1);
    }
    cycle = TRUE;

    //对 overlapped 结构初始化
    overlapped.hEvent = NULL;
    overlapped.Offset = -BUFFER_SIZE;
    overlapped.OffsetHigh = 0;

    //用来 cycle 判断文件什么时候读完
    while (cycle)
    {
        //计算文件的偏移量
```

```
        overlapped.Offset = overlapped.Offset + BUFFER_SIZE;

        //从原文件读数据送入缓冲区
if (ReadFile(handle_src, buffer, BUFFER_SIZE, &NumberOfBytesRead, &overlapped) == FALSE)
        {
            switch (dwError = GetLastError())
            {
                //读文件结尾
            case ERROR_HANDLE_EOF:
                cycle = FALSE;
                break;
            case ERROR_IO_PENDING:
if (GetOverlappedResult(handle_src, &overlapped, &NumberOfBytesRead, TRUE) == FALSE)
                {
                    cout << "GetOverlappedResult Error!" << endl;
                    exit(1);
                }
                break;
            default:
                break;
            }
        }

        //当不满一个个缓存区时，说明达到文件末尾，结束循环
        if (NumberOfBytesRead < BUFFER_SIZE)
            cycle = FALSE;

        //对文件内容进行的操作
        for (index = 0; index < NumberOfBytesRead; index++)
            func(&buffer[index]);

        //将缓冲区中的数据写入目标文件
if (WriteFile(handle_dst, buffer, NumberOfBytesRead, &NumberOfBytesWrite, NULL) == FALSE)
        {
            cout << "WriteFile Error!" << endl;
            exit(1);
        }
    }

    //关闭文件句柄
    CloseHandle(handle_src);
```

```
    CloseHandle(handle_dst);

}

void main()
{
    //分配缓冲区
    buffer = new char[BUFFER_SIZE];
    //记录执行 filter 函数的开始时间
    DWORD tick;

    //用于三种模式各自的平均时间
    DWORD nobuffer_average_time = 0;
    DWORD sequen_average_time = 0;
    DWORD overlp_average_time = 0;

    // 定义文件对和函数指针
    vector<pair<string, string>> file_pairs_nobuffer = {
        {"source.txt", "nobuffer_1.txt"}, {"source_1.txt", "nobuffer_2.txt"},
        {"source_2.txt", "nobuffer_3.txt"}, {"source_3.txt", "nobuffer_4.txt"},
        {"source_4.txt", "nobuffer_5.txt"}, {"source_5.txt", "nobuffer_6.txt"},
        {"source_6.txt", "nobuffer_7.txt"}, {"source_7.txt", "nobuffer_8.txt"},
        {"source_8.txt", "nobuffer_9.txt"}, {"source_9.txt", "nobuffer_10.txt"}};

    vector<pair<string, string>> file_pairs_sequen = {
        {"source.txt", "sequen_1.txt"}, {"source_1.txt", "sequen_2.txt"},
        {"source_2.txt", "sequen_3.txt"}, {"source_3.txt", "sequen_4.txt"},
        {"source_4.txt", "sequen_5.txt"}, {"source_5.txt", "sequen_6.txt"},
        {"source_6.txt", "sequen_7.txt"}, {"source_7.txt", "sequen_8.txt"},
        {"source_8.txt", "sequen_9.txt"}, {"source_9.txt", "sequen_10.txt"}};

    vector<pair<string, string>> file_pairs_overlp = {
        {"source.txt", "overlp_1.txt"}, {"source_1.txt", "overlp_2.txt"},
        {"source_2.txt", "overlp_3.txt"}, {"source_3.txt", "overlp_4.txt"},
        {"source_4.txt", "overlp_5.txt"}, {"source_5.txt", "overlp_6.txt"},
        {"source_6.txt", "overlp_7.txt"}, {"source_7.txt", "overlp_8.txt"},
        {"source_8.txt", "overlp_9.txt"}, {"source_9.txt", "overlp_10.txt"}};

    void (*funcs[])(char*) = {f1, f2, f3, f4, f5, f1, f2, f3, f4, f5};
```

```
    // 记录时间并运行无缓存模式
    cout << "*无文件高速缓存模式正在运行....." << endl;
    DWORD start_time = GetTickCount();
    process_files_nobuffer(filter_nobuffer, file_pairs_nobuffer, funcs);
    DWORD nobuffer_time = GetTickCount() - start_time;
    cout << "无文件高速缓存模式总用时：" << nobuffer_time << " ms." << endl;

    // 记录时间并运行文件缓存模式
    cout << "*使用文件高速缓存模式正在运行....." << endl;
    start_time = GetTickCount();
    process_files_sequen(filter_sequen, file_pairs_sequen, funcs);
    DWORD sequen_time = GetTickCount() - start_time;
    cout << "使用文件高速缓存模式总用时：" << sequen_time << " ms." << endl;

        // 记录时间并运行异步模式
    cout << "*异步传输模式正在运行....." << endl;
     start_time = GetTickCount();
    process_files_overlp(filter_overlp, file_pairs_overlp, funcs);
    DWORD overlp_time = GetTickCount() - start_time;
    cout << "异步传输模式总用时：" << overlp_time << " ms." << endl;

    // 输出平均时间对比
    cout << "*三种模式的平均用时如下：" << endl;
    cout << ".无模式高速缓存模式平均用时：" << nobuffer_time / 10 << " ms." << endl;
    cout << ".使用文件高速缓存模式平均用时：" << sequen_time / 10 << " ms." << endl;
    cout << ".异步传输模式平均用时：" << overlp_time / 10 << " ms." << endl;

    system("pause");
    return;

}


/************************THE END*******************************/
```