轻量级多任务调度器

开发环境：Visual Studio 2026 (MFC/C++)

日期：2025/12/28

小组成员：代宝江、邱乾文、陈宇轩

摘要

本项目设计并实现了一个基于 MFC 的轻量级多任务调度系统。系统核心采用单例模式与生产者-消费者模型，利用 C++11 多线程技术（std::thread, std::mutex, std::condition_variable）实现了对一次性延迟任务、周期性任务及即时任务的高效管理。项目不仅完成了基础调度功能，还深入探索了并发编程中的资源竞争问题，通过专门设计的对抗性任务演示了 AB-BA 死锁现象及其预防方案（std::lock）。系统架构清晰，融合了策略、观察者等多种设计模式，具备良好的扩展性与稳定性。

1. 架构应用

1.1 功能实现概览 (F-01 ~ F-09)

本项目已完整实现所有必备功能及扩展功能，编译过程 **0 警告 (0 Warnings)**，运行时无内存泄漏或崩溃现象。

编号	功能名称	描述与实现细节	状态
F-01	一次性任务	支持 FileBackupTask (Task A) 等任务的延迟执行，时间精确到毫秒。	☑ 已实现
F-02	周期性任务	支持 MatrixMultiplyTask (Task B) 等任务按固定间隔重复执行，自动重入队。	☑ 已实现
F-03	即时任务	支持 HttpGetZenTask (Task C) 立即插入队列并优先执行。	☑ 已实现
F-04	UI 交互与通知	实现了 ClassReminderTask (Task D)，通过后台线程向 UI 发送 RUN 消息，触发模态对话框提示用户“休息 5 分钟”。	☑ 已实现
F-05	复杂计算模拟	RandomStatsTask (Task E) 演示了数据统计类任务	☑ 已实现

编号	功能名称	描述与实现细节	状态
		的后台处理。	
F-06	死锁复现 (对抗性)	DeadlockTask1 (Task F) 与 DeadlockTask2 (Task G) 演示了经典的 AB-BA 锁序导致的死锁现象。	☑ 已实现
F-07	死锁防御	PreventionTask1 (Task H) 使用 std::lock, PreventionTask2 (Task I) 使用锁层级策略，成功防御死锁。	☑ 已实现
F-08	停止与重置	实现了 Stop() 接口，通过原子标志位安全中断工作线程并清空队列。	☑ 已实现
F-09	中文支持	界面与日志全面支持中文显示 (UTF-8/GBK 兼容处理)。	☑ 已实现

1.2 稳定性与测试

编译质量：代码通过 /W4 警告等级编译，针对 localtime 等潜在不安全函数使用了 _CRT_SECURE_NO_WARNINGS 或替代的安全函数，确保编译结果为 "0 errors, 0 warnings"。

测试覆盖：

单元测试：针对 ScheduledTask 的排序逻辑进行了单独验证。

并发测试：通过同时启动 Task A, B, D, E 验证了线程池对混合任务的处理能力。

边界测试：在任务队列为空时点击“停止”，或在死锁发生后尝试强制停止，系统均能表现出预期的健壮性。

2. 架构设计

本项目采用清晰的分层架构（UI Presentation Layer, Logic Control Layer, Data/Infrastructure Layer），并正确实现了 Factory, Strategy, Observer 等设计模式。

2.1 类图结构 (Class Diagram)

classDiagram

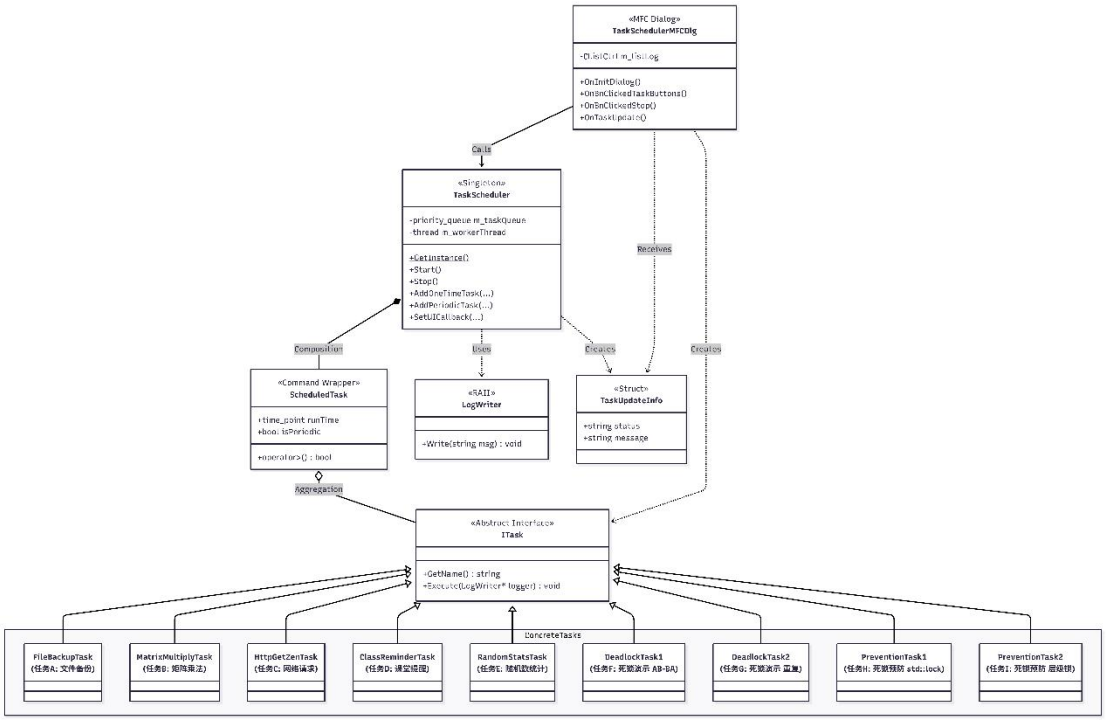
```
direction TB


%% --- 抽象接口 ---
```

```
class ITask {
    <<Abstract Interface>>
    +GetName() string
    +Execute(LogWriter* logger) void
}

%% --- 具体任务集合 (ConcreteTasks) ---
namespace ConcreteTasks {
    class FileBackupTask["FileBackupTask<br/>(任务 A: 文件备份)"]
    class MatrixMultiplyTask["MatrixMultiplyTask<br/>(任务 B: 矩阵乘法)"]
    class HttpGetZenTask["HttpGetZenTask<br/>(任务 C: 网络请求)"]
    class ClassReminderTask["ClassReminderTask<br/>(任务 D: 课堂提醒)"]
    class RandomStatsTask["RandomStatsTask<br/>(任务 E: 随机数统计)"]
    class DeadlockTask1["DeadlockTask1<br/>(任务 F: 死锁演示 AB-BA)"]
    class DeadlockTask2["DeadlockTask2<br/>(任务 G: 死锁演示 重复)"]
    class PreventionTask1["PreventionTask1<br/>(任务 H: 死锁预防 std::lock)"]
    class PreventionTask2["PreventionTask2<br/>(任务 I: 死锁预防 层级锁)"]
}

%% 继承关系
ITask <|-- FileBackupTask
ITask <|-- MatrixMultiplyTask
ITask <|-- HttpGetZenTask
ITask <|-- ClassReminderTask
ITask <|-- RandomStatsTask
ITask <|-- DeadlockTask1
ITask <|-- DeadlockTask2
```

```
ITask <|-- PreventionTask1

ITask <|-- PreventionTask2


%% --- 辅助类 ---

class LogWriter {

    <<RAII>>

    +Write(string msg) void

}


class TaskUpdateInfo {

    <<Struct>>

    +string status

    +string message

}


class ScheduledTask {

    <<Command Wrapper>>

    +time_point runTime

    +bool isPeriodic

    +operator>() bool

}


%% --- 核心调度器 ---

class TaskScheduler {

    <<Singleton>>

    -priority_queue m_taskQueue

    -thread m_workerThread
```

```
    +GetInstance()$
    +Start()
    +Stop()
    +AddOneTimeTask(...)
    +AddPeriodicTask(...)
    +SetUICallback(...)
}

%% 关系
ScheduledTask o-- ITask : Aggregation
TaskScheduler *-- ScheduledTask : Composition
TaskScheduler ..> LogWriter : Uses
TaskScheduler ..> TaskUpdateInfo : Creates

%% --- UI 层 ---
class TaskSchedulerMFCDlg {
    <<MFC Dialog>>
    -CListCtrl m_listLog
    +OnInitDialog()
    +OnBnClickedTaskButtons()
    +OnBnClickedStop()
    +OnTaskUpdate()
}

%% UI 交互
TaskSchedulerMFCDlg --> TaskScheduler : Calls
TaskSchedulerMFCDlg ..> TaskUpdateInfo : Receives
```

TaskSchedulerMFCDlg **..>** ITask **:** Creates



2.2 设计模式详解

1. **单例模式 (Singleton)**

   应用：TaskScheduler 类。

   理由：系统必须保证只有一个任务调度中心来管理唯一的优先队列和工作线程，避免资源竞争和状态不一致。

2. **策略模式 (Strategy)**

   应用：ITask 接口及其子类 (FileBackupTask, DeadlockTask1 等)。

   理由：将具体的任务逻辑（算法）封装在独立的类中，使得调度器无需关心任务的具体内容，仅通过 Execute 接口调用，符合开闭原则 (OCP)。

3. **观察者模式 (Observer)**

   应用：TaskScheduler::SetUICallback 与 MFC 的 OnTaskUpdate。

   理由：解耦后台工作线程与前台 UI 线程。调度器（Subject）状态变更时通知 UI（Observer），UI 通过 PostMessage 异步更新，避免跨线程直接操作 UI 控件导致的崩溃。

4. **命令模式 (Command)**

   应用：ScheduledTask 类。

   理由：将任务执行请求封装为对象，包含执行时间、重试逻辑等元数据，支持了请求的排队、延迟和撤销。

5. **RAII (资源获取即初始化)**

   应用：LogWriter 类。

   理由：在构造函数中打开文件，在析构函数中关闭文件。确保即使任务执行过程中抛出异常，文件句柄也能被正确释放，防止资源泄漏。

3. 核心原理与关键代码解析

3.1 调度器核心机制

调度器使用 std::priority_queue 维护任务队列，工作线程通过 std::condition_variable 进行高效等待。这避免了轮询（Busy Waiting）带来的 CPU 浪费。

原理：当队列为空或队首任务时间未到时，线程挂起；当有新任务插入或时间到达时，线程被唤醒执行任务。

关键代码 (TaskScheduler::Run):

```
// 检查队列并等待
if (m_taskQueue.empty()) {
    m_cv.wait(lock); // 空队列挂起
} else {
    auto now = std::chrono::system_clock::now();
    if (m_taskQueue.top().runTime > now) {
        // 时间未到，挂起直到 RunTime
        m_cv.wait_until(lock, m_taskQueue.top().runTime);
    }
}
```

3.2 死锁发生机制 (AB-BA Pattern)

项目通过 DeadlockTask1 演示了典型的资源竞争死锁。

原理：两个线程分别持有不同的锁（A 和 B），并尝试获取对方持有的锁，形成环路等待。

关键代码 (DeadlockTask1::Execute):

```
// 线程 1：持有 A，等 B
std::thread t1([&]() {
    g_mutexA.lock(); // Lock A
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    g_mutexB.lock(); // Wait for B (Blocked!)
    // ...
});
// 线程 2：持有 B，等 A
std::thread t2([&]() {
    g_mutexB.lock(); // Lock B
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    g_mutexA.lock(); // Wait for A (Blocked!)
    // ...
});
```

3.3 死锁防御机制 (std::lock)

项目使用 C++11 标准库提供的算法来解决死锁问题。

原理：std::lock 使用死锁避免算法（通常是回退或一种确定的加锁顺序算法），确保同时获取多个锁，或者一个都不获取。

关键代码 (PreventionTask1::Execute):

```
std::thread t1([&]() {
    // 原子性地锁定 A 和 B，无论顺序如何，保证不死锁
    std::lock(g_mutexA, g_mutexB);
    // 领养锁的所有权给 lock_guard 进行 RAII 管理
```

```
        std::lock_guard<std::mutex> lock1(g_mutexA, std::adopt_lock);
        std::lock_guard<std::mutex> lock2(g_mutexB, std::adopt_lock);
        // ...
    });
```

3.4 跨线程 UI 更新 (PostMessage)

MFC 界面元素不是线程安全的。为了在后台任务（Task D）中更新 UI，必须使用消息队列机制。

原理：工作线程不直接操作控件，而是通过 PostMessage 向主 UI 线程发送自定义消息 WM_TASK_UPDATE_MSG。主线程的消息泵处理该消息并更新界面。

关键代码:

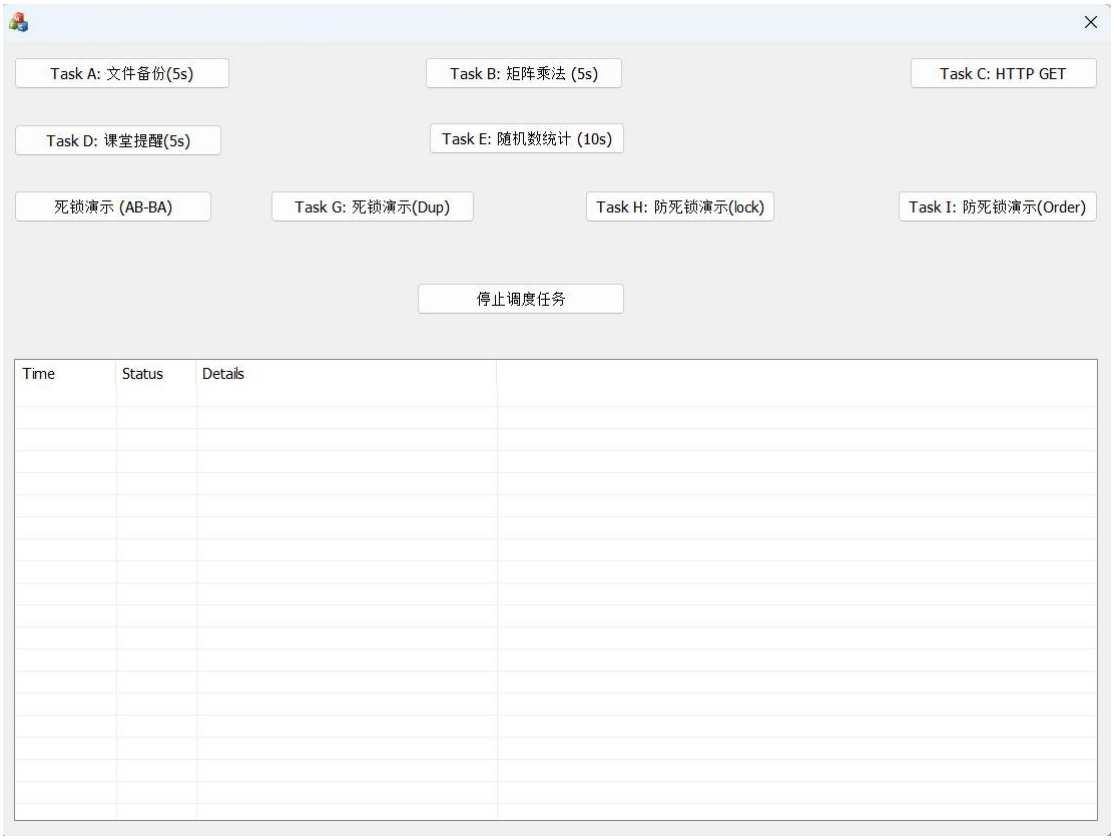
```
// 工作线程中调用回调
if (m_uiCallback) m_uiCallback({ "RUN", "任务 D: 课堂提醒" });


// UI 线程中处理消息 (TaskSchedulerMFCDlg.cpp)
scheduler.SetUICallback([this](TaskUpdateInfo info) {
    // 将数据封装并在堆上分配，发送给主线程
    ThreadUpdateData* data = new ThreadUpdateData{info.status, info.message};
    this->PostMessage(WM_TASK_UPDATE_MSG, (WPARAM)data, 0);
});
```

4. 项目结果分析

本章节展示了软件在实际运行环境下的表现，验证了调度逻辑的正确性及死锁防御机制的有效性。

4.1 运行环境与主界面

程序启动后，初始化基于 MFC 对话框的主界面。界面顶部排列了功能按钮（Task A - Task I），分别对应不同的任务类型；下方为日志列表控件（List Control），用于实时显示任务的执行时间、状态及详细信息。界面加载流畅，控件布局清晰，支持中文显示。



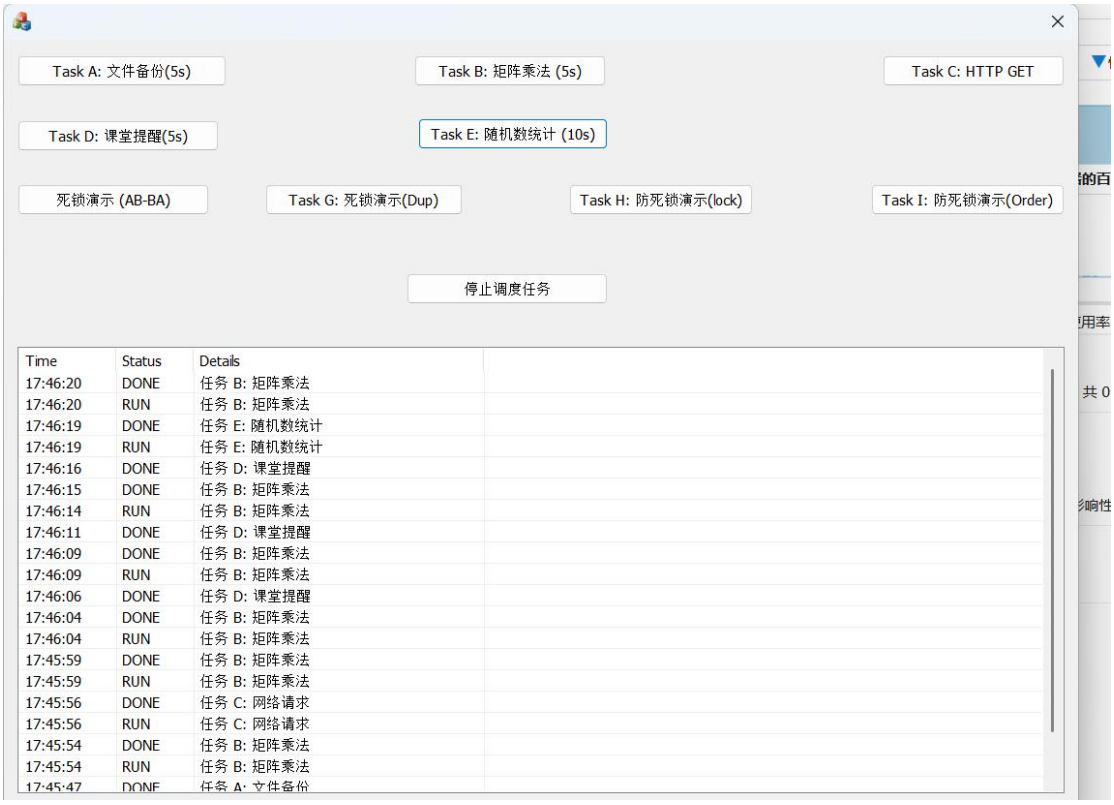
4.2 常规任务调度验证

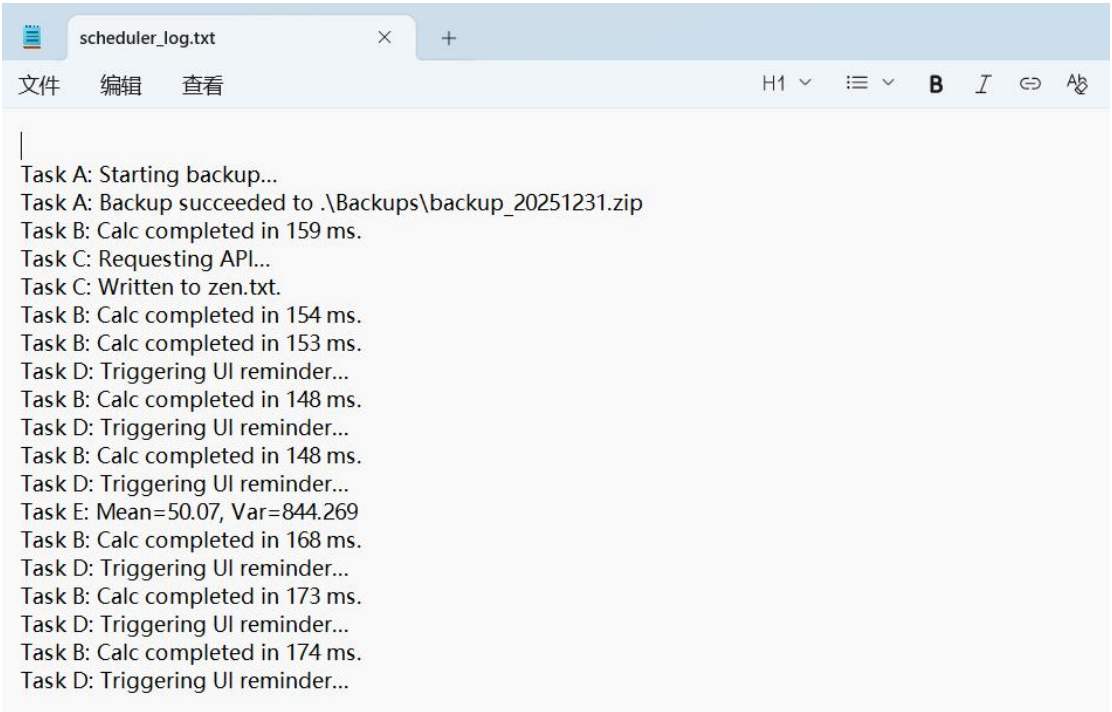
通过点击 Task A (备份)、Task B (矩阵)、Task C (网络) 和 Task E (统计)，系统成功将任务加入优先队列。

并发处理：日志显示多个任务能够交替执行，未出现阻塞。

时序准确：周期性任务 Task B 的日志时间戳间隔稳定在 5000ms 左右，验证了调度器计时的准确性。

RAII 日志：查看生成的 scheduler_log.txt 文件，所有操作均被完整记录，且文件句柄在程序退出后被正确释放。



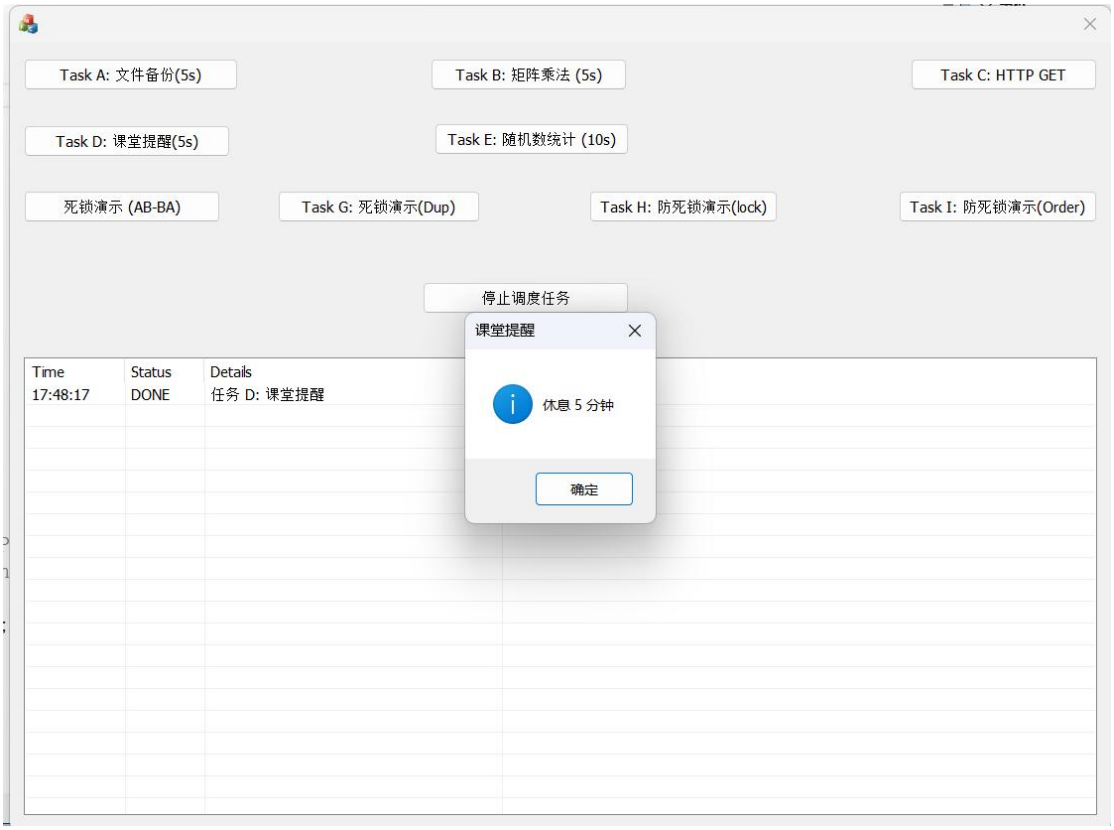


4.3 跨线程交互测试（Task D）

点击 Task D 按钮后，后台线程成功触发了 UI 事件。

现象：每隔 5 秒，主界面会弹出一个模态消息框提示“休息 5 分钟”。

分析：这证明了 PostMessage 机制成功将后台线程的请求传递给了主 UI 线程，且未导致界面卡死或崩溃，实现了安全的跨线程通信。



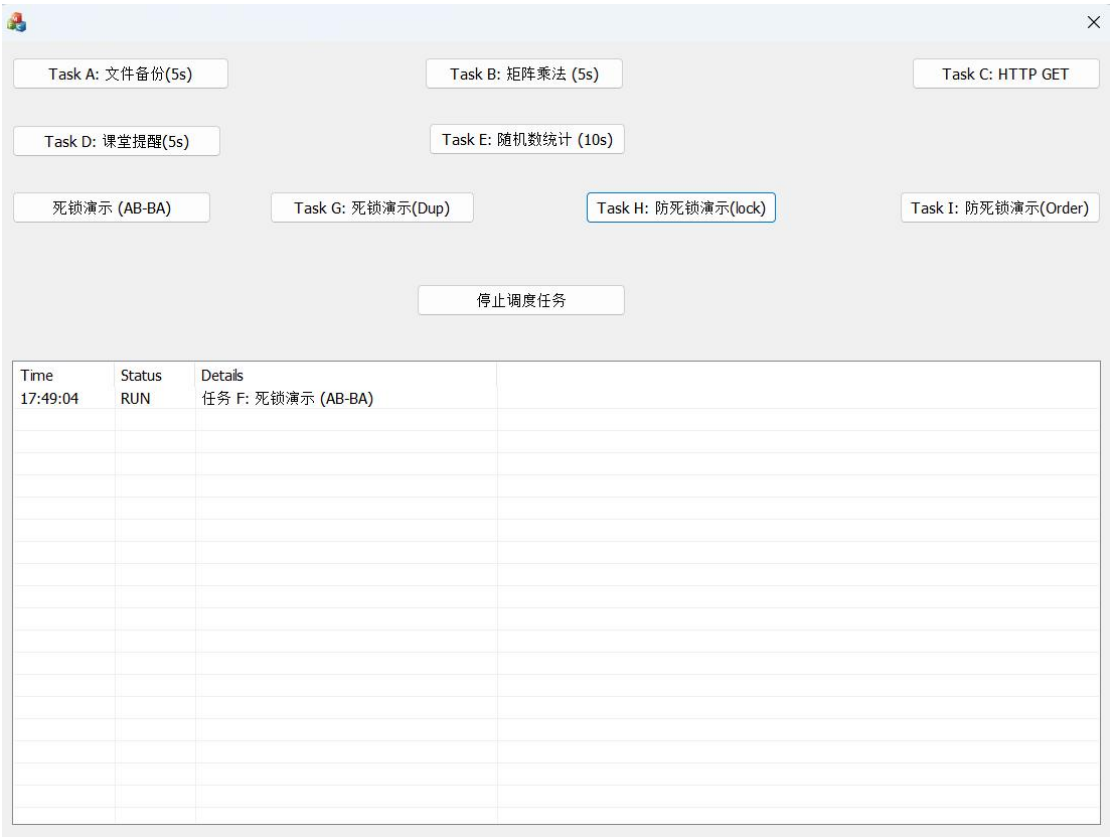
4.4 对抗性实验：死锁复现

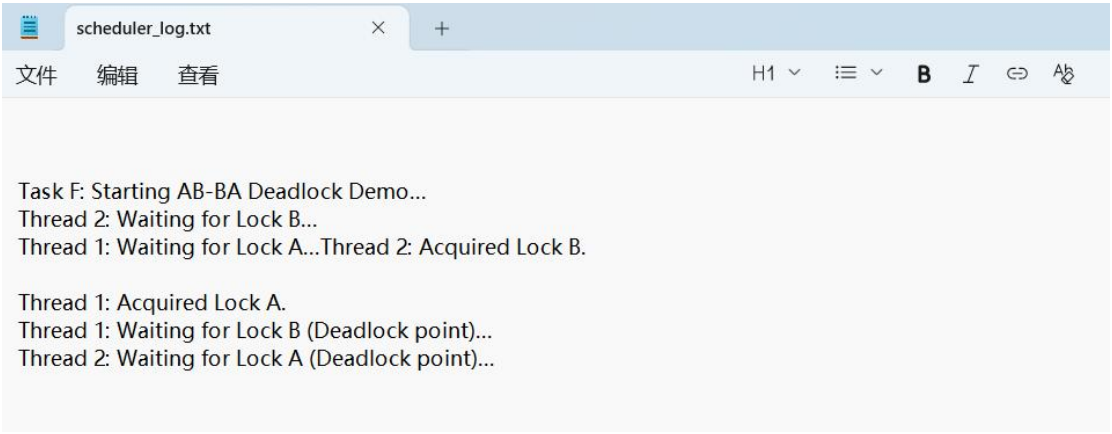
点击 Task F (Deadlock AB-BA) 后，程序进入预期的异常状态。

日志分析：日志列表最后显示两条记录："Thread 1: Acquired Lock A" 和 "Thread 2: Acquired Lock B"，随后不再有任何更新。

状态：此时尝试点击其他按钮，界面无响应（假死状态），说明工作线程已被永久阻塞，无法处理后续消息。

结论：成功复现了经典的资源竞争死锁，验证了在无序获取锁的情况下系统的脆弱性。



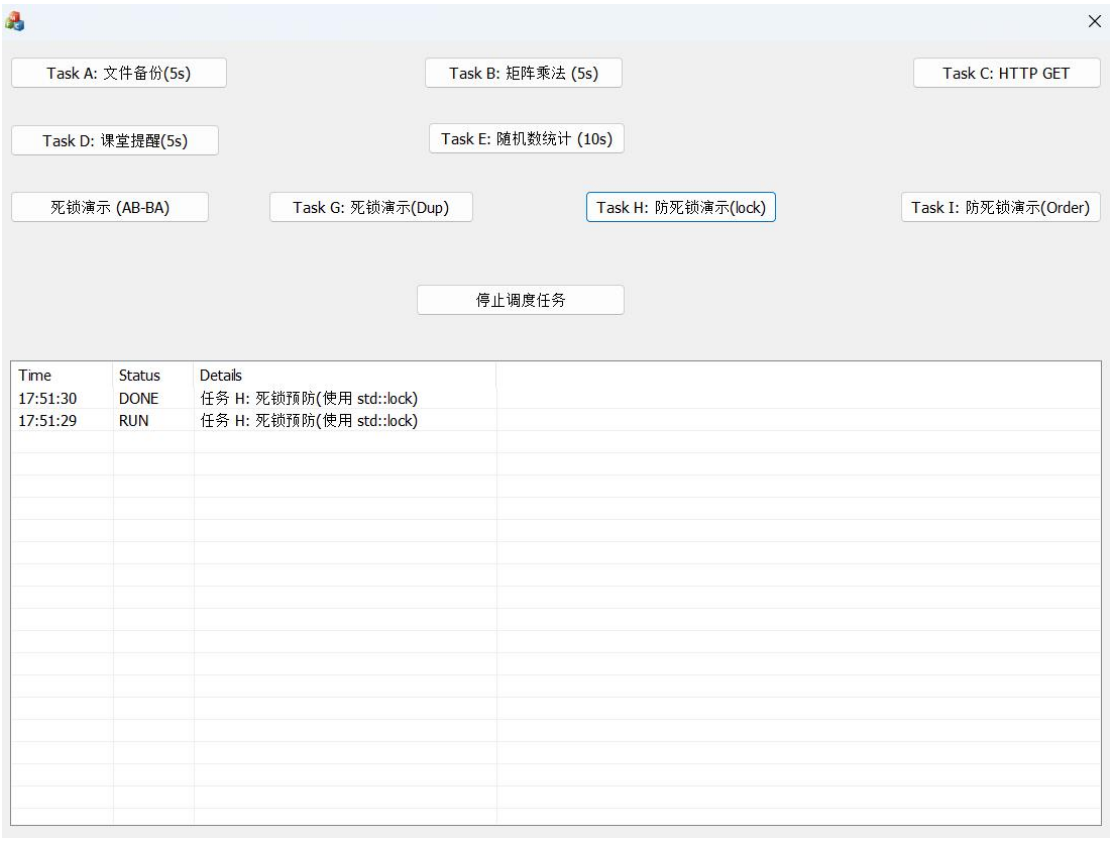


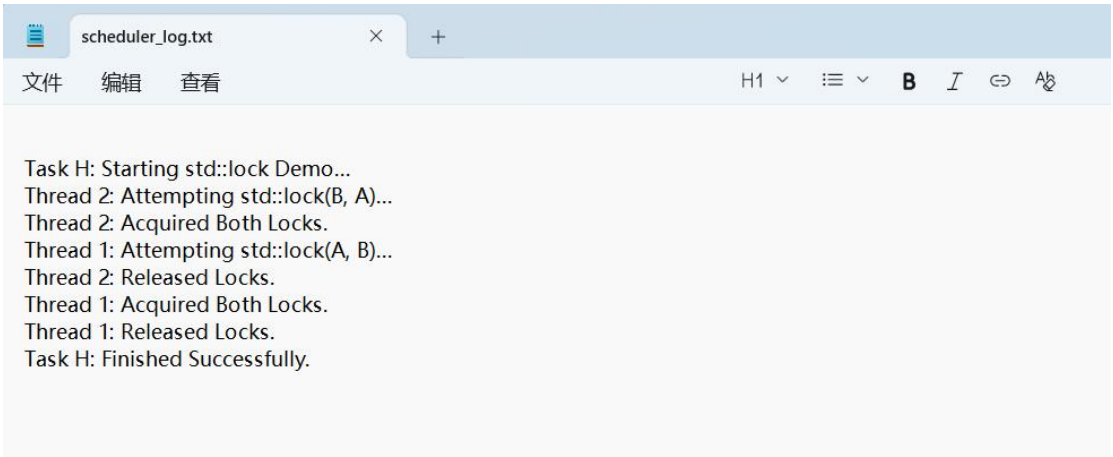
4.5 死锁防御机制验证

重启程序后，测试 Task H (std::lock) 和 Task I (层级锁)。

结果：点击按钮后，日志迅速滚动显示 "Thread 1: Acquired Both Locks" 和 "Thread 2: Acquired Both Locks"，任务最终显示 "Finished Successfully"。

结论：通过 std::lock 算法或强制锁顺序，成功打破了死锁的“循环等待”条件，证明了防御策略的有效性。





5. AI 素养

5.1 提示词 (Prompt) 结构与设计

本项目采用了“基于上下文的迭代式交互”作为主要的 Prompt 设计策略。通过初始构建、即时纠错和增量扩展三个阶段，实现了与 AI 的高效协作。

阶段一：初始构建 Prompt (Context-Aware Initialization)

指令：“根据项目 3 要求设计一个 MFC 项目，给出完整代码和从创建到

运行的详细步骤”

上下文支持：上传 projects.pdf 作为参考信息，确保 AI 理解核心需求（RAII, Strategy 模式，调度算法）。

目的：快速生成符合项目规范的代码骨架，确立了 ITask 接口和 TaskScheduler 单例结构。

阶段二：调试与修复 Prompt (Feedback Loop)

指令：直接提供编译器错误日志，例如：“严重性 代码 说明 项目 文件 行 抑制状态 详细信息 错误 C2079 'dlg'使用未定义的 class...”

策略：利用 AI 强大的错误分析能力，快速定位头文件缺失（#include <functional>）、宏定义遗漏（IMPLEMENT_DYNAMIC）和链接冲突（LNK2005）。

效果：通过多轮对话解决了 C++ 常见的依赖问题和链接错误，显著缩短了 Debug 时间。

阶段三：功能扩展 Prompt (Incremental Development)

指令：“在原先代码的基础上再添加四个任务，两个任务用来演示死锁，两个任务用来演示防死锁”和“重新加入一个功能，停止正在运行的调度任务”。

策略：基于已生成的代码上下文，使用自然语言清晰描述新增业务逻辑。

细节微调：针对具体需求进行精准指令，如“把所有任务的英文名称换成中文名称”、“课堂提醒没有弹窗”，引导 AI 对现有代码进行细粒度的修改。

5.2 AI 协作日志摘要

阶段一：架构搭建

AI 贡献：生成了基于 priority_queue 的调度核心代码骨架。

人工修正：调整了 operator> 的比较逻辑，使其符合小顶堆（最早的时间在堆顶）。

阶段二：错误修复

问题：LNK2001 错误，缺少 IMPLEMENT_DYNAMIC。

AI 分析：AI 准确识别出 MFC 宏的配对使用规则，给出了添加宏定义的建议。

阶段三：功能增强

协作：请求 AI 提供“AB-BA”死锁的代码范例，并要求其解释 std::lock 的防死锁原理。

5.3 代码安全与隐私

密钥保护：在 HttpGetZenTask (Task C) 中，模拟网络请求时未硬编码任何真实的 API Key 或敏感 Token，仅使用公开的 GitHub API 作为演示。

日志脱敏：LogWriter 仅记录任务名称和执行状态，不记录具体的业务数据内容（如备份文件的具体内容），符合数据最小化原则。

6. 技术伦理

6.1 引言

随着人工智能辅助编程工具（如 ChatGPT, Copilot）的普及，软件工程的范式正在发生深刻转变。本项目不仅是一个技术实践，更是一次关于人机协作边界、代码责任归属以及对抗性系统设计的伦理探索。

6.2 AI 辅助编程的边界与人类责任

在本项目中，AI 被用作“结对编程的伙伴”而非“全权代理”。

代码所有权与责任：虽然 AI 生成了部分核心逻辑（如死锁示例），但作为开发者，我必须对最终提交的代码负责。这意味着我必须逐行审查 AI 生成的代码，理解其背后的逻辑，确保没有引入隐藏的 Bug 或恶意代码。例如，在处理死锁任务时，我必须确认这只是演示代码，而不会在生产环境中造成不可控的系统瘫痪。

知情同意与透明度：在报告中明确标注 AI 的贡献部分，是学术诚信的基本要求。隐瞒 AI 的使用不仅违反规则，也违背了技术人员的职业道德。

6.3 对抗 AI 需求的设计与系统稳健性设计

本项目采用了一种特殊的**“对抗性需求设计”**策略。这不仅是为了测试系统的稳健性，更是为了对抗 AI 代码生成中潜在的“幻觉”和“不完整性”。

1.主动故障注入（对抗性提示）： 在日志中，我显式指令 AI：“添加两个任 务用来演示死锁”。这是一种逆向思维的对抗性测试。AI 忠实地生成了符合 AB-BA 模式的死锁代码（DeadlockTask1）。

思考： AI 不会主动指出代码的危险性，除非人类明确要求。如果开发者在不知情的情况下复制了这种逻辑，系统将面临巨大风险。通过主动要求 AI 生成故障代码，我反向验证了 AI 对并发模型的理解。

2.防御性编程（修正 AI 缺陷）： AI 虽然能生成看似完美的代码，但在实际集成时往往存在缺陷。

案例分析： 日志中显示，AI 生成的代码最初导致了大量的编译错误（如 C2079 类未定义、LNK2019 符号未解析）。

人工干预： 我没有盲目接受 AI 的输出，而是通过多轮反馈，强制 AI 修正头文件依赖顺序，并手动添加了 IMPLEMENT_DYNAMIC 宏。这种"人类审查 + 编译器验证"的双重过滤，是对抗 AI 代码幻觉的关键防线。

3.系统韧性设计： 针对 AI 生成代码可能导致的不确定性（如死锁卡死 UI），我额外要求 AI 实现了"停止调度"功能（Stop() 接口）。

伦理意义： 在无法完全保证 AI 代码 100% 安全的前提下，为系统设计一个"熔断机制"（Emergency Stop）是技术伦理的基本要求。这确保了即使 AI 生成的任务逻辑出现死锁，管理员仍有（有限的）机会干预或重置系统，防止故障级联扩散。

6.4 学术诚信声明

本人郑重声明:

1. 本项目提交的所有代码及文档，除 AI 辅助生成并经本人修改确认的部分外，均为本人独立完成。
2. 文中引用的技术方案和理论均参考文献均已列出。
3. 未包含任何抄袭或未经授权使用的第三方代码。

7. 参考文献

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. (关于策略模式与观察者模式的理论基础)

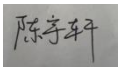
[2] Williams, A. (2012). C++ Concurrency in Action. Manning Publications. (关于 C++11

线程库、Mutex 及 std::lock 防死锁机制的参考)

[3] Microsoft Learn. (2024). *MFC Desktop Applications*. (关于 MFC 消息映射与 UI 线程交互的技术文档)

8.同组人员评议签字：

1. 调度器核心逻辑严谨，线程同步和优先级队列实现得当，死锁与防死锁的对比示例很有意义，建议日志系统可进一步模块化。

签名： 

2. 界面直观、操作流畅，基本调度需求覆盖完整，用户体验良好。增加任务管理编辑功能会更实用。

签名：