电梯调度系统设计方案报告

1. 项目背景及需求

背景：

电梯调度是操作系统进程调度思想的典型应用场景。本系统模拟多电梯协同工作的逻辑，通过动态分配请求优化资源利用率，体现调度算法在多线程环境下的作用。

需求：

- 支持自定义楼层数和电梯数（默认20层、5部电梯）。
- 每部电梯需包含数字键、开关门键、报警键，并实时显示状态（如楼层、方向）。
- 楼层外部按钮互联，按下后所有电梯同步响应。
- 调度算法需高效分配请求，减少等待时间。
- 初始状态所有电梯停靠在1层，无请求时保持静止。

2. 开发环境

语言：C++ **框架**：Qt 6 **工具**：Visual Studio2022 + Qt extension, Cmake **平台**：Windows

3. 项目整体结构图

```
ElevatorSystem（主界面）
├── SimulationMainWindow（模拟系统界面）
│   ├── ElevatorDisplayWindow（电梯监控窗口）
│   └── Elevator（单个电梯）
└── Utilities（通用枚举类）
```

4. 关键类的介绍

ElevatorSystem

职责：程序入口，管理初始参数（电梯数、楼层数），启动模拟窗口。 **关键方法**：

```C++
void BeginSimulation(); // 启动模拟窗口
```

SimulationMainWindow

职责：主界面管理，处理楼层按钮事件，分配外部请求，创建ElevatorDisplayWindow（电梯监控窗口）
关键成员：

```cpp
QButtonGroup* elevator_buttons; // 电梯按钮组(上，下)
std::vector<Elevator*> elevators; // 电梯对象列表
std::vector<FloorButtonState> floorButtonStates; // 楼层按钮状态
```

关键方法：

```cpp
void AssignExternalRequests(int floor, Direction dir); // 调度算法核心
void HandleFloorArrived(int floor, ElevatorState elevatorDirection);
```

Elevator

职责：单个电梯的状态管理（移动、开关门、报警），处理内部请求。 **关键成员**：

```cpp
int current_floor;                      // 当前楼层
std::set<int> internal_targets;         // 电梯内目标楼层
std::set<int> external_up_requests;     // 外部上行请求
std::set<int> external_down_requests;   // 外部下行请求
std::vector<QPushButton*> floorButtons; // 电梯内楼层按钮
ElevatorState state;                    // 当前状态（Idle/Up/Down等）
```

关键方法：

```cpp
void AddInternalTarget(int floor);                  // 电梯内目标
void AddExternalRequest(int floor, Direction dir);  // 电梯外请求
void DecideNextAction();                            // 决定移动方向
void MoveToNextFloor();                             // 逐层移动逻辑
void UpdateDisplay();                               // 更新电梯状态显示
```

ElevatorDisplayWindow

职责：显示所有电梯的实时状态（如楼层、门状态），是所有电梯窗口的父窗口

5. 调度算法设计（重点）

5.1 LOOK算法核心思想

`LOOK`算法是电梯调度的经典算法，其核心逻辑为：

1. 电梯向当前方向（上行/下行）移动，处理沿路的所有请求。
2. 当前方向无更多请求时，立即切换方向（无需移动到物理终点，如最高层或最底层）。

3. 动态调整移动范围，仅覆盖有请求的楼层。 在项目中，`LOOK`算法主要体现在`DecideNextAction()`和`MoveToNextFloor()`方法中，以下结合代码详细说明。

5.2 LOOK算法在代码中的实现

方向选择（`DecideNextAction`）

电梯通过分析内部目标和外部请求，动态决定移动方向：

```C++
// Elevator.cpp - DecideNextAction()
int up_min = INT_MAX, down_max = INT_MIN;

// 处理内部目标中的上行和下行请求
for (int f : internal_targets) {
    if (f > current_floor) up_min = std::min(up_min, f);
    if (f < current_floor) down_max = std::max(down_max, f);
}

// 处理外部上行请求中高于当前楼层的请求
for (int f : external_up_requests) {
    if (f > current_floor) up_min = std::min(up_min, f);
}

// 处理外部下行请求中高于当前楼层的请求（视为上行方向）
for (int f : external_down_requests) {
    if (f > current_floor) up_min = std::min(up_min, f);
}

// 类似逻辑处理下行方向...

// 根据最小上行或最大下行目标确定方向
if (up_min != INT_MAX) {
    direction = Direction::Up;
    state = ElevatorState::Up;
} else if (down_max != INT_MIN) {
    direction = Direction::Down;
    state = ElevatorState::Down;
}
```

逻辑解析：

1. 优先处理当前方向上的请求（如上行时，仅关注高于当前楼层的请求）。
2. 若当前方向无请求，立即切换方向（`LOOK`算法的核心特征）。

逐层移动与目标更新（`MoveToNextFloor`）

电梯按方向逐层移动，并在到达目标楼层时清除请求：

```C++
// Elevator.cpp - MoveToNextFloor()
if (direction == Direction::Up) {
    int min_above = INT_MAX;
    // 查找内部目标中最近的楼层
    for (int f : internal_targets) {
        if (f > current_floor) min_above = std::min(min_above, f);
    }
    // 查找外部上行请求中最近的楼层
    for (int f : external_up_requests) {
        if (f > current_floor) min_above = std::min(min_above, f);
    }
    // 若当前方向无请求，尝试从外部下行请求中查找（切换方向前的最后一次检查）
    if (min_above == INT_MAX) {
        for (int f : external_down_requests) {
            min_above = std::min(min_above, f);
        }
    }
    // 确定下一个目标楼层
    if (min_above != INT_MAX) next = min_above;
}
```

逻辑解析：

1. 电梯向当前方向移动，仅处理该方向上的请求。
2. 若当前方向无更多请求，尝试从反向请求中查找目标（体现`LOOK`的动态方向切换）。

5.3 请求合并与优先级

所有请求（内部按钮 + 外部按钮）被统一管理，确保去重和有序：

```C++
// Elevator.h
std::set<int> internal_targets;          // 内部目标（按键楼层）
std::set<int> external_up_requests;      // 外部上行请求
std::set<int> external_down_requests;    // 外部下行请求

// 添加请求时自动去重
void Elevator::AddInternalTarget(int floor) {
    if (internal_targets.insert(floor).second) {
        // 更新按钮状态
    }
}
```

5.4 状态机与定时器驱动

电梯通过状态机（`ElevatorState`）和定时器模拟多线程行为：

```C++
// Elevator.cpp - MoveToNextFloor() 的开门逻辑
if (stop) {
    emit FloorArrived(current_floor, state);
    state = ElevatorState::Opening;
    UpdateDisplay();
    m_openTimer = new QTimer(this);
    m_openTimer->singleShot(1000, [=]() {
        state = ElevatorState::Open;
        UpdateDisplay();
        m_stayOpenTimer->singleShot(2000, [=]() {
            state = ElevatorState::Closing;
            m_closeTimer->singleShot(1000, [=]() {
                DecideNextAction();  // 关门后重新决策方向
            });
        });
    });
}
```

逻辑解析：

1. 开门 → 停留 → 关门 → 重新决策方向，形成完整状态循环。
2. 定时器（QTimer）模拟时间流逝，避免阻塞主线程。

```
Idle → Up/Down → Opening → Open → Closing → Idle
```

5.5 调度算法优化点

方向切换策略： 当前代码在方向切换时可能重复遍历请求集合，可优化为预计算所有候选目标。

外部请求优先级： 外部请求（如高峰时段的上行请求）可加权处理，避免饥饿问题。

动态权重分配：

```C++
// 伪代码示例：根据请求时间动态调整优先级
void AssignExternalRequests(int floor, Direction dir) {
    int weight = (current_time - request_time) * 2; // 等待时间越长，权重越高
    best_elevator->AddExternalRequest(floor, dir, weight);
}
```

6. 多线程与事件处理

模拟多线程：通过Qt信号槽和定时器实现异步逻辑。例如，电梯移动使用`QTimer::singleShot`模拟逐层移动：

```C++
// Elevator.cpp
QTimer::singleShot(600, this, &Elevator::MoveToNextFloor);
```

线程安全：使用Qt的事件队列避免竞态条件，请求分配和状态更新通过信号传递。

7. 其他功能实现

报警功能:触发报警后，电梯暂停所有操作3秒

```C++
// Elevator.cpp
void Elevator::HandleAlarm() {
    ClearAllTimers();
    is_alarm_active = true;
    QTimer::singleShot(3000, [=]() { ... });
}
```

互联按钮:楼层按钮按下后，所有电梯同步记录外部请求

```C++
// SimulationMainWindow.cpp
connect(up_button, &QPushButton::clicked, this, [=]() {
    AssignExternalRequests(i, Direction::Up);
});
```