

**操作系统进程管理项目**

题目 电梯调度系统

姓 名 刘淑仪

学 号 2251730

学 院 软件学院

专 业 软件工程

教 师 张惠娟

二〇二四 年 五 月 六 日

**电梯调度系统**

刘淑仪 2251730 软件学院 软件工程

**一、项目简介**

**（一）项目目的**

1. 通过控制电梯调度，实现操作系统调度过程；

2. 学习特定环境下多线程编程方法

3. 学习调度算法。

**（二）项目需求**

某一层楼20层，有五部互联的电梯。基于线程思想，编写一个电梯调度程序。每部电梯有数字楼层键、关门键、开门键、上行键、下行键、报警键等；有数码显示器指示当前电梯状态；每层楼、每部电梯门口，有上行、下行按钮、数码显示。

**（三）项目开发环境**

系统：Windows 11 家庭中文版

IDE：PyCharm Community Edition 2023.3.3

Python 解释器：Python版本为3.8.16，且已通过pip安装了PyQt5

**二、项目架构**

1. 常量：用于存储程序中的常量便于调试修改，如电梯数量、电梯层数、上升下降时间、电梯开关门时间、所有按钮颜色、电梯的各种状态、电梯的移动状态、外部按钮产生的任务分配状态等。
2. 外部总体封装与程序入口：负责初始化变量、创建对象和启动线程。
3. 用户前端界面设计：在该部分中，实现了展示电梯的实时状态功能，允许用户通过点击电梯内外部楼层按钮发出上行和下行请求，并提供紧急制动功能，用户可通过警报键操作。此外，为确保界面实时更新，我们设计了定时刷新机制。
4. 电梯运行：负责根据任务队列控制电梯的运行状态（上行、下行或停止）以及开关门操作。
5. 外部多线程处理：负责接收和处理用户的上行和下行请求，通过寻找距离最近的电梯来分配任务。

**三、用户前端界面设计**

**（一）概述**

前端界面采用Qt库实现，使用外部工具QtDesigner作出前端界面后，再使用PyUIC将对应的.ui文件转换为.py文件，再进行代码优化以更适宜后续后端对接。

**（二）界面介绍**

**1.电梯实时状态显示区**

电梯实时状态显示由两个部分构成，分别是lcd数字显示屏显示当前楼层与ListWidget显示当前电梯运行状态。

1. lcd数字显示屏：通过数字显示屏显示当前楼层，左边配有上下箭头以直观显示电梯运行上下楼层。
2. ListWidget：显示电梯运行状态，如上行中、下行中、开门中、等待中、关门中以及故障。其中开门至关门使用不同颜色来提升用户观感（从绿变黄的过程表明电梯在开门-等待-关门），正常运行时为浅绿色，故障时为粉红色醒目提醒。

**2.楼层按钮区**

每台电梯都配有楼层按钮来选择想要到达的楼层。当点击对应按钮时，按钮会变为黄色。当到达指定楼层（即当前任务完成）时按钮状态会恢复。

**3.上下行按钮区**

除起始层和顶层分别只有上行键和下行键，每个楼层都配有上下行键。用户可以通过点击这些按钮发出上行和下行请求来调用最近的电梯响应。

**3.故障与恢复、开门关门按钮区域**

每台电梯均提供故障和恢复按钮，以便在电梯出现故障时进行模拟和处理。故障按钮可以用红色表示，再次点击即可恢复为正常。开关门可以延长或减少已经停靠在某楼层等待中的时间（不可再电梯上下行时按开关门按钮）

**（三）界面运行效果**

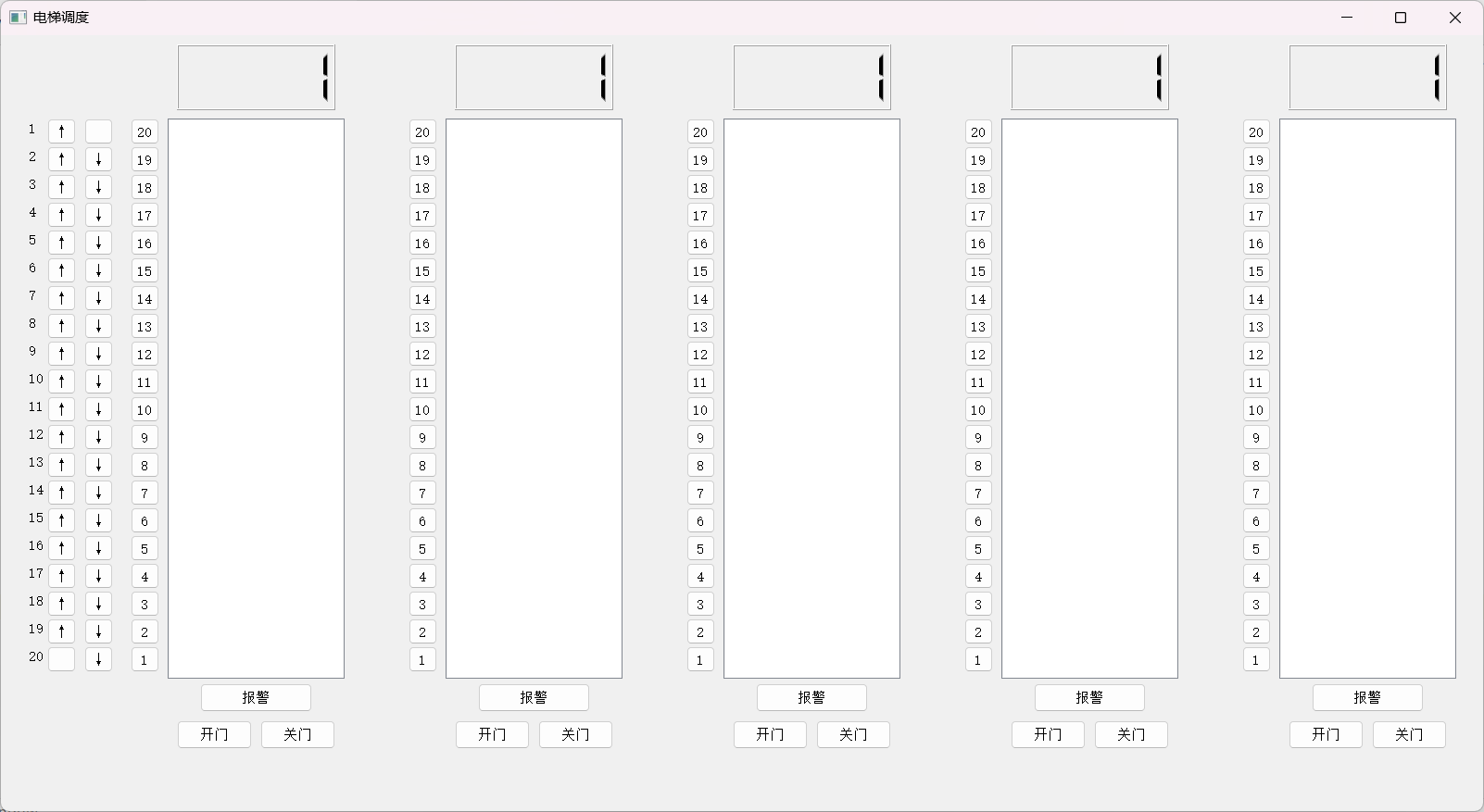


图1 前端示例图

**四、核心算法**

**（一）电梯调度算法概述**

**1.** **先来先服务算法（FCFS）**

该算法根据乘坐电梯的先后次序调度，虽然公平、简单，但是一旦载荷较大，就会严重增加等待时间。

**2.** **最短寻找楼层时间优先算法（SSTF）**

算法每次寻找可以最快满足的楼层，这样可以减少等待时间，但部分楼层（如顶楼和1楼）可能长时间无法响应，出现类似于操作系统中“饥饿”（starvation）的现象。

**3.** **扫描算法（SCAN）**

该算法使电梯在最底层和最顶层之间连续往返，所有方向相同的请求在一趟中完成，以确保总体等待时间的稳定。

**4.** **LOOK 算法**

该算法对扫描算法改进，当电梯发现移动方向上不再有目标时立即改变方向。

**（二）核心算法**

**1.** **主要使用算法**

本程序实现了如下功能：

* 外部请求通过扫描所有电梯，选择最适合的电梯。
* 电梯在运行过程中根据当前楼层和目标楼层之间的关系动态调整运行方向。
* 当一个方向上没有目标楼层时，电梯会改变方向，继续扫描另一方向上的目标楼层。

故本程序使用的是SCAN（电梯调度算法中的扫描算法）算法，但依然有些许不同，如Outer类里，find\_best\_elevator 方法只是选择了最适合的电梯，而没有直接涉及电梯的运行方向改变。它主要根据当前外部请求的楼层位置和电梯的当前状态选择最合适的电梯。而Elevator类里，run 方法中的电梯运行逻辑更接近 SCAN 算法，因为它在一个方向上扫描所有请求，直到该方向上没有请求为止，然后改变方向继续扫描。但是，它的具体实现中还包括了其他的细节，比如门的开关操作、故障处理等。

**2.** **类的主要功能**

1. Elevator类：该类继承自 QThread ，使得电梯可以作为一个独立的线程运行。它包含了多个方法，用于处理电梯的运行状态、楼层移动、门的开关以及故障处理。

· move\_one\_floor 方法：该方法控制电梯移动一层楼的过程，根据参数 move\_state 确定电梯的运行方向（向上或向下）。同时模拟电梯上升或下降的过程，通过时间片 time\_slice 进行模拟，每次循环表示经过一个时间片。在模拟运行过程中，通过检查电梯状态，如果出现故障则调用 trouble\_solving 方法处理故障。

· door\_operation 方法：该方法控制电梯门的开关操作，包括开门、等待和关门。通过时间片 time\_slice 模拟门的开关过程，同时更新门的开关进度条。当门的开关动作完成后，将电梯状态设置为正常状态。

· trouble\_solving 方法：该方法处理电梯出现故障的情况，包括将电梯状态设置为故障、清空电梯的目标楼层列表以及取消与电梯相关的未完成任务。

1. Outer类：该类继承自 QThread ，是一个外部任务处理线程。主要职责是处理电梯系统中的外部任务，即用户在楼层按上行或下行按钮时产生的任务，并将这些任务分配给最合适的电梯。

· find\_best\_elevator(outer\_task) 方法：该方法用于找到距离最近的电梯编号。它遍历所有电梯，跳过处于故障状态的电梯，并计算每个电梯到达外部任务所在楼层的距离。根据电梯的运行方向和当前任务所在楼层的位置，选择最合适的电梯。最后返回最合适的电梯编号。

·add\_task\_to\_queue(elevator\_num, out\_task, descending=False) 方法：该方法将任务加入到指定电梯的任务队列中。根据参数 descending 的取值，可以选择升序或降序方式将任务添加到队列中。添加任务后，将任务状态设为等待态。

1. MainWindow类：该类继承自 QtWidgets.QMainWindow ，用于创建一个用户界面，实现了电梯调度系统的功能。界面包括了电梯内部按钮、电梯楼层显示、电梯运行状态指示、外部上下楼请求按钮以及电梯报警按钮等组件。

· init\_ui\_elements 方法：初始化界面元素，包括电梯 LCD 数字显示、电梯箭头、电梯内外部按钮等。设置定时器，用于定时更新界面。调用 show 方法展示界面。

·elevator\_{}\_button\_clicked方法：处理电梯内部按钮的点击事件，包括将按钮点击信息加入任务列表中、更新按钮颜色等。中间的{}包含warning或direction，功能相似不多赘述。

· paint\_item 方法：用于展示电梯的运行状态，包括更新界面中电梯对应楼层的背景颜色和文字提示。

·update方法：用于定时更新界面，包括更新电梯 LCD 数字显示、电梯箭头、按钮颜色等。

**五、线程管理**

**（一）概述**

在该项目中，我们利用多线程处理各种任务，比如外部任务处理线程（Outer）和电梯运行线程（Elevator）。这样的设计下，线程同步变得至关重要，以防止数据竞争并维护系统的正确性。我们主要采用互斥锁（mutex）来实现线程同步。

互斥锁是一种同步原语，用于解决多线程中的临界区问题。当一个线程持有互斥锁时，其他线程必须等待，直到该锁被释放。通过使用互斥锁，我们可以确保在同一时刻只有一个线程能够访问和修改共享数据。

**（二）关键部分**

**1.Elevator线程**

主要的线程管理部分集中在run方法中。这个方法作为电梯线程的主循环，负责协调和执行电梯的运动、门操作和故障处理。该部分的核心功能如下：

互斥锁管理：通过在关键操作前后获取和释放互斥锁（mutex），确保对共享资源如电梯状态和当前楼层的访问是线程安全的。

状态检查与响应：循环中不断检查电梯的状态，根据电梯当前的状态（如故障、待命、移动等）和目标楼层列表决定执行的操作。

任务执行：包括移动电梯到新的楼层（move\_one\_floor）、进行门操作（door\_operation）和处理电梯故障（trouble\_solving）。

故障检测与处理：在任何关键操作中，如发现电梯处于故障状态，立即调用故障处理方法，确保电梯可以恢复或安全停止。

相关代码如下：

def run(self):  
 *"""  
 电梯运行线程  
 """* while True:  
 mutex.lock()  
 if elevator\_states[self.elevator\_num] == ELEVATOR\_STATE.FAULT:  
 self.trouble\_solving()  
 mutex.unlock()  
 continue  
  
 *# 向上扫描状态* if elevator\_move\_states[self.elevator\_num] == MOVE\_STATE.UP:  
 if elevator\_up\_target\_list[self.elevator\_num] != []:  
 *# 到层开门* if elevator\_up\_target\_list[self.elevator\_num][0] == elevator\_cur\_floor[self.elevator\_num]:  
 self.door\_operation()  
 *# 到达以后 把完成的任务删去(分为内外两方面)* for outer\_task in outer\_tasks\_list:  
 if outer\_task.floor == elevator\_cur\_floor[self.elevator\_num] :  
 outer\_task.task\_state = TASK\_STATE.FINISHED *# 交给outer处理* break  
  
 if elevator\_up\_target\_list != []:  
 elevator\_up\_target\_list[self.elevator\_num].pop(0)  
  
 elif elevator\_up\_target\_list[self.elevator\_num][0] > elevator\_cur\_floor[self.elevator\_num]:  
 self.move\_one\_floor(MOVE\_STATE.UP)  
 *# 当没有上行目标而出现下行目标时 更换状态* elif elevator\_up\_target\_list[self.elevator\_num] == [] and elevator\_down\_target\_list[  
 self.elevator\_num] != []:  
 elevator\_move\_states[self.elevator\_num] = MOVE\_STATE.DOWN  
  
 *# 向下扫描状态时(与上面一致)* elif elevator\_move\_states[self.elevator\_num] == MOVE\_STATE.DOWN:  
 if elevator\_down\_target\_list[self.elevator\_num] != []:  
 if elevator\_down\_target\_list[self.elevator\_num][0] == elevator\_cur\_floor[self.elevator\_num]:  
 self.door\_operation()  
 for outer\_task in outer\_tasks\_list:  
 if outer\_task.floor == elevator\_cur\_floor[self.elevator\_num]:  
 outer\_task.task\_state = TASK\_STATE.FINISHED  
 break  
 if elevator\_down\_target\_list != []:  
 elevator\_down\_target\_list[self.elevator\_num].pop(0)  
 elif elevator\_down\_target\_list[self.elevator\_num][0] < elevator\_cur\_floor[self.elevator\_num]:  
 self.move\_one\_floor(MOVE\_STATE.DOWN)  
 elif elevator\_down\_target\_list[self.elevator\_num] == [] and elevator\_up\_target\_list[  
 self.elevator\_num] != []:  
 elevator\_move\_states[self.elevator\_num] = MOVE\_STATE.UP  
  
 mutex.unlock()

**2.Outer线程**

在Outer类中的主要线程内容集中在run方法内，这是线程的主循环，负责不断处理和分配外部楼层请求到合适的电梯。

以下是该方法的功能和流程概述：

1. 循环监控和任务分配：

线程持续监控全局的外部任务列表outer\_tasks\_list，寻找还未被分配的任务（状态为TASK\_STATE.UNASSIGNED）。

2. 寻找最适合的电梯：

对于每个未分配的任务，线程调用find\_best\_elevator方法，这个方法通过比较所有可用电梯与请求楼层的距离和电梯状态，找出最佳的电梯来响应该请求。

3. 任务加入电梯队列：

一旦找到合适的电梯，根据电梯当前楼层与任务要求的楼层及移动方向，将任务通过add\_task\_to\_queue方法添加到电梯的上行或下行目标列表中。

如果任务楼层与电梯当前楼层相同但电梯的运动状态不满足任务要求，或者电梯尚未达到任务楼层，任务也会被添加到对应的队列。

4. 任务列表更新：

完成后，清除列表中已经标记为完成的任务（状态为TASK\_STATE.FINISHED），以维持列表的最新状态。

以下是该方法的线程安全和资源同步概述：

在操作共享资源如任务列表和电梯状态时，使用mutex互斥锁来避免多线程间的冲突，保证数据的一致性和线程安全。

相关代码如下：

def run(self):  
 *"""  
 电梯运行线程  
 """* while True:  
 mutex.lock()  
 if elevator\_states[self.elevator\_num] == ELEVATOR\_STATE.FAULT:  
 self.trouble\_solving()  
 mutex.unlock()  
 continue  
  
 *# 向上扫描状态* if elevator\_move\_states[self.elevator\_num] == MOVE\_STATE.UP:  
 if elevator\_up\_target\_list[self.elevator\_num] != []:  
 *# 到层开门* if elevator\_up\_target\_list[self.elevator\_num][0] == elevator\_cur\_floor[self.elevator\_num]:  
 self.door\_operation()  
 *# 到达以后 把完成的任务删去(分为内外两方面)* for outer\_task in outer\_tasks\_list:  
 if outer\_task.floor == elevator\_cur\_floor[self.elevator\_num] :  
 outer\_task.task\_state = TASK\_STATE.FINISHED *# 交给outer处理* break  
  
 if elevator\_up\_target\_list != []:  
 elevator\_up\_target\_list[self.elevator\_num].pop(0)  
  
 elif elevator\_up\_target\_list[self.elevator\_num][0] > elevator\_cur\_floor[self.elevator\_num]:  
 self.move\_one\_floor(MOVE\_STATE.UP)  
 *# 当没有上行目标而出现下行目标时 更换状态* elif elevator\_up\_target\_list[self.elevator\_num] == [] and elevator\_down\_target\_list[  
 self.elevator\_num] != []:  
 elevator\_move\_states[self.elevator\_num] = MOVE\_STATE.DOWN  
  
 *# 向下扫描状态时(与上面一致)* elif elevator\_move\_states[self.elevator\_num] == MOVE\_STATE.DOWN:  
 if elevator\_down\_target\_list[self.elevator\_num] != []:  
 if elevator\_down\_target\_list[self.elevator\_num][0] == elevator\_cur\_floor[self.elevator\_num]:  
 self.door\_operation()  
 for outer\_task in outer\_tasks\_list:  
 if outer\_task.floor == elevator\_cur\_floor[self.elevator\_num]:  
 outer\_task.task\_state = TASK\_STATE.FINISHED  
 break  
 if elevator\_down\_target\_list != []:  
 elevator\_down\_target\_list[self.elevator\_num].pop(0)  
 elif elevator\_down\_target\_list[self.elevator\_num][0] < elevator\_cur\_floor[self.elevator\_num]:  
 self.move\_one\_floor(MOVE\_STATE.DOWN)  
 elif elevator\_down\_target\_list[self.elevator\_num] == [] and elevator\_up\_target\_list[  
 self.elevator\_num] != []:  
 elevator\_move\_states[self.elevator\_num] = MOVE\_STATE.UP  
  
 mutex.unlock()

**3.MainWindow线程**

在MainWindow类中，线程管理主要涉及到界面更新和与后台逻辑（如电梯控制和任务调度）的交互。虽然这个类本身不创建新的线程（除了Qt本身的GUI线程），它与多线程环境中运行的电梯控制线程紧密交互，并采取了一些关键措施来确保线程安全和数据一致性。以下为具体内容概述：

1.互斥锁的使用：

全局互斥锁（mutex）：在修改或访问共享资源（如电梯的当前楼层、状态列表或任务列表）时，MainWindow通过使用互斥锁来保证线程安全。例如，在按钮事件处理函数中，它会锁定互斥锁以防止在更新电梯状态或任务队列时发生竞态条件。

2.事件处理与线程同步：

事件驱动的更新：MainWindow利用Qt的信号和槽机制处理用户事件（如按钮点击）。这些事件处理函数中使用互斥锁同步数据访问，以确保与电梯控制线程之间的数据一致性。

定时器的使用：通过QTimer设置一个定时任务来定期更新界面，如显示电梯的当前状态和位置。在update方法中，同样使用互斥锁来安全地访问共享数据。

3.用户界面与后台逻辑的协调：

界面响应与后台任务：界面通过响应用户操作（如按钮点击），生成任务（比如电梯调度请求）并加入到全局任务列表中。这些操作均在互斥锁的保护下进行，以同步前端的用户操作和后端的任务处理。

状态反馈：界面需要根据电梯的运行状态动态更新，如电梯位置、门的开关状态和电梯是否处于故障状态。这些状态的更新在update方法中处理，亦须确保线程间数据的一致性和同步。

**六、项目整体运行测试**

**（一）单个电梯上下行**

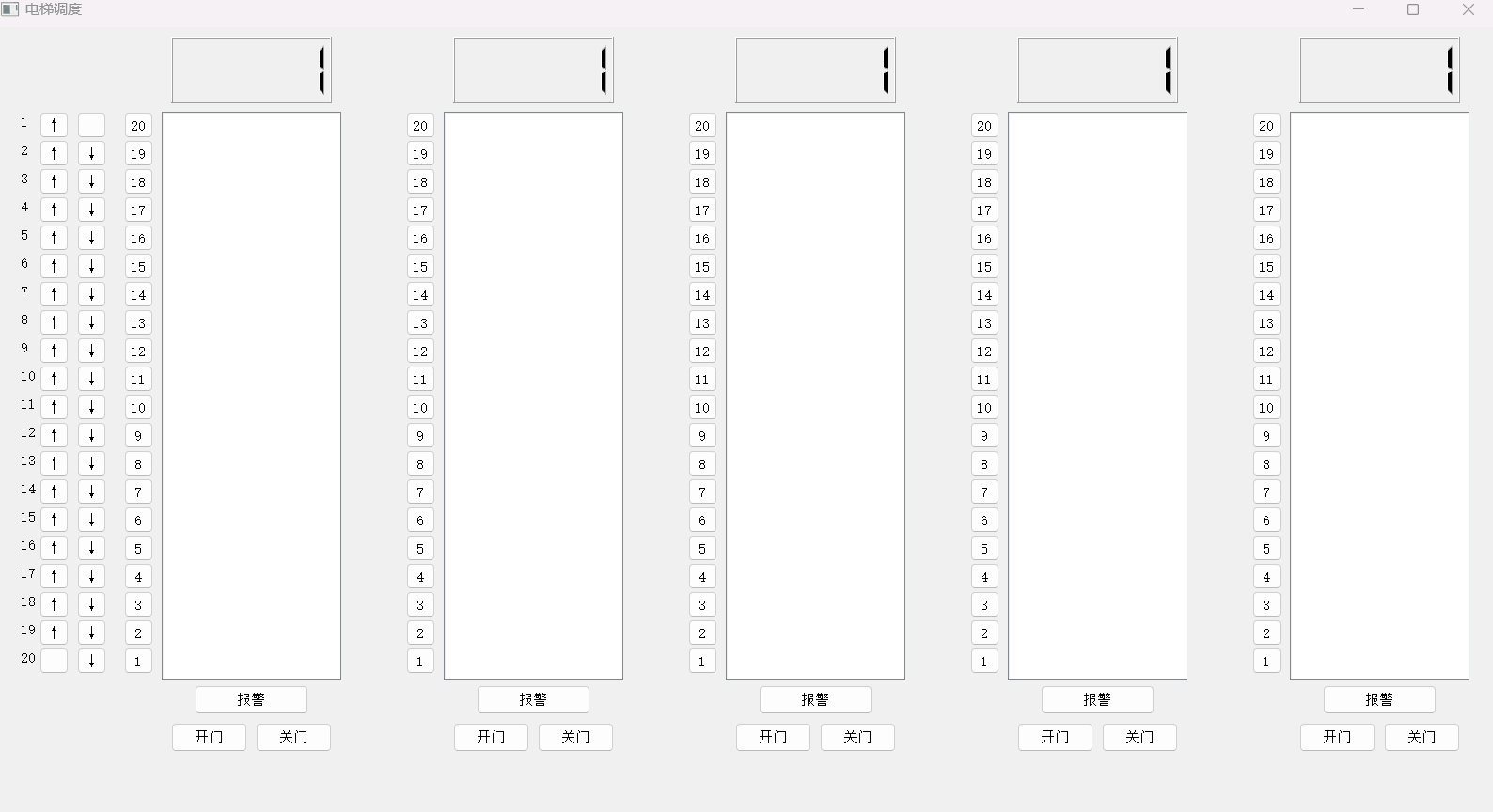


图2 单电梯上下行示例图

**（二）每个楼层上下行按钮**

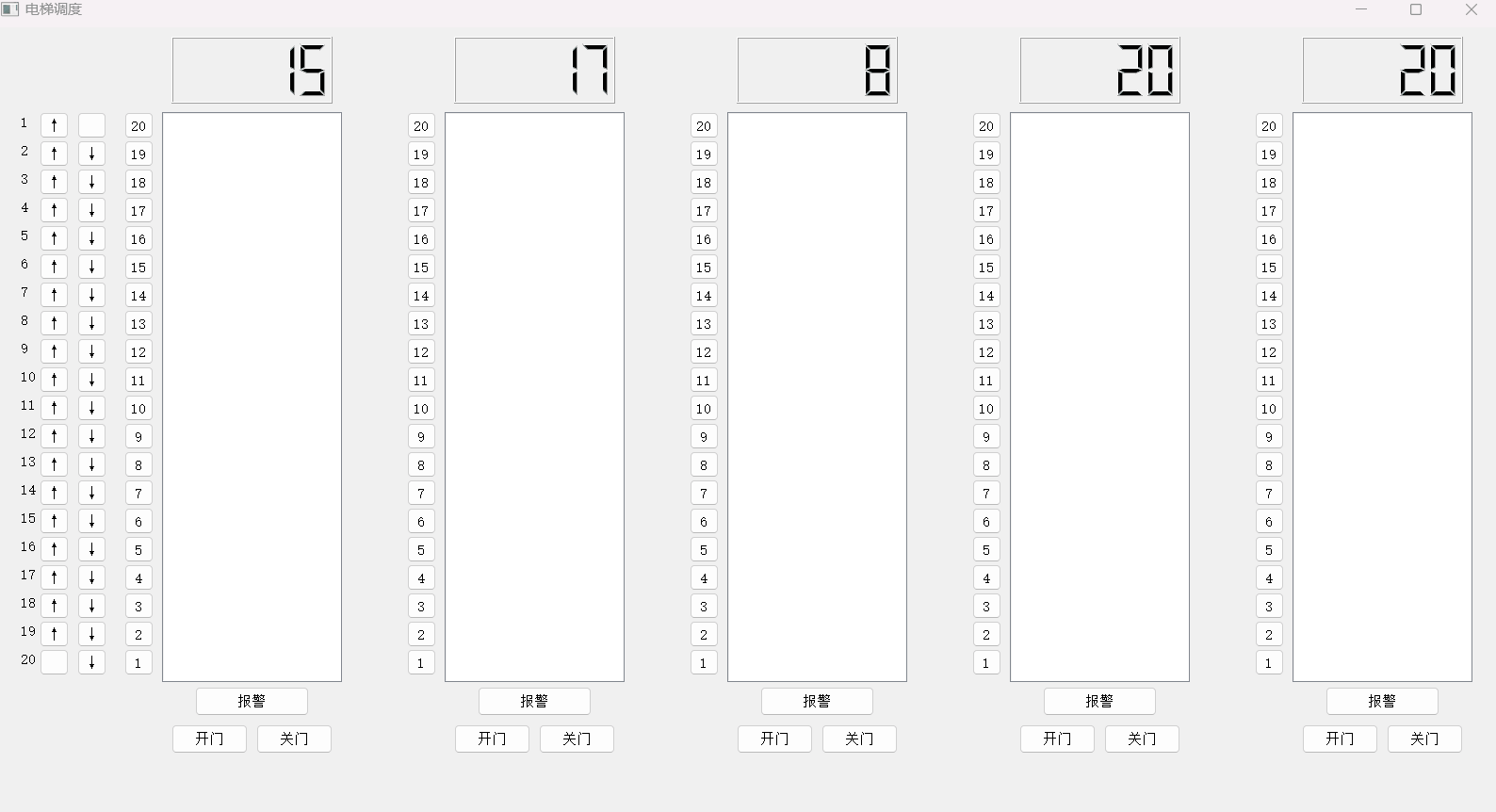


图3 每层楼上下行按钮示例图

**（三）报警按钮与开关门按钮**

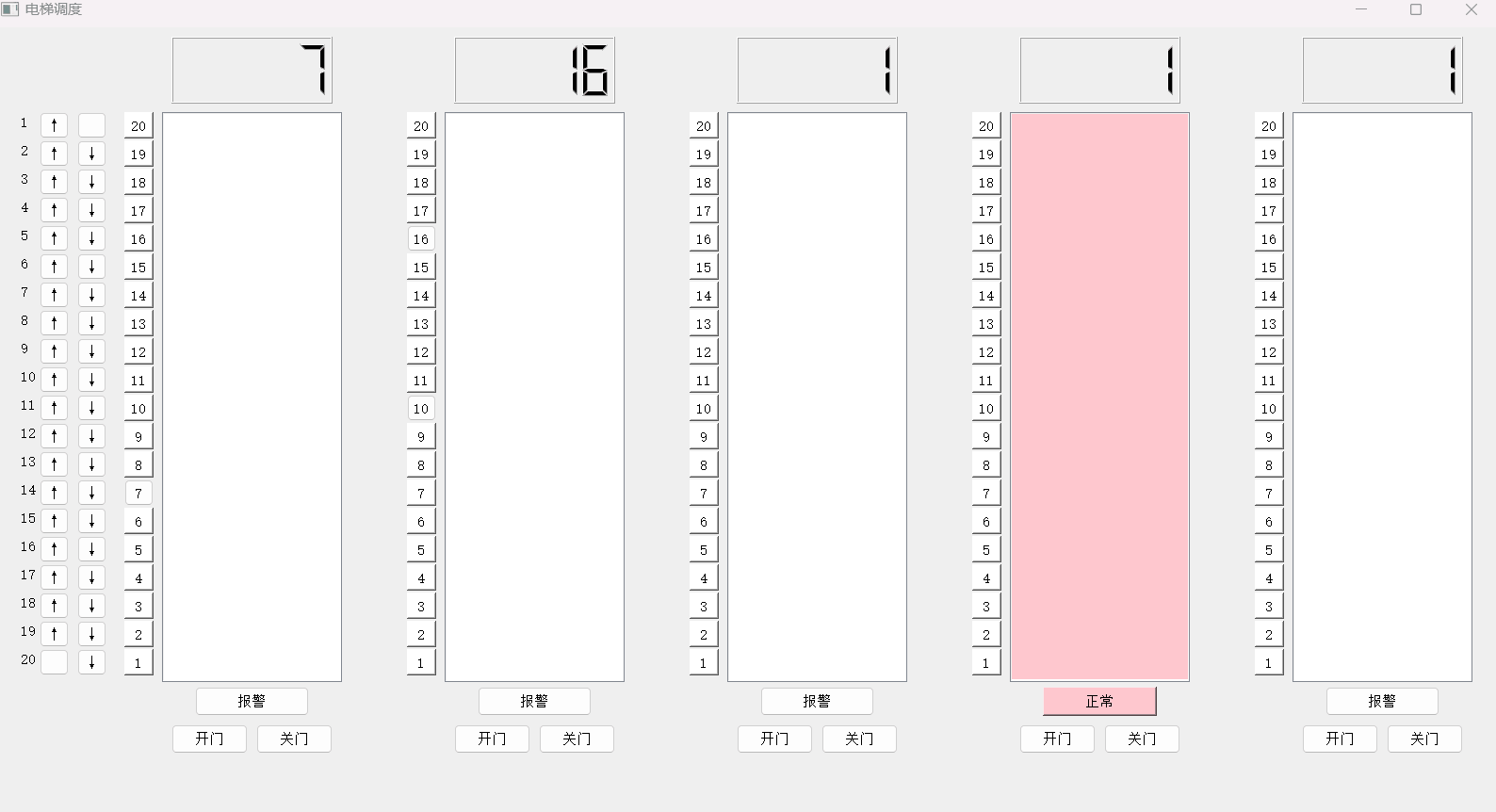


图4 报警与开关门按钮示例图