**操作系统课程项目文档**

**电梯调度**



实验名称 进程管理项目：电梯调度

姓名 刘震

学号 2352471

学院 计算机科学与技术学院

专业 软件工程

任课教师 张惠娟

日期 2025.5.11

1. **项目概览**
   1. **项目简介**

本项目为《操作系统》课程的多线程模拟实验，旨在通过实现一个基于图形用户界面（GUI）的电梯调度系统，深入理解操作系统中线程管理与调度算法的核心概念。项目采用 Python 3.11 作为主要开发语言，结合 PyQt5 框架进行界面开发，实现了模拟多线程环境下的任务分发、状态同步与用户交互等关键功能。

在模拟场景中，一栋20层大楼配备五部相互联动的电梯。每部电梯以独立线程运行，具备独立的状态控制逻辑，包括楼层移动、开关门、方向判断、请求响应等功能。系统统一管理所有外部请求（如每层楼的上下行按钮）与内部请求（电梯内部的数字按键），并通过设计电梯调度算法实现合理、高效的电梯任务分配。

**1.2 界面说明**



图1：项目可执行文件界面说明

**1.3 文件内容**

├── elevator\_simulation.exe # 程序可执行文件

├── Report.docx # 项目说明文档

├── src/

│ ├── main.py # 程序运行入口

│ ├── ui.py # 主界面逻辑

│ ├── base.py # 多线程类

│ └── dispatch.py # 调度类和算法

├── README.md # 项目运行说明

1. **类设计说明**

本项目采用面向对象的结构设计，将不同功能模块封装为多个类，主要包括用户界面类、调度控制类与电梯线程类。各类之间通过事件驱动与数据共享进行通信，体现出典型的多线程协调结构。

* 1. **用户界面类设计**

图形界面采用 ElevatorUI 类实现，继承自 PyQt5 的 QWidget，构建了完整的电梯调度可视化面板。该类负责显示每部电梯的当前楼层、状态信息以及响应用户的操作输入，如内部楼层按钮、外部上行/下行请求、开关门指令与报警控制等。

其核心方法 \_setup\_ui 用于初始化界面组件，包括数码管显示、电梯状态标签、按钮与日志面板等，并采用 QGridLayout 布局方式进行排布。\_start\_threads 方法则启动所有电梯的后台线程，并连接线程更新信号至 \_update\_ui 方法，用于根据电梯当前状态动态更新显示界面。

此外，ElevatorUI 包含多个事件处理方法，如 \_on\_internal 处理内部楼层选择， \_on\_external 处理外部上下楼请求， \_on\_open 和 \_on\_close 控制门开关， \_on\_alert 控制报警模式切换。界面交互操作通过 dispatcher 类与电梯线程联动，从而驱动电梯调度与响应逻辑。

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_ | 初始化界面、布局与线程，并绑定信号 |
| \_setup\_ui | 搭建 GUI 组件，包括按钮、显示器、日志区域等 |
| \_start\_threads | 启动每部电梯的线程并绑定信号更新 |
| \_update\_ui | 根据调度器状态刷新电梯的楼层与状态显示 |
| \_on\_internal | 响应电梯内部楼层按钮请求 |
| \_on\_external | 响应楼层外部上行/下行按钮请求，并分配电梯 |
| \_on\_open | 控制电梯开门（若电梯空闲且未报警） |
| \_on\_close | 控制电梯关门（若电梯空闲且未报警） |
| \_on\_alert | 切换报警状态并重新分配请求任务 |
| closeEvent | 程序关闭时安全终止所有电梯线程 |

表1：用户界面类设计函数作用

* 1. **调度类设计**

调度逻辑由 Dispatcher 类实现，是整个系统的核心调度模块。该类内部维护了五部电梯的当前楼层、目标楼层集合、运动状态、报警状态、开门状态等数据结构，并对所有调度请求进行处理与分发。

该类中的 assign\_internal 方法用于记录某部电梯的内部请求；assign\_external 方法用于处理楼层外部的上下行请求，并基于调度算法选择合适的电梯进行任务分配。该调度策略遵循三阶段优先规则：首先尝试选择方向相符且将经过该楼层的电梯，其次考虑就近的空闲电梯，最后退而选最近的任何方向电梯，实现了方向匹配与效率优先的结合。

update\_elevator 方法为模拟电梯每秒运行的逻辑执行入口，具体包括上下移动、判断是否需要开门、清除已完成请求以及更新电梯运动状态。电梯状态的更新由 \_update\_state 方法实现，根据当前楼层与剩余请求综合判断电梯应上行、下行或保持静止。

在突发状态处理方面，toggle\_alert 方法用于切换某电梯的报警状态，在进入报警模式时，会重新分配该电梯未完成的外部请求，以确保系统整体调度的健壮性和响应性。

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_ | 初始化电梯状态、请求队列与调度数据结构 |
| assign\_internal | 添加某部电梯的内部楼层请求 |
| assign\_external | 处理外部楼层请求并选择合适电梯执行 |
| update\_elevator | 模拟单部电梯的运行状态推进（移动、开关门、清除请求） |
| \_update\_state | 根据目标楼层与当前位置计算新的移动方向 |
| toggle\_alert | 切换电梯报警状态，并重新分配未完成请求 |

表2：调度类设计函数作用

* 1. **多线程类设计**

电梯运行过程由 ElevatorThread 类负责实现，该类继承自 QThread，用于创建每部电梯的后台线程，模拟其独立运行行为。每个线程通过调用 dispatcher.update\_elevator 周期性更新电梯的运行状态，并通过信号机制向界面发送更新指令，从而完成 UI 层的状态同步。

线程内部的主循环运行在 run 方法中，该方法持续执行直至接收到中断请求。在每一次循环中，线程负责推进电梯运行一步（移动楼层、判断开门、处理请求等），并通过 update\_signal 发送电梯编号用于界面刷新。同时加入 sleep(1) 以模拟真实电梯调度的节奏与响应延时。

线程的中断与释放通过 UI 的 closeEvent 方法统一管理，确保在程序退出时能安全终止所有线程，避免资源泄露与后台阻塞。

|  |  |
| --- | --- |
| **函数名** | **作用描述** |
| \_\_init\_\_ | 绑定调度器对象与电梯编号，初始化线程 |
| run | 周期性更新电梯状态并发送信号用于界面刷新 |

表3：多线程类设计函数作用

1. **调度算法说明**

本项目中设计的电梯调度算法模拟了操作系统中的线程调度策略。电梯线程的任务获取过程类似于操作系统调度器为可运行线程分配处理器时间片的过程。在操作系统中，线程调度强调响应速度、公平性与资源利用效率；在电梯系统中，则转换为响应请求及时、均衡电梯负载与减少等待时间。本项目通过 Dispatcher 类实现了调度逻辑，其中 assign\_external 与 assign\_internal 函数是核心调度算法的体现，负责处理用户请求与调度决策。

调度算法采用三阶段优先机制：首先优先考虑方向一致且将途经该楼层的电梯（模拟优先级匹配），若无合适电梯则考虑空闲电梯，最后退而选择离请求楼层最近的任意运行电梯（模拟最近就绪线程）。整个调度过程没有使用抢占策略，而是采用非抢占、动态指派调度策略，符合非实时操作系统中较温和的调度模型。

* 1. **assign\_internal 函数说明**

该函数用于处理电梯内部的楼层选择请求。用户在电梯内部按下某一楼层按钮时，将该请求加入该电梯的目标楼层集合中，即 targets 列表对应元素。若请求楼层与当前楼层一致，则不做处理。该函数不涉及调度选择，仅用于记录内部指令，由后续 update\_elevator 函数执行路径决策。

其本质等价于线程自身添加任务队列，体现了线程私有调度的概念。通过这种方式，内部请求与该线程（电梯）强绑定，避免共享资源竞争与调度冲突。

def assign\_internal(self, elevator\_id: int, floor: int):

"""

处理某部电梯的内部楼层请求。

elevator\_id：电梯编号（从1开始）

floor：用户在电梯内部选择的目标楼层

"""

idx = elevator\_id - 1 # 将电梯编号转换为从0开始的索引

# 若当前楼层不是目标楼层，添加到该电梯的目标集合中

if floor != self.floors[idx]:

self.targets[idx].add(floor)

# 返回该电梯编号（原样返回）

return elevator\_id

* 1. **assign\_external 函数说明**

该函数用于响应电梯外部的上行或下行请求，如某层用户按下“▲”或“▼”按钮时触发。函数逻辑分为三阶段依次尝试分配最合适的电梯：

第一阶段优先选择当前运行方向与请求方向一致且将经过请求楼层的电梯，模拟了调度器优先选择执行路径一致的线程，以减少上下文切换开销。第二阶段在前者不可行时，查找空闲状态的电梯，以实现资源利用最大化，模拟调度器选择空闲 CPU 分配任务的策略。第三阶段退而求其次，选择与请求方向无关但当前位置最近的任意电梯，保障请求不会长时间得不到响应，体现调度器的“最短响应时间优先”倾向。

无论在哪个阶段选中电梯，该函数都会将请求楼层加入该电梯的目标集合中，并加入全局 external\_requests 集合，用于 update\_elevator 时识别是否需要开门响应。

若在所有阶段均无可用电梯，则返回 -1，表明当前请求无法调度，相当于线程资源饱和时的任务拒绝策略。

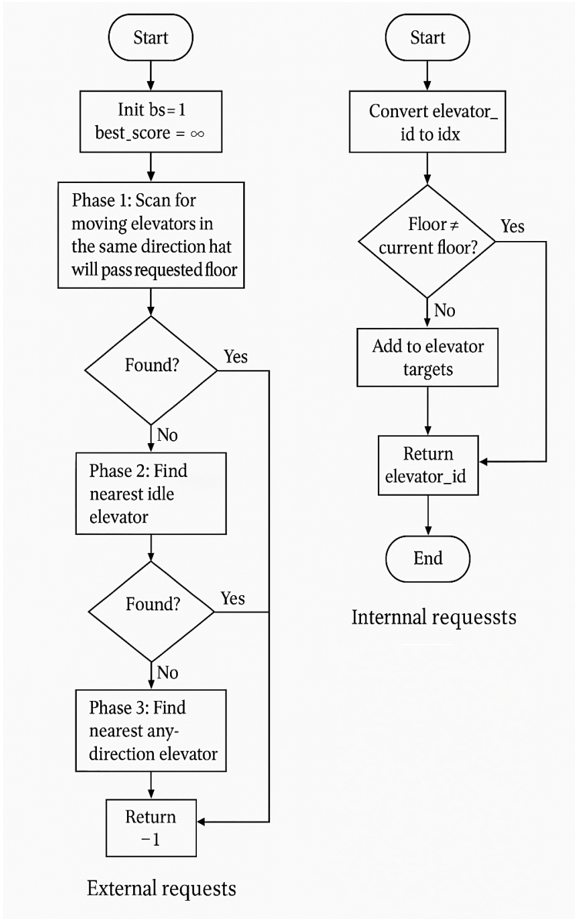


图2：assign\_external 和 assign\_internal 函数流程图演示

def assign\_external(self, floor: int, direction: str = None) -> int:

best\_idx = -1 # 最佳电梯索引初始化为无效值

best\_score = float('inf') # 最小距离初始化为无穷大

# -------- 阶段 1：查找方向一致、且会途经该楼层的电梯 --------

for idx in range(ELEVATOR\_NUM):

if self.alerts[idx]: # 如果该电梯处于报警状态，跳过

continue

# 判断电梯是否在该方向上运行且将经过请求楼层

if direction == 'up' and self.states[idx] == 1 and self.floors[idx] <= floor:

score = floor - self.floors[idx] # 计算“距离”

elif direction == 'down' and self.states[idx] == -1 and self.floors[idx] >= floor:

score = self.floors[idx] - floor

else:

continue # 不满足条件则跳过

# 如果该电梯比当前最佳电梯更近，则更新记录

if score < best\_score:

best\_score, best\_idx = score, idx

# -------- 阶段 2：查找空闲状态（idle）的电梯 --------

if best\_idx == -1: # 如果第一阶段未找到匹配电梯

idle\_score = float('inf')

for idx in range(ELEVATOR\_NUM):

if self.alerts[idx] or self.states[idx] != 0: # 跳过报警或非空闲电梯

continue

score = abs(self.floors[idx] - floor) # 计算与请求楼层的绝对距离

if score < idle\_score:

idle\_score, best\_idx = score, idx

# -------- 阶段 3：任意电梯中选择距离最近者 --------

if best\_idx == -1: # 若仍未分配到合适电梯

any\_score = float('inf')

for idx in range(ELEVATOR\_NUM):

if self.alerts[idx]: # 跳过报警中的电梯

continue

score = abs(self.floors[idx] - floor)

if score < any\_score:

any\_score, best\_idx = score, idx

# -------- 成功分配电梯后，记录该请求 --------

if best\_idx >= 0:

self.targets[best\_idx].add(floor) # 将目标楼层加入电梯目标集合

self.external\_requests.add(floor) # 记录到全局外部请求集合

if self.floors[best\_idx] == floor: # 若电梯刚好已在该楼层，直接开门

self.opens[best\_idx] = True

return best\_idx + 1 # 返回电梯编号（转为从1开始）

return -1

1. **项目总结**

本项目在设计上具有较高的灵活性与可调性，尤其体现在电梯调度算法的实现上。通过 assign\_external 函数中参数 direction 的传入与三阶段调度逻辑的设置，系统能够动态地调整调度优先级。开发者可以根据具体策略，控制方向优先、距离优先或空闲优先等因素的排序，从而实现具有策略适配能力的调度行为。这种结构不仅使得调度逻辑清晰易懂，也便于在未来扩展更多调度算法或权重控制机制。

项目在系统结构上保持了良好的模块划分，调度逻辑、线程管理与界面更新各自独立，线程之间通过信号实现低耦合通信，避免了数据竞争与共享冲突。每部电梯拥有独立运行状态、目标集合与开门逻辑，在并发运行中保证了状态一致性。整个系统具有较强的响应性与可维护性，能够正确处理多用户并发请求、开门/关门控制以及报警模式下的请求重分配。

系统仍存在一定的局限性。在当前实现中，外部请求一旦被某部电梯接管，其状态将不会再被其他电梯抢占，这可能导致个别请求长时间未被处理，从而出现类似“饥饿”问题。调度策略中未引入时间因子或请求等待权重，导致在极端负载下，调度公平性可能不足。此外，系统未对电梯容量、运行时间、上下行分离等真实调度要素进行建模，逻辑上仍属于理想化抽象模型。

项目所采用的调度策略在思路上与操作系统中的动态非抢占调度算法较为接近。电梯的请求队列与状态模拟了线程的就绪队列与执行状态，三阶段分配逻辑映射为操作系统中的方向优先匹配、空闲处理器利用与就近策略补偿。调度器并不强制打断正在运行的线程，而是根据状态与资源状况合理分派任务，这种温和式策略在非实时系统调度中具有代表性。同时，报警状态下的任务迁移，也体现了系统在异常情况下的请求回收与资源重分配机制，具备一定的容错性与恢复能力。

该项目有效地将操作系统的调度理论与图形化的电梯控制场景结合，通过直观、实时的界面反馈，使抽象的线程调度过程变得可视化、可交互，增强了对调度算法本质的理解与掌握。