处理机管理 – 电梯调度

2152402 段婷婷

1. 项目目的

- 学习调度算法
- 学习特定环境下多线程编程方法
- 通过实现电梯调度，体会操作系统调度过程

2. 项目简述与功能

- 某一栋楼有 20 层，有五部电梯。
- 每个**电梯内部**设置数字键，可以在电梯内部选择目标楼层；电梯内部设置显示屏，用于显示电梯所处楼层与运行状态（向上、向下或停止）
- **电梯外部**，每层楼有上行和下行按钮，可以按下按钮发出电梯请求。
- 当电梯外部的按钮被按下时，根据**电梯调度算法**选择响应该请求的电梯。
- 每部电梯都有**故障暂停键**，按下电梯就会停止运行，直到故障修复按下重启键。
- 所有电梯初始状态都在第一层。每个电梯如果在它的上层或者下层没有相应请求情况下，则在原地保持不动。
- 程序基于线程的思想，用 Qthread 实现多线程编程。

3. 项目设计

3.1 项目开发环境

- 系统：Windows 11 家庭中文版
- IDE：PyCharm 2022.3.3 (Community Edition)
- Python 解释器：通过 conda 部署 Python 环境，Python 版本为 3.8；通过 pip 安装 PyQt5

3.2 项目运行方式

- 直接运行：

  已经通过 pyinstaller 生成了 Windows 上的可执行文件。在 Windows 系统上点击 My_Elevator.exe，即可运行程序

  注意，My_Elevator.exe 一定要与 background.png，door_close.jpg，door_half.jpg，door_open.jpg 四张图片放在一个目录下。
- 编译运行
  - Python 版本：python 3.7
  - 安装 PtQt5（pip install PyQt5）
- 运行源码：

  进入源码所在目录，运行源码（python My_Elevator.py）

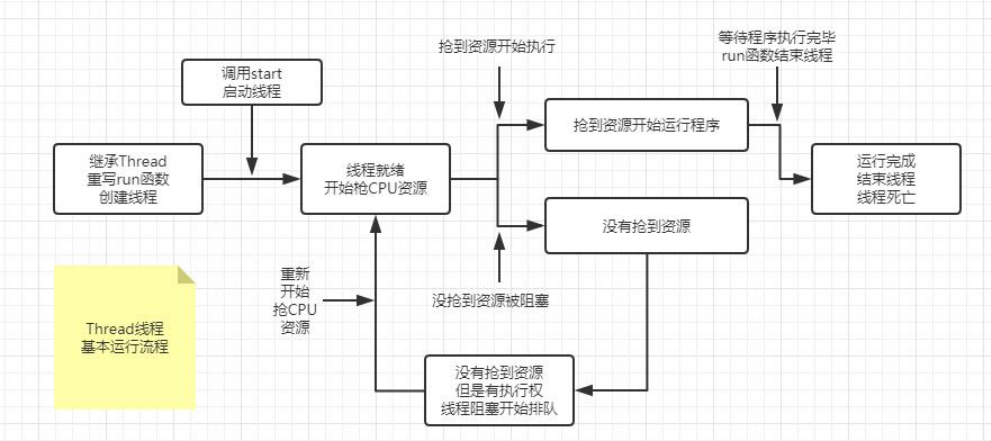
3.3 项目多线程

本项目使用 PyQt5 库中的 QThread 以实现多线程编程。下面是实现多线程的基本步骤：

- **创建自定义线程类：**

定义一个继承自 QThread 的类，该类将作为线程执行体。在此类中重写 run()方法，该方法包含线程的实际执行逻辑。

- **连接信号和槽：**
  为了在线程执行期间与主线程进行通信，使用信号和槽机制。定义自定义信号，然后在适当的地方发射该信号。
- **创建线程对象：**
  在主程序中，创建自定义线程类的实例。
- **启动线程：**
  调用线程对象的 start()方法来启动线程。
- **处理线程的结束：**
  如果需要在线程完成后执行一些操作，可以使用 finished 信号连接到相应的槽函数。



3.4 项目整体设计

- Elevator_Window 类：负责图形页面的定义与显示。
- Elevator_Thread 类：为电梯运行的线程，为每个电梯都创建该线程的实例，实现电梯的多线程运行。
- Elevator_Update 函数：更新电梯运行状态并对图形显示页面进行相应的更改，是控制电梯运行的关键函数。
- Set_Elevator_Goal_Internal 函数、Set_Elevator_Goal_External_Up 函数、Set_Elevator_Goal_External_Down 函数：当有按钮被按下时，根据调度算法选择响应该请求的电梯。
- 用全局变量记录电梯运行的状态、待处理的请求等。

4. 电梯调度算法

4.1 常见的电梯调度算法

- 先来先服务 (FCFS)：按照乘客请求的顺序进行服务。当有乘客请求时，电梯按照请求的先后顺序逐个服务。这种算法简单易实现，但可能导致不公平的等待时间和较长的响应时间。
- 最短寻找时间优先 (SSTF)：选择离当前楼层最近的乘客请求进行服务。电梯

在每次选择下一个目标楼层时，会考虑当前位置和乘客请求的位置，选择离当前楼层最近的目标楼层。这种算法可以减少等待时间和响应时间，但可能导致某些乘客请求长时间等待。

- 扫描算法（SCAN）：电梯在一个方向上连续服务乘客请求，直到该方向上没有更多请求，然后改变方向继续服务。这种算法模拟了电梯在一个方向上扫描楼层的行为。它可以减少乘客请求的平均等待时间，但可能导致某些乘客请求在电梯当前方向上长时间等待。
- 循环扫描算法（C-SCAN）：类似于扫描算法，但在达到最高或最低楼层后，电梯会立即返回到另一端，而不是改变方向。这种算法可以减少最远楼层的等待时间，但可能导致其他楼层的乘客等待时间增加。
- 最短时间预测（STP）：根据乘客请求的方向和目标楼层预测未来的请求，然后选择最佳的服务顺序。该算法基于对乘客行为的分析和预测，以提供最佳的服务顺序，以最小化总体等待时间。

4.2 本项目采用的电梯调度算法

本项目综合了上述算法的思想，平衡算法的复杂性、性能等因素，使用算法如下。

(1) 算法思想：① 当电梯外部有按钮被按下时，即时选择相应该请求的电梯并且加入该电梯的任务列表；② 电梯在一个方向上连续相应乘客请求，尽量少换方向，同时也可以避免 SSTF 导致的饥饿现象；③ 尽量使得多部电梯同时运行，避免某些电梯长期运行而另一些电梯长期闲置。

(2) 算法实现：

- 电梯内部按钮被按下，直接加入该电梯的任务列表。
- 电梯外部的按钮被按下时，按照下述优先原则选择相应请求的电梯：

  ① 未运行的电梯

  ② 正在运行，运行方向与请求方向一致、并且无需变换方向就能运行到请求楼层的电梯（例如，在 15 层按电梯向上的按钮，则选择正在向上运行，且电梯当前楼层不高于 15 层的电梯。）

  ③ 正在运行，运行方向与请求方向一致、但变换方向才能运行到请求楼层的电梯（例如，在 15 层按电梯向上的按钮，则选择正在向上运行，且电梯当前楼层高于 15 层的电梯。）

  ④ 正在运行，运行方向与请求方向相反、且需要变换方向才能运行到请求楼层的电梯（例如，在 15 层按电梯向上的按钮，则选择正在向下运行，且电梯当前楼层不高于 15 层的电梯。）

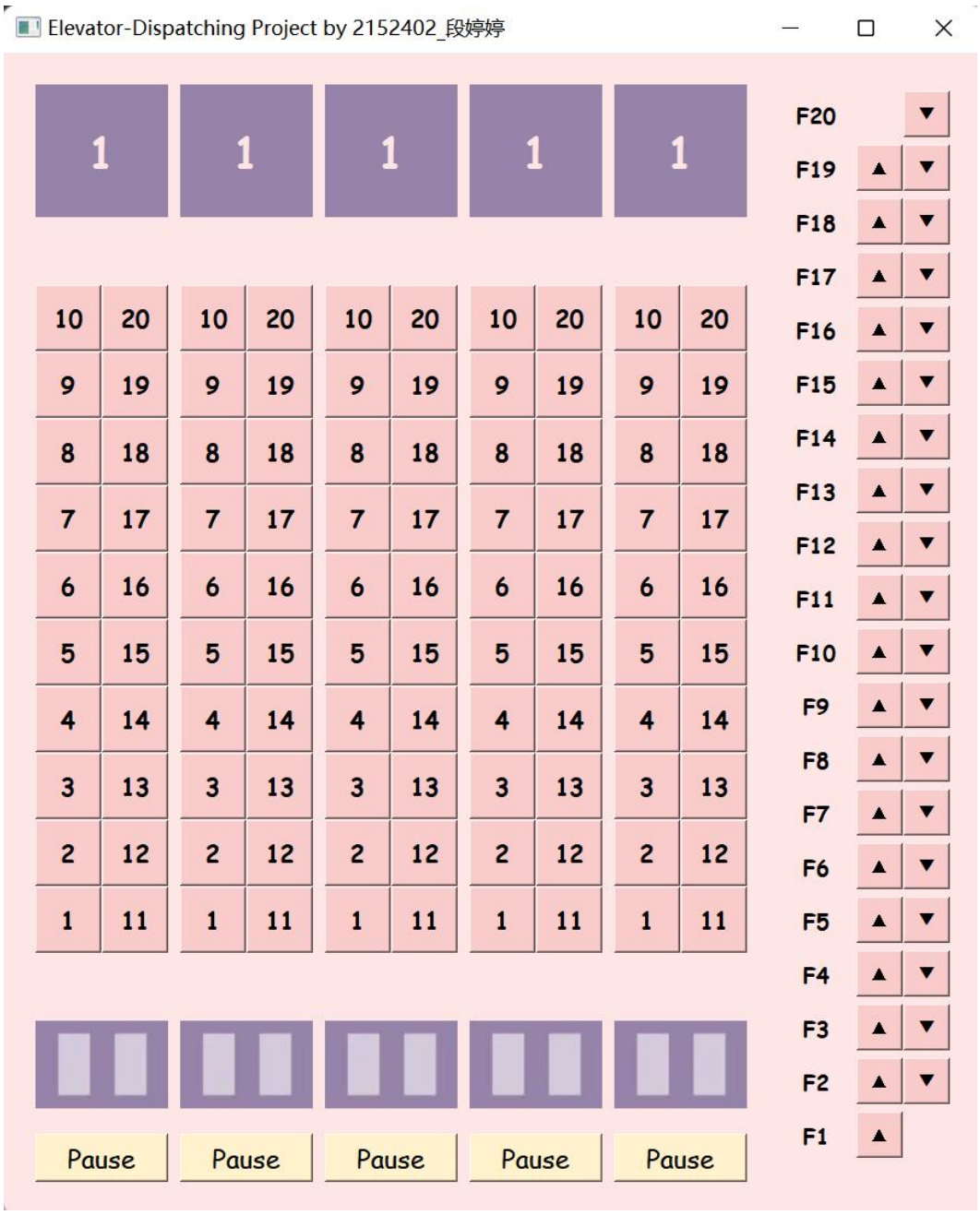
  ⑤ 正在运行，运行方向与请求方向相反、且无需变换方向就能运行到（注意并非无需变换方向就能相应）请求楼层的电梯（例如，在 15 层按电梯向上的按钮，则选择正在向下运行，且电梯当前楼层高于 15 层的电梯。）

(3) 算法合理性分析：

- ① 是基于尽量使得多部电梯同时运行的思想。
- ② 是在运行的电梯中，相应请求的最理想的电梯，可以理解为顺路就能接上该层乘客。
- ③④⑤ 的情况类似，都是电梯需要完成现有方向所有的任务后，变换方向，方能相应该请求。但是③④优于⑤，因为它们响应该请求所花的路程普遍更少（非绝对）。

5. 项目实现

5.1 图形化界面



总体来说分为两个部分：左侧分别显示 5 部电梯内部的按钮、屏幕、电梯门的情况；右侧显示电梯外部各楼层的电梯按钮，每层分别有一个向上和向下的电梯按钮（最底层和最高层除外）。

下面详细阐述左侧部分的各类图标：

① 左侧部分上方的紫色方形表示电梯内部的楼层显示屏，负责显示电梯所处的楼层和运行状态（向上、向下、不动）；

② 中间两个纵列的粉色按钮是电梯内部的楼层按钮，按下后会变成鹅黄色；

③ "Pause" 按钮为故障按钮，按下后电梯停止运行，按钮变为"Start"，按下 Start 按钮则电梯恢复运行；

④ 下方的紫色块表示门的状态，默认关闭，但当电梯到达目标楼层时，会显示开门与关门的动画。

5.2 全局变量

主要分为三类：图形化页面的实例；记录电梯任务列表的数组；记录电梯运行状态的数组。

5.2.1 图形化页面的实例

```
My_Elevator_Window = Elevator_Window()  # 图形化页面的实例
```

5.2.2 记录电梯任务列表的数组

```
elevator_goal_internal = []  # 电梯内部按下的目标楼层
elevator_goal_external_up = []  # 分配给该电梯的外部按下的向上的目标楼层
elevator_goal_external_down = []  # 分配给该电梯的外部按下的向下的目标楼层
elevator_goal_total = []  # 分配给该电梯的所有目标楼层
external_up = set([])  # 记录外面按下的所有向上电梯，这是为了避免重复按键导致某楼层被加入多个电梯的目标楼层
external_down = set([])  # 记录外面按下的所有向下电梯，这是为了避免重复按键导致某楼层被加入多个电梯的目标楼层
```

5.2.3 记录电梯运行状态的数组

```
state = []  # 电梯状态 0表示停止 1表示向上运行 -1表示向下运行
pause = []  # 电梯是否暂停运行
floor = []  # 当前楼层
open_door = []  # 电梯是否需要开门
```

5.3 类

5.3.1 Elevator_Window 类

该类继承至 QWidgts，是一个电梯调度系统的主窗口界面部分。它使用了 PyQt 库来创建窗口和布局，并包含了电梯内部和外部的按钮以及一些标签和样式设置。

主要的布局管理器是 whole_layout，它是一个水平布局，将窗口分为左右两个部分。左侧是电梯内部的布局管理器 grid_elevator_internal，右侧是电梯外部的布局管理器 grid_elevator_external。

在电梯内部的布局中，有一些 QLabel 用于显示当前楼层和电梯门状态，还有一些 QPushButton 用于表示电梯内部的楼层按钮和暂停按钮。这些按钮和标签都通过 grid_elevator_internal 布局管理器进行布局。

在电梯外部的布局中，有一些 QLabel 用于显示楼层号，以及一些 QPushButton 用于表示电梯外部的上行和下行按钮。这些按钮和标签都通过 grid_elevator_external 布局管理器进行布局。

最后，设置了窗口的样式、标题和位置，并显示出来。

特别说明的是，此类中也设置了各个按钮点击事件与相应处理函数的连接。

5.3.2 Elevator_Thread 类

该类继承自 QThread，用于在后台运行电梯的状态更新和门的开关动画。

该类定义了两个信号 update_signal 和 open_signal，分别用于更新电梯状态和控制门的开关动画。

在 run 方法中，通过循环不断发送 update_signal 信号来更新电梯状态。然后使用 time.sleep 方法暂停 1 秒钟，模拟电梯状态的更新间隔。

如果 open_door[self.int − 1]为 1，表示该电梯需要打开门。在这种情况下，通过发送 open_signal 信号来控制门的开关动画。

5.4 函数

5.4.1 Elevator_Update 函数

Elevator_Update 函数用于**更新电梯的状态、所在楼层、任务列表**并对图形页面做相应更改。按照下述顺序进行：

① 首先检查电梯是否被暂停，如果被暂停，则不需要更新状态，直接返回。

```
if pause[elevator_num] == 0:  #电梯被暂停了，不需要更新状态
    return
```

② 然后根据电梯的运行状态 state，更新电梯所在楼层 floor。

```
if state[elevator_num] == -1:  #电梯要向下运行
    floor[elevator_num] -= 1
elif state[elevator_num] == 1:  #电梯是向上运行的
    floor[elevator_num] += 1
```

③ 接下来是处理当前需要开门的情况：如过当前楼层位于电梯的任务列表，且运行方向与任务请求方向不矛盾，则开门，并且更新任务列表（取出已完成的任务）。

```
# 下面讨论当前需要开门的情况----------
if floor[elevator_num] in elevator_goal_internal[elevator_num]:  #现在电梯到达了一个内部按下的目标层
    open_door[elevator_num] = 1
    My_Elevator_Window.findChild(QPushButton, "{0}+{1}".format(elevator_num_1, floor[elevator_num])).setStyleSheet(
        "QPushButton{background-color: #FACBCB}") #移除电梯内部的标识
    elevator_goal_internal[elevator_num].discard(floor[elevator_num])

if elevator_goal_total[elevator_num]:# 如果有目标层才可能需要开门
#若电梯到达一个外部按下的目标层，而且方向一致(向上)
    if (state[elevator_num] == 1 or min(elevator_goal_total[elevator_num]) >= floor[elevator_num]) \
            and floor[elevator_num] in elevator_goal_external_up[elevator_num]:
        open_door[elevator_num] = 1
        My_Elevator_Window.findChild(QPushButton, "up{0}".format(floor[elevator_num])).setStyleSheet(
            "QPushButton{background-color: #FACBCB}")
        elevator_goal_external_up[elevator_num].discard(floor[elevator_num])
        external_up.discard(floor[elevator_num])
    # 若电梯到达一个外部按下的目标层，而且方向一致(向下)
    if (state[elevator_num] == -1 or max(elevator_goal_total[elevator_num]) <= floor[elevator_num]) \
            and floor[elevator_num] in elevator_goal_external_down[elevator_num]:
        open_door[elevator_num] = 1
        My_Elevator_Window.findChild(QPushButton, "down{0}".format(floor[elevator_num])).setStyleSheet(
            "QPushButton{background-color: #FACBCB}")
        elevator_goal_external_down[elevator_num].discard(floor[elevator_num])
        external_down.discard(floor[elevator_num])
```

```
# 若电梯是静止的，但是外部按下了该楼层，也是要开门的
if state[elevator_num] == 0:
    if floor[elevator_num] in elevator_goal_external_up[elevator_num]:
        open_door[elevator_num] = 1
        My_Elevator_Window.findChild(QPushButton, "up{0}".format(floor[elevator_num])).setStyleSheet(
            "QPushButton{background-color: #FACBCB}")
        elevator_goal_external_up[elevator_num].discard(floor[elevator_num])
        external_up.discard(floor[elevator_num])
    elif floor[elevator_num] in elevator_goal_external_down[elevator_num]:
        open_door[elevator_num] = 1
        My_Elevator_Window.findChild(QPushButton, "down{0}".format(floor[elevator_num])).setStyleSheet(
            "QPushButton{background-color: #FACBCB}")
        elevator_goal_external_down[elevator_num].discard(floor[elevator_num])
        external_down.discard(floor[elevator_num])
```

④根据当前状态和目标楼层列表的情况更新电梯的状态：如果当前运行方向上还有目标楼层，则不改变状态；否则，若反方向上有目标楼层，则改变运行方向；若两个方向都没有目标楼层，则置 state 为 0，表示停止。

```
# 下面更新电梯的状态state--------------------------------------------------------------------
if len(goal_list) == 0: #没有目标楼层了
    state[elevator_num] = 0
elif state[elevator_num] == 0: #当前是不动的，随便选一个目标楼层的方向移动，优先去内部按的
    to_floor = 1
    if len(goal_internal_list) != 0:
        to_floor = goal_internal_list[0]
    else:
        to_floor = goal_list[0]
    if to_floor > floor[elevator_num]:
        state[elevator_num] = 1
    else:
        state[elevator_num] = -1
elif state[elevator_num] == 1:# 目前是往上走，那么看看上面有没有楼层需要
    tag = 0
    for to_floor in goal_list:
        if to_floor > floor[elevator_num]:
            tag = 1
            break
    if tag == 0:
        state[elevator_num] = -1
else:
    tag = 0
    for to_floor in goal_list:
        if to_floor < floor[elevator_num]:
            tag = 1
            break
    if tag == 0:
        state[elevator_num] = 1
```

⑤ 最后，根据更新后的电梯状态和电梯楼层，更新电梯内部的显示屏内容。

```
label_floor = My_Elevator_Window.findChild(QLabel, "Floor{0}".format(elevator_num_1))

if state[elevator_num] == -1:  # 电梯要向下运行
    label_floor.setText("↓" + str(floor[elevator_num]))
elif state[elevator_num] == 1:  # 电梯是向上运行的
    label_floor.setText("↑" + str(floor[elevator_num]))
else:
    label_floor.setText(str(floor[elevator_num]))
```

5.4.2 Set_Elevator_Goal_Internal 函数

电梯内部按下按钮后，调用 Set_Elevator_Goal_Internal 函数来更新电梯的任务列表。

函数接受两个参数：elevator_num 表示电梯的编号，to_floor 表示要设定的目标楼层。

函数首先使用 findChild 方法找到对应电梯和目标楼层的按钮，并将按钮的样式修改为设定目标楼层的状态。然后，将目标楼层添加到电梯的内部目标楼层集合和总目标楼层集合中。

```
def Set_Elevator_Goal_Internal(elevator_num, to_floor):  # 设定电梯内部的目标楼层

    My_Elevator_Window.findChild(QPushButton, "{0}+{1}".format(elevator_num, to_floor)).setStyleSheet(
        "QPushButton{background-image: url(background.png)}")
    elevator_goal_internal[elevator_num - 1].add(to_floor)
    elevator_goal_total[elevator_num - 1].add(to_floor)
```

5.4.3 Set_Elevator_Goal_External_Up 函数

电梯外部按下上楼的按钮后，调用 Set_Elevator_Goal_External_Up 函数，根据电梯调度算法选择响应该请求的电梯并且更新该电梯的任务列表。

函数接受一个参数：to_floor 表示外部上楼请求所在的楼层。

① 检查是否已经存在相同的外部上楼请求，如果存在则直接返回。

② 将外部上楼请求楼层添加到外部上楼请求集合 external_up 中。

③ 使用 findChild 方法找到对应楼层的上行按钮，并将按钮的样式修改为表示该楼层有外部上楼请求的状态。

```
def Set_Elevator_Goal_External_Up(to_floor):  # 设定电梯外上楼请求所在的楼层

    if to_floor in external_up:
        return

    external_up.add(to_floor)

    My_Elevator_Window.findChild(QPushButton, "up{0}".format(to_floor)).setStyleSheet("QPushButton{background-image: url(background.png)}")
```

④ 根据前面讨论过的电梯调度算法，会选择一个电梯来响应该外部上楼请求。

```
# 下面是选择哪个电梯来响应这个请求的关键代码!
# 先看有没有空闲的电梯，有的话直接加上，然后return
for i in range(5):
    if state[i] == 0:
        elevator_goal_external_up[i].add(to_floor)
        elevator_goal_total[i].add(to_floor)
        return
# 看看有没有现在在 to_floor 以下的楼层，而且是向上走的电梯
for i in range(5):
    if state[i] == 1 and floor[i] <= to_floor:  #这里我是找到一个就加入了，没有再选择
        elevator_goal_external_up[i].add(to_floor)
        elevator_goal_total[i].add(to_floor)
        return
# 最后的话就直接选个在 to_floor 以下的楼层（那肯定是向下走的，显然选to_floor以下向下走的电梯比选to_floor以上向下走的电梯更优）
min_floor = min(floor)
lowest_elevator = floor.index(min_floor)

if min_floor <= to_floor:
    elevator_goal_external_up[lowest_elevator].add(to_floor)
    elevator_goal_total[lowest_elevator].add(to_floor)
    return
```

```
# 到这，说明所有的电梯都在to_floor上面，优先选往下走的；
for i in range(5):
    if(state[i] == -1):
        elevator_goal_external_up[i].add(to_floor)
        elevator_goal_total[i].add(to_floor)
        return

# 否则选楼层最高的
max_floor = max(floor)
highest_elevator = floor.index(max_floor)
elevator_goal_external_up[highest_elevator].add(to_floor)
elevator_goal_total[highest_elevator].add(to_floor)
```

5.4.4 Set_Elevator_Goal_External_Down 函数

电梯外部按下下楼的按钮后，调用 Set_Elevator_Goal_External_Down 函数，根据电梯调度算法选择响应该请求的电梯并且更新该电梯的任务列表。

函数内容与 Set_Elevator_Goal_External_Up 函数一致，此处不再赘述。

5.4.5 Elevator_Pause 函数

当图形界面的"Pause"或"Start"按钮被按下后，会调用 Elevator_Pause 函数来更新电梯的故障情况。

```
def Elevator_Pause(elevator_num):
    if pause[elevator_num - 1] == 0:
        pause[elevator_num - 1] = 1
        My_Elevator_Window.findChild(QPushButton, "pause{0}".format(elevator_num)).setText("Pause")
    else:
        pause[elevator_num - 1] = 0
        My_Elevator_Window.findChild(QPushButton, "pause{0}".format(elevator_num)).setText("Start")
```

5.4.6 Open_Door 函数

该函数根据不同的参数显示电梯门不同的状态图片，以实现电梯门开关的动画效果。

```
def Open_Door(type, elevator_num):
    if type == 0:
        My_Elevator_Window.findChild(QLabel, "open{0}".format(elevator_num)).setStyleSheet(
            "QLabel{background-image: url(door_half.jpg);background-position: center}")
    elif type == 1:
        My_Elevator_Window.findChild(QLabel, "open{0}".format(elevator_num)).setStyleSheet(
            "QLabel{background-image: url(door_open.jpg);background-position: center}")
    elif type == 2:
        My_Elevator_Window.findChild(QLabel, "open{0}".format(elevator_num)).setStyleSheet(
            "QLabel{background-image: url(door_close.jpg);background-position: center}")
```

6．运行界面展示

由截图可知：此时电梯 1 位于 6 层，正在向下运行，（**注意到虽然电梯 1 在 6 层，且电梯外部 6 层有按下的按钮，但是没有响应，这是因为电梯运行的方向与请求的方向不是一致的**）；电梯 2 处于 14 层，到达目标楼层，门打开，接下来准备向下运行；电梯 3 位于 13 层，正在向上运行；电梯 4 位于 17 层，到达目标楼层，门打开（此处截图时门处于半开状态），接下来准备向上运行；电梯 5 位于 16 层，正在向下运行。

7. 项目总结与心得

拿到这个项目，我的第一步是调研不同的电梯调度算法、各种语言以及框架的实现。考虑到 PyQt5 通过将 Qt 的功能与 Python 语言的灵活性和易用性相结合，为开发者提供了一种创建功能丰富、跨平台的 GUI 应用程序的便捷方式，我最终决定以 python 为编程语言，并以 PyQt5 库为框架来完成。这对于没有写过 python、没有用过任何框架写过前后端的我来说是极具挑战性的，因为这意味着从 0 到 1。在此项目的推动下，在几天时间内我入门 python 和 PyQt5，并且完成了项目。虽然还有颇多不足和待改进之处，但是我想我对自己的成果是满意的。

在完成项目的过程中，出现过的最困扰我最久的问题是界面中图标闪烁的问题。在尝试解决这个问题的过程中，走了很多弯路，但也有很多收获。一开始我觉得是多线程同时修改图形界面的冲突导致的，所以尝试在修改页面之前加锁。在反复修改的过程中，我对锁这一机制的理解更加深刻。最终解决问题的方法是：在线程执行期间使用信号与槽机制与主线程进行通信，通过发送信号在主线程中修改图形界面，来避免在多个线程中同时修改界面的情况，从而解决图标闪烁甚至消失的问题。由此我知道了，在多线程编程中，应该尽量避免在多线程中修改 GUI。

此项目多线程编程的实践，加深了我对线程这一概念的理解，也让我觉得自己不再是纸上谈兵。总而言之，我收获颇丰。