电梯调度系统

一、实验目的

本实验的目的是通过编写一个多线程电梯调度系统，来加深对多线程编程的了解和掌握，可列为以下几点：

* 通过控制电梯调度，实现操作系统调度过程；
* 学习特定环境下多线程编程方法
* 学习调度算法。

二、实验题目

* 基本任务

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。

* 功能描述
  + 电梯应有一些按键，如：数字键、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。
* 五部电梯相互联结，即当一个电梯按钮按下去时，其它电梯相应按钮同时点亮，表示也按下去了。
* 电梯调度算法；
  + 所有电梯初始状态都在第一层；
  + 每个电梯没有相应请求情况下，则应该在原地保持不动；
  + 电梯调度算法自行设计。

三、实验环境

Qt creator 的C++编译模式，具体的构建环境如下：



四、实验方案

\*系统概述

本电梯调度系统包含多个电梯和多个楼层。每个电梯有自己的状态，包括电梯当前所在楼层、电梯当前运行状态等。每个楼层都有一个电梯调度按钮，用于请求电梯服务。系统的主要功能如下：

>根据用户请求，选择最优的电梯进行调度。

>电梯按照请求的楼层进行运行，直到达到目标楼层。

>电梯内部有按钮，可以选择要去的楼层。

>当电梯到达目标楼层时，会停下并开门等待用户下电梯。

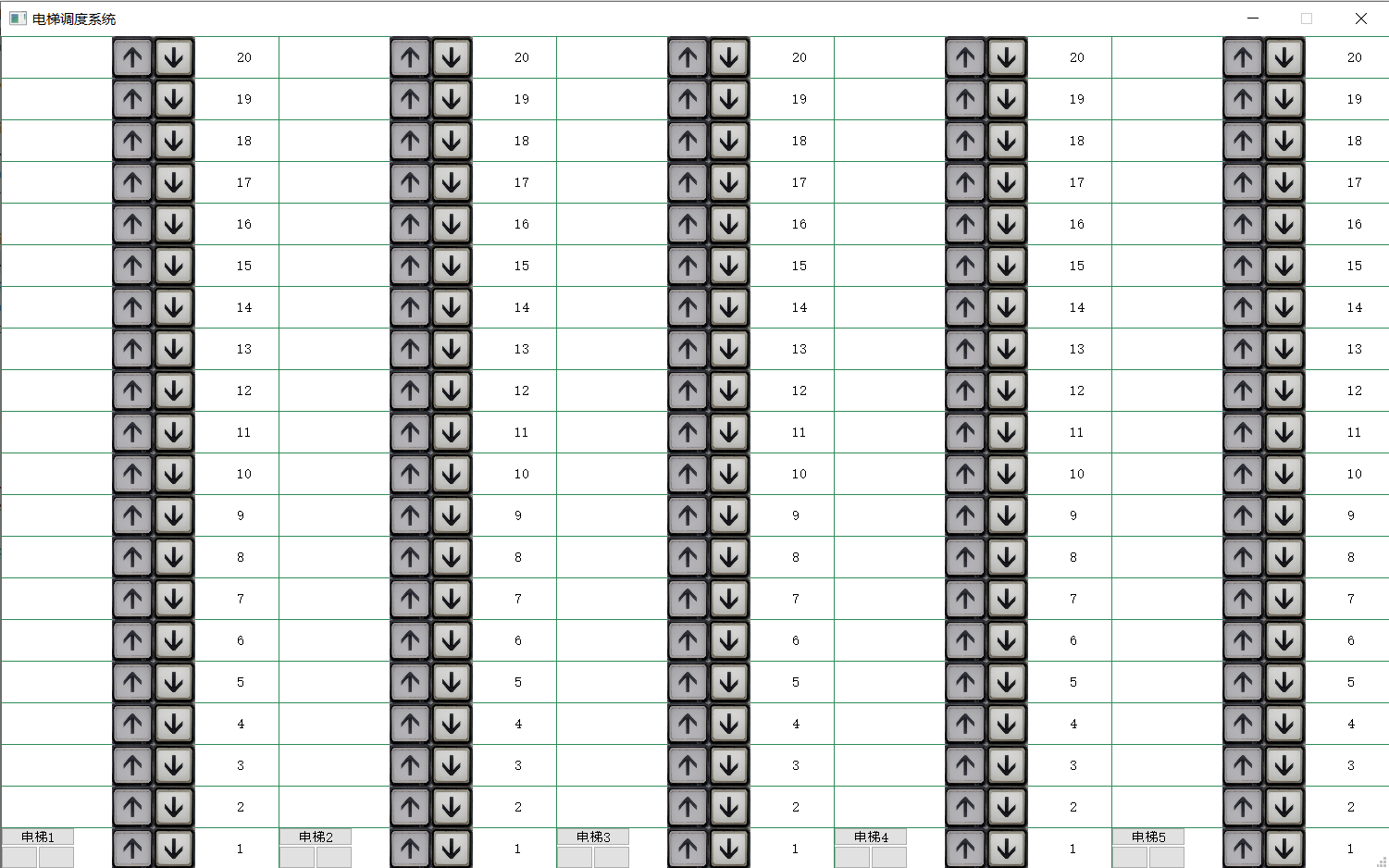
\*系统设计

本系统的设计主要包括以下几个方面：

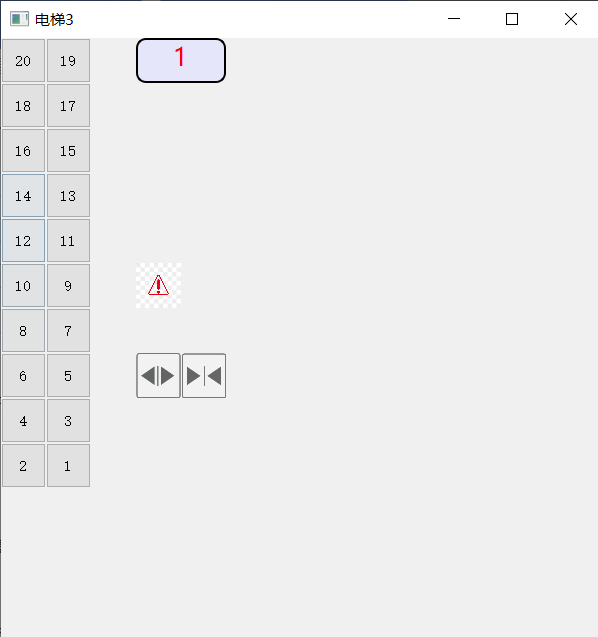
1. 界面设计

使用Qt框架的UI组件，通过纯代码、未采用QT Designer，设计电梯调度系统的图形界面。界面包括多个电梯和多个楼层，每个楼层有电梯调度按钮，用于请求电梯服务。

具体主界面和5部电梯的内部界面，程序启动所有界面同时打开，主界面如下，包括电梯和上行下行键，以及对应的楼层：



电梯的内部界面如下，包括电梯的数字键选择楼层，警报键，开关门按钮，以及一个楼层的数码显示器：



2. 电梯调度算法设计

电梯调度算法是本系统的核心。我们采用了基于电梯运动方向和距离的贪心算法，来选择最优的电梯进行调度。基本的思想是顺向截停，就近响应，处理完一个方向的请求，才会调转方向处理剩下的。

具体的算法如下：

>对于每个请求，遍历所有电梯，选择运动方向和请求方向相同的电梯。

>如果没有找到符合要求的电梯，则选择最近的电梯。

>如果有多个电梯都符合要求，则选择距离最近的电梯。

>如果有空闲的电梯，则所有的空闲电梯都添加该请求。

>创建了一个请求后，将请求排序。

>顺向处理新来的同向请求，将后来的请求但是更近的请求处理。

相关的核心代码如下：

//处理请求

void System::**HandleRequest**(const ElevatorRequest& request) {

//下面是按照规则的最优分配（也就是选择最优的电梯添加该请求（这里最优应该为时间最优,此处每层楼的运动时间相等就只计算距离））

int minDistance=MAXNUM;

int minTurn=0;

for(int i=0;i<elevators.length();i++){

if(elevators[i]->direction==request.direction&&elevators[i]->currentfloor==request.requestfloor)

{

elevators[i]->requests.append(request);

continue;

}

if(elevators[i]->direction==Direction::none){//当前空闲的电梯全部添加此请求

//添加进对应电梯的请求队列

elevators[i]->requests.append(request);

}

else if(elevators[i]->direction==Direction::down&&request.direction==Direction::down){//若是下行电梯下行请求

if(elevators[i]->currentfloor > request.requestfloor){

if(minDistance>qFabs(elevators[i]->currentfloor-request.requestfloor)){

minDistance=qFabs(elevators[i]->currentfloor-request.requestfloor);

minTurn=i;

}

}

else{//此时电梯需要先经历一次转向

int k=0;

for(k=0;k<elevators[i]->requests.length();k++)

if(elevators[i]->requests[k].direction!=Direction::down)

break;

int thistime=qFabs(elevators[i]->currentfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor))

+qFabs(request.requestfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor));

if(minDistance>thistime){

minDistance=thistime;

minTurn=i;

}

}

}

else if(elevators[i]->direction==Direction::up&&request.direction==Direction::up){//若是上行电梯上行请求

if(elevators[i]->currentfloor < request.requestfloor){

if(minDistance>qFabs(elevators[i]->currentfloor-request.requestfloor)){

minDistance=qFabs(elevators[i]->currentfloor-request.requestfloor);

minTurn=i;

}

}

else{//此时电梯需要先经历一次转向

int k=0;

for(k=0;k<elevators[i]->requests.length();k++)

if(elevators[i]->requests[k].direction!=Direction::up)

break;

int thistime=qFabs(elevators[i]->currentfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor))

+qFabs(request.requestfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor));

if(minDistance>thistime){

minDistance=thistime;

minTurn=i;

}

}

}

else if(elevators[i]->direction==Direction::up&&request.direction==Direction::down){//若是上行电梯下行请求

//此时电梯需要先经历一次转向

int k=0;

for(k=0;k<elevators[i]->requests.length();k++)

if(elevators[i]->requests[k].direction!=Direction::up)

break;

int thistime=qFabs(elevators[i]->currentfloor - (k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor))

+qFabs(request.requestfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor));

if(minDistance>thistime){

minDistance=thistime;

minTurn=i;

}

}

else if(elevators[i]->direction==Direction::down&&request.direction==Direction::up){//若是下行电梯上行请求

//此时电梯需要先经历一次转向

int k=0;

for(k=0;k<elevators[i]->requests.length();k++)

if(elevators[i]->requests[k].direction!=Direction::down)

break;

int thistime =qFabs(elevators[i]->currentfloor-(k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor))

+qFabs(request.requestfloor- (k>0?elevators[i]->requests[k-1].requestfloor:elevators[i]->targetfloor));

if(minDistance>thistime){

minDistance=thistime;

minTurn=i;

}

}

}

if(elevators[minTurn]->direction!=Direction::none){//电梯静止的情况已经处理过

//添加进对应电梯的请求队列

elevators[minTurn]->requests.append(request);

}

}

3. 多线程编程实现

在本系统中，每个电梯都是一个独立的线程。通过使用Qt框架中的QThread类，实现多线程编程。

每个电梯线程都有自己的状态和运行逻辑。当电梯被调度时，它会根据请求的楼层运行到目标楼层。当用户按下电梯内部的按钮时，电梯会根据请求的楼层运行到目标楼层。

每个楼层接收用户的请求，并将请求发送给调度线程。调度线程根据请求的楼层选择最优的电梯进行调度。

通过QT的信号和槽机制实现线程间的通信：子线程通过主线程间接的使用主线程资源，以及对QPushButton按钮的信号的处理。

4. 数据结构设计

为了方便电梯调度算法的实现，我们使用了一些数据结构来存储电梯和楼层的状态信息。其中，电梯的状态信息包括当前所在楼层、电梯当前运行方向、电梯内部的按钮状态等；楼层的状态信息包括当前是否有请求等。

下面是电梯类的代码作为数据结构的例子：

//电梯类对象

class Elevator :public QObject{

Q\_OBJECT

private:

//电梯编号

int id;

//当前楼层

int currentfloor;

//目标楼层

int targetfloor;

//方向（上行、下行、静止）

Direction direction;

//开关门

bool controldoor;

//报警键

bool warning;

//电梯内部界面

QWidget \*ElevatorPage;

//电梯是否结束运行

bool m\_quit;

//互斥锁

QMutex ele\_mutex;

// 请求队列

QVector<ElevatorRequest> requests;

//可视化UI组件

//20个楼层按钮/一个报警按钮/开关门按钮/上行按钮/下行按钮

QPushButton\* floorButton[FloorNum];

QPushButton\* alarmButton;

QPushButton\* closeButton;

QPushButton\* openButton;

//电梯主体

QPushButton\* upBorder;

QPushButton\* leftdoor;

QPushButton\* rightdoor;

friend class MainWindow;

friend class System;

friend class DigitalDisplay;

friend class RunEleThread;

public:

//构造函数，默认初始楼层为1，静止，关门，不报警

QWidget\* exUI;//创建指针指向主窗口

explicit Elevator(int id, QWidget\*p);

//数码显示器

DigitalDisplay\* digitaldisplay;

//电梯移动

void **Elemove**();

//电梯门控制

void **Closedoor**();

void **Opendoor**();

//报警

void **Warning**();

//绘制一个电梯

void **DrawElevator**();

//电梯内部

void **ElevatorPageShow**();

//处理请求

void **HandleRequestSingle**(const ElevatorRequest& mid);

//处理电梯调度的函数声明

void **\_Sort\_Request**();

//处理来自子线程的资源请求,启动电梯运行

void **Run**();

//完成请求后的处理

void **HandleReached**();

signals:

//开始处理请求

void **startElemove**(Elevator\* ele);

//完成请求后的操作

void **reached**(Elevator\* ele);

};

5. 功能

>电梯一般的实际需要。

>每层楼的上下楼请求。

>按钮等待电梯到达后显示的变化，电梯到达目标楼层自动开关门（由于线程阻塞为增添开门和关门间的时间延迟）。

>退出程序的exit按钮。

>关闭电梯页面，点击电梯的上标重新开启电梯内部页面。

6. 改进

>算法改进，本次采用最基本的顺向截停算法，可以改进算法最短寻找楼层时间优先算法(SSTF)、扫描算法(SCAN)、LOOK 算法等等。

>添加一个信号与槽函数，增添电梯到达楼层完成一个请求，增添到达提示音。这里因为技术操作重复，省去。

>由于多线程的资源竞争，五部电梯能够同时的上下移动，但不能同时开关门。

实验步骤

本实验的步骤如下：

>实现数据结构，用于存储电梯和楼层的状态信息。

>使用Qt框架的UI组件，设计电梯调度系统的图形界面。

>实现电梯调度算法，并将其集成到系统中。

>使用QThread类实现多线程编程，将每个电梯作为一个独立的线程。

>编写测试代码，对系统进行测试，并进行调试。

实验结果

经过实验，我们成功实现了一个多线程电梯调度系统。系统具有如下特点：

>可以根据用户请求，选择最优的电梯进行调度。

>电梯按照请求的楼层进行运行，直到达到目标楼层。

>电梯内部有按钮，可以选择要去的楼层。

>当电梯到达目标楼层时，会停下并开门等待用户下电梯。

实验总结

本实验通过使用Qt和C++编写一个多线程电梯调度系统，加深了对多线程编程和Qt框架的了解和掌握。在实验过程中，我们学习了如何使用Qt框架的UI设计工具，如何使用QThread类实现多线程编程，以及如何设计和实现电梯调度算法。

通过本实验，我不仅掌握了多线程编程的基本原理和技术，还提高了对Qt框架的熟练度。同时，我也认识到了多线程编程中可能出现的一些问题，如资源竞争、死锁等，需要在编程中予以避免和解决。