Operating Systems* Lab *II* LaTeX

* Teacher: Shuyu Shi. TA: Gravity

1st 张逸凯 171840708 (转专业到计科, 非重修)

Department of Computer Science and Technology

Nanjing University

zykhelloha@gmail.com

实验任务/需求
开发环境
程序说明及功能展示
进程与线程的通信、同步技术
    不同楼层发送电梯请求的资源互斥访问机制
        电梯作为临界资源定义
        状态变量作为信号量定义
    遵循的准则
        空闲让进
        忙则等待
        有限等待
        让权等待
    证明不会产生死锁
调度算法实现
    期望最邻近算法
    优先级调度算法:
    全响应调度算法
    随机调度算法
    电梯内按键调度
遇到的问题及解决方法
验收实验
总结💯

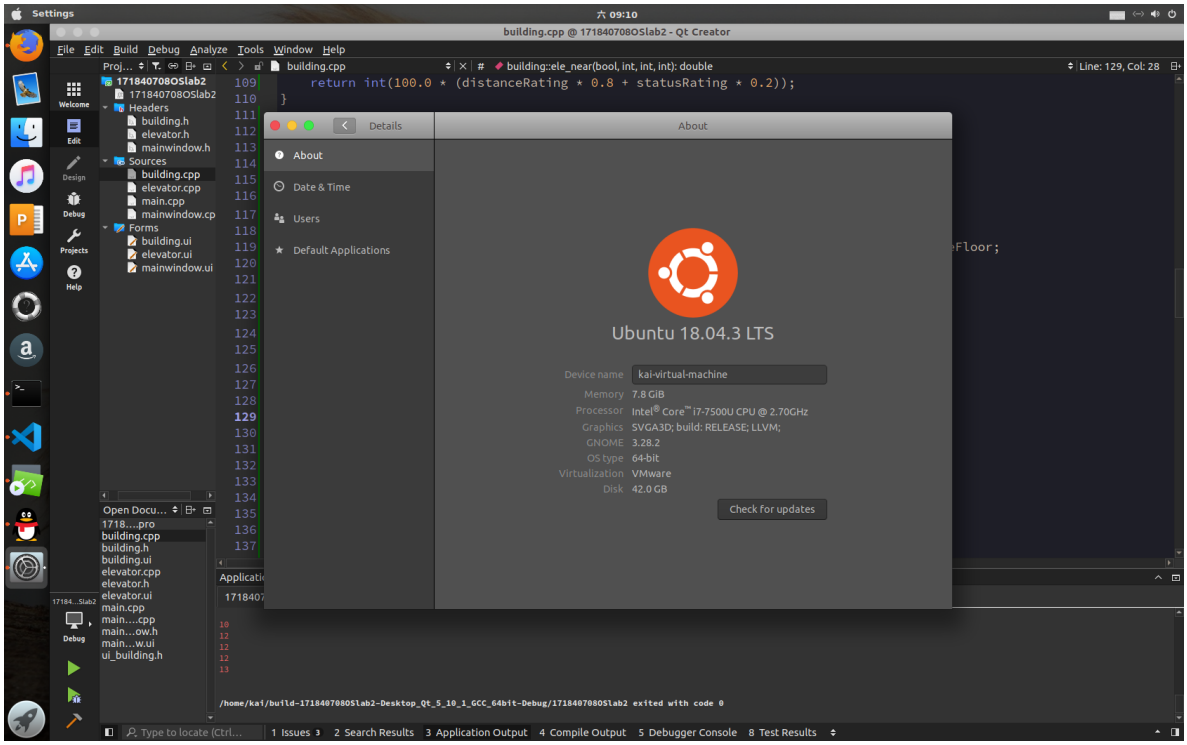
实验任务/需求

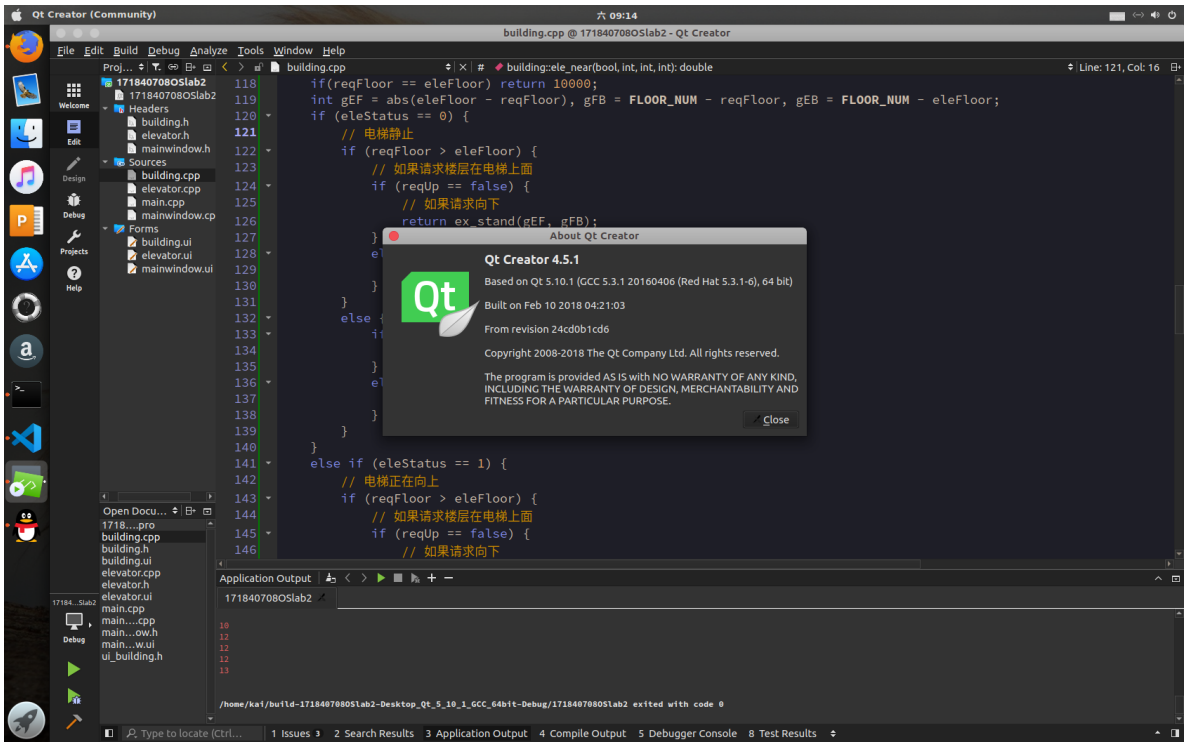
在`Linux`上实现电梯模拟程序, 体现进程间、线程间通信、同步技术, 电梯调度算法.

开发环境

操作系统: `Ubuntu 18.04.3 LTS` (看起来像Mac OS其实真的是Linux, 只是装了个苹果皮肤😊)

开发平台: `Qt 5.10.1(GCC 5.3.1)`



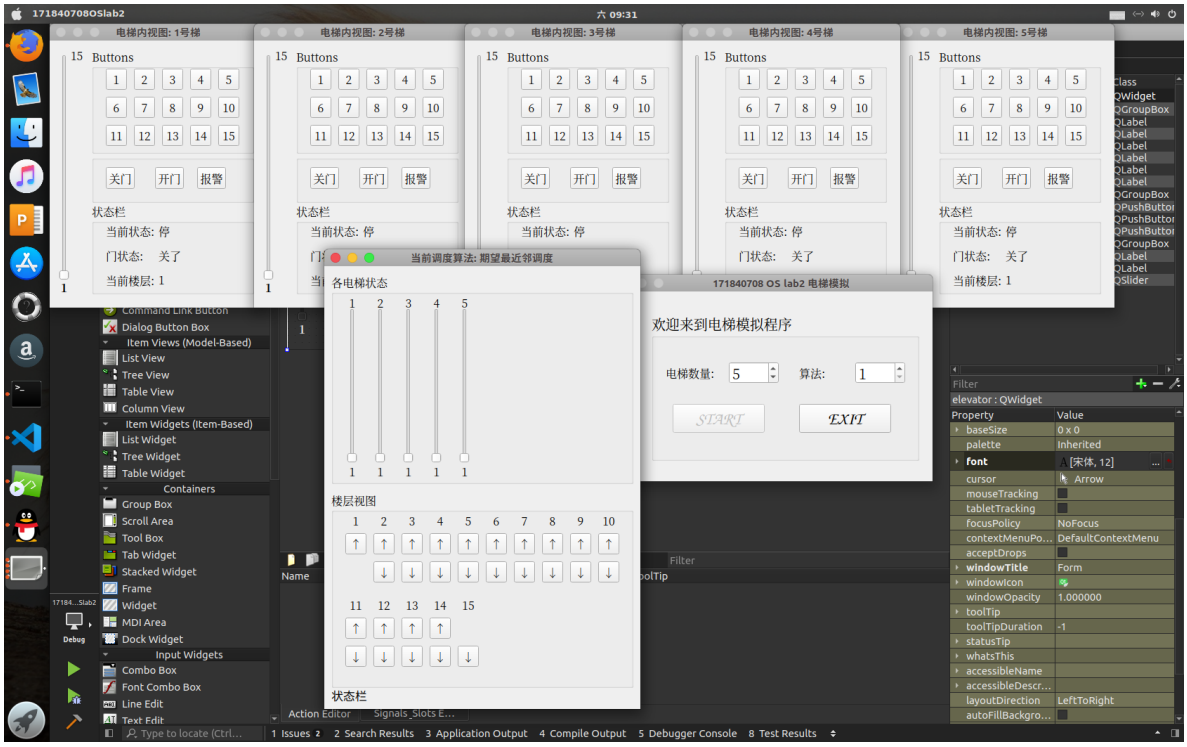


程序说明及功能展示

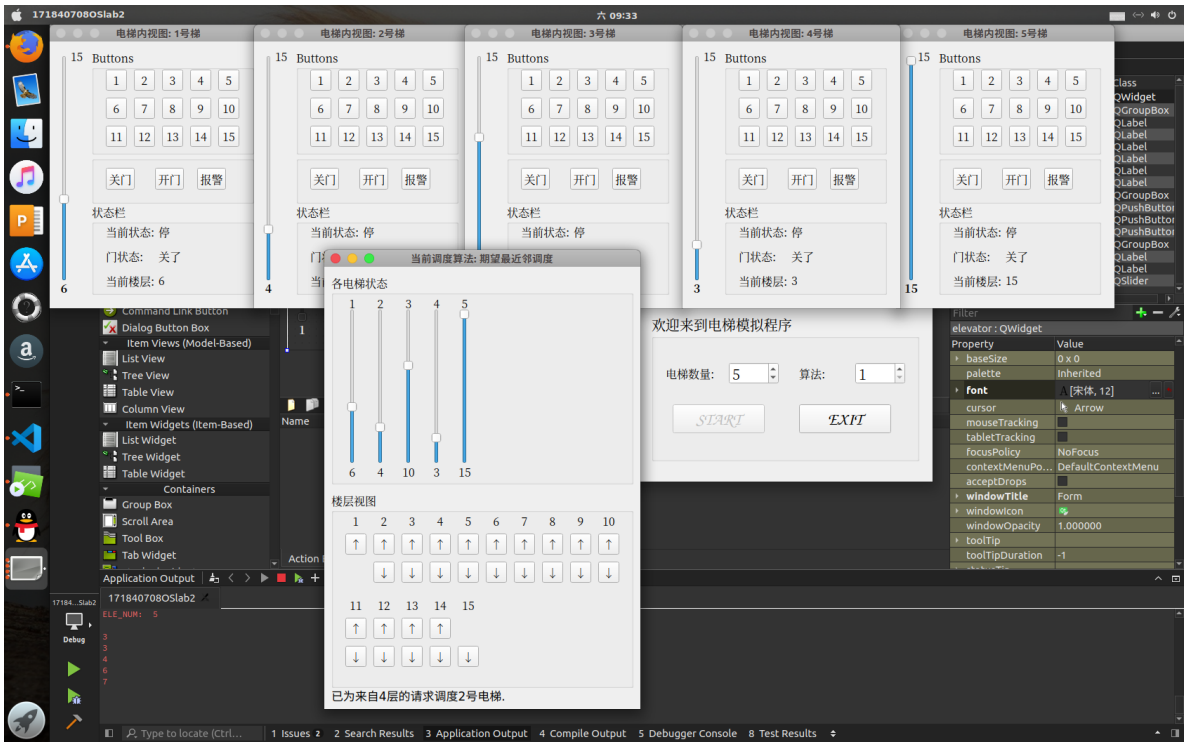
程序启动后, 可以选择**电梯数量**和**调度算法**, 老师给的PDF中说只需要模拟三层建筑中电梯的运行情况, 但是这里楼层总数被设置为`15`层, 以增加调度算法和模拟过程的可测试性以及并行鲁棒性.

点击 START, 画面中生成 电梯数量+1 个窗口, 分别是电梯内的控制视图和电梯外每层楼的控制视图, 其中电梯内视图包含了所有的楼层按钮, 开门关门和报警键. 电梯外每层楼视图包含了每层楼的上下按钮.

下图显示的刚启动时的界面:



下图显示了电梯调度进行时的实时显示:



其中电梯外楼层视图包含各个楼层的按钮, 请求上下行, 按下一个按钮后电梯未到不可再按(显示为灰色, 如下图12, 13, 14层上行按钮), 这模拟现实中要坐电梯时, 按下按钮等待电梯来时, 按钮亮起不可再按的过程. 系统实时刷新现实各个电梯状态, 在状态栏现实调度信息:

其中电梯视图的状态栏包含电梯当前运行状态: 停, 上升, 下降; 点击*开门按钮*, 程序在状态栏模拟了开门状态. 点击报警按钮, 系统弹出对话框模拟报警过程:

进程与线程的通信、同步技术

在本电梯模拟程序中, 不同算法中渗透的进程与线程的通信同步技术略有不同, 但也有很多共性:

不同楼层发送电梯请求的资源互斥访问机制

这是电梯调度的根本需求, 也是所有调度算法需要解决的问题, 这些调度请求并发执行, 它们之间存在不同的相互制约关系, 这也是渗透通信与同步技术最主要的部分, 如下是多个请求怎么传递的过程:

在下面这一段 `Qt` 代码中, `connect(floorBtnsDown[unsigned(i)], &QPushButton::clicked, this, [=]{floorBtnsDown[unsigned(i)]->setEnabled(false);});`, `connect(floorBtnsDown[unsigned(i)], &QPushButton::clicked, this, [=]{ele_select_send(false, i);});` 将每一个点击楼层外按钮的事件触发, 将调用调度电梯算法的函数 `ele_select_send`, 此函数根据不同的算法执行不同的分配任务.

```
for(int i = 0; i < FLOOR_NUM; i++){
        QLabel *floorNo = new QLabel(ui->groupBox_btns);
            floorNo->setGeometry(20 + 40 * (i % 10), 30 + 120 * (i / 10), 30,
30);
            floorNo->setAlignment(Qt::AlignHCenter);
            floorNo->setText(QString::number(i + 1, 10));
            floorNo->show();
        QPushButton *floorBtnUp = new QPushButton(ui->groupBox_btns);
            floorBtnUp->setGeometry(20 + 40 * (i % 10), 60  + 120 * (i / 10),
30, 30);
            floorBtnUp->setText("↑");
            floorBtnUp->show();
            floorBtnsUp.push_back(floorBtnUp);
            connect(floorBtnsUp[unsigned(i)], &QPushButton::clicked, this, [=]
{floorBtnsUp[unsigned(i)]->setEnabled(false);});
            connect(floorBtnsUp[unsigned(i)], &QPushButton::clicked, this, [=]
{ele_select_send(true, i);});
        QPushButton *floorBtnDown = new QPushButton(ui->groupBox_btns);
            floorBtnDown->setGeometry(20 + 40 * (i % 10), 100 + 120 * (i / 10),
30, 30);
            floorBtnDown->setText("↓");
            floorBtnDown->show();
            floorBtnsDown.push_back(floorBtnDown);
            connect(floorBtnsDown[unsigned(i)], &QPushButton::clicked, this, [=]
{floorBtnsDown[unsigned(i)]->setEnabled(false);});
            connect(floorBtnsDown[unsigned(i)], &QPushButton::clicked, this, [=]
{ele_select_send(false, i);});
    }
```

每个请求传递进来之后被 `push` 进一些数据结构内, 轮询扫描这些数据结构, 模拟 `CPU` 时间片轮转的过程, 不断刷新当前状态, 模拟多个进程并发执行的过程.

```
 public:
        int ELE_NUM = 5;
        int FLOOR_NUM = 20;
        int ELE_SELECT_MODE = 1;
        std::vector<elevator*> eles;
        std::vector<QSlider*>  eleSliders;
        std::vector<QLabel*>   eleCurrents;
        std::vector<QPushButton*> floorBtnsUp;
        std::vector<QPushButton*> floorBtnsDown;
```

电梯作为临界资源定义

在这一需求中, 临界资源可以被定义为**一台电梯不能同时为不同楼层开门**. 电梯为不同楼层开门这一过程必须互斥的访问. 但是电梯又可以被各楼层共享, 这符合临界资源的定义.

本程序如何保证互斥访问呢? *首先由电梯的运行设定保证了不存在一台电梯同时满足两层楼的请求的情况, 必须电梯到位, 开门关门之后请求才被满足,* 这由程序逻辑, 运行设定保证了不同楼层是互斥访问电梯的.

状态变量作为信号量定义

定义每个电梯的状态变量为信号量, 判断信号量的过程可以被定义为PV操作. 下面这段位于`elevator.cpp`中的代码, 就是check临界区的进入是否可行.

注意其中`status = 0; open_door(); status = beforeStatus;`代码, 这就是保证临界区互斥访问的又一重要因素, 信号量机制, 准备开门时电梯的运行状态标志位即被置为"静止"状态, 所以即使在`open_door()`之前因为用户按下按钮楼层请求打断了执行`open_door()`函数的操作, 电梯也不会到其他楼层完成访问, 而且在第二段代码中电梯在这种打断情况下会拒绝其他电梯的请求, 保证了当一个进程进入临界区使用临界资源时, 另一个进程必须等待, 只有`open_door()`执行完毕才可以, 这里`destsInsider`是关于电梯内请求的`map`.

```
void elevator::check_state(bool &upDest, bool &downDest) {
    // 电梯内请求满足否
    for (int i = 1; i < currentFloor; ++i) {
        if (destsInsider[i] == 1 || destsOutside[i] == 1)
            downDest = true;
    }
    // 开门;
    if (destsInsider[currentFloor] == 1 || destsOutside[currentFloor] == 1) {
        int beforeStatus = status;
        status = 0;
        open_door();
        status = beforeStatus;
    }
    // 电梯外请求满足否
    for (int i = currentFloor + 1; i <= FLOOR_NUM; ++i) {
        if (destsInsider[i] == 1 || destsOutside[i] == 1)
            upDest = true;
    }
}
```

根据信号量判断拒绝请求代码:

```
bool elevator::recive_request(bool up, int floor, bool forceRecive) {
    // 拒绝request
    if(!forceRecive && (( up && status == 2 && currentFloor > floor )
                        || ( !up && status == 1 && currentFloor < floor )
                       ))
        return false;

    destsOutside[floor] = 1;
    status == 0 ? check_when_pause() : check_when_run();
    return true;
}
```

遵循的准则

空闲让进

期望最邻近算法, 优先级调度算法, 随机调度算法 如果一个电梯是空闲的, 那么它有可能被分配给当前请求, 即期望最邻近的电梯可以被分配给请求.

全响应调度算法: 如果一个电梯是空闲的, 那么它必将被分配给请求.

忙则等待

由"电梯作为临界资源定义"小节可以知道所有算法对此准则满足: 当电梯在为其他楼层开门时, 其他试图请求电梯的进程将进入等待区.

有限等待

对请求访问的进程, 保证在有限时间内可以进入临界区, 这里只考虑有限请求集合, 每个请求的等待时间有限.

期望最邻近算法, 全响应调度算法, 随机调度算法 对于期望最邻近算法, 易证每个电梯对于当前请求的期望到达时间有限, 且电梯集合有限, 由确界原理易知对于电梯期望集合的任意子集下界存在, 让这个电梯响应请求即可. 全响应和随机调度一定保证了有分配, 根据调度原理易证等待时间有限.

优先级调度算法

由信号量机制知可能存在所有电梯拒接请求的情况, 这里设置了一个变量`forceRecive`, 保证在所有电梯拒接请求时, 强制优先级最高的那一台机器接受请求. 这样就避免了无限等待的情况.

```
bool building::send_request(bool up, int floor, elevator *ele, bool forceRecive)
{
    return(ele->recive_request(up, floor, forceRecive));
}
```

让权等待

如果进程不能进入临界区, 则应立即释放处理机, 防止进程忙等待.

```
void building::ele_select_send(bool up, int floor) {
    ui->label_bar->setText("开始处理来自"+ QString::number(floor + 1, 10) +"层的电梯调度请求...");
    if (ELE_SELECT_MODE == 1) {
        eleRatings.clear();
        // qDebug() << "ELE_NUM: " << ELE_NUM << endl;
        for(int i = 0; i < ELE_NUM; i++)
            eleRatings.push_back({i, ele_near(up, floor, eles[unsigned(i)]->currentFloor, eles[unsigned(i)]->status)});
        std::sort(eleRatings.begin(), eleRatings.end(),
                  [](std::pair<int, int> &a, std::pair<int, int> &b){
                        return a.second < b.second;
        });
//        for (int i = 0; i < eleRatings.size(); ++i) {
//            qDebug() << eleRatings[i].second << " ";
//        }
```

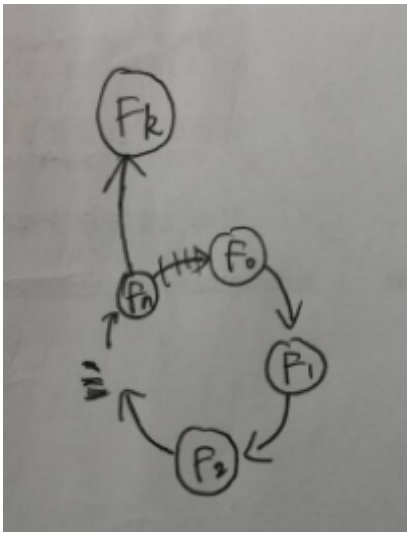
```
//        qDebug() << endl;

        bool successSend = false;
        for(auto i : eleRatings) {
            if (send_request(up, floor, eles[unsigned(i.first)])){
                successSend = true;
                ui->label_bar->setText("已为来自"+ QString::number(floor + 1, 10)
+"层的请求调度" + QString::number(i.first + 1, 10) + "号电梯.");
                return;
            }else{
                ui->label_bar->setText("为来自"+ QString::number(floor + 1, 10)
+"层调度" + QString::number(i.first + 1, 10) + "号电梯的请求被拒绝.");
                continue;
            }
        }
        if(successSend == false) {
            send_request(up, floor, eles[unsigned(eleRatings.begin()->first)],
true);
            ui->label_bar->setText("已为来自"+ QString::number(floor + 1, 10) +"层
的请求强制调度" + QString::number(eleRatings.begin()->first + 1, 10) + "号电梯.");
        }
    }
    // 优先级调度算法
    else if (ELE_SELECT_MODE == 2) {


// ...
```

代码中采用`for`循环轮询判断各个进程是不是可以进入临界区(这个过程可以被定义为处理机时间片分配给各个进程).

证明不会产生死锁

theorem 1 请求时间间隔 > 0, 则不可能满足循环等待条件;

$proof$ 1 首先每个楼层释放电梯资源与否与其他电梯无关, 而且对临界资源的定义是一台电梯下的, 由循环等待定义知: 存在处于等待状态的楼层集合 $\{F_i\}$, F_i申请的资源被F_{i+1}所占有, 并且构成环, **由上述介绍的所有电梯拒接了申请之后的强制分配机制, 并且请求时间间隔 > 0** $\Rightarrow$ 在最后或者中间某个在分配资源时, 由于请求被拒绝, 强制分配了循环等待资源环外的某个等待进程资源 $\Rightarrow$ 存在F_k在等待环中出度为0, 如下图:



由死锁定义可知不能构成等待环, 得证.

调度算法实现

期望最邻近算法

这是一个自己思考的算法, 思路缘起是现实生活中的电梯调度情况, 每一个用户都希望分配到最快到达的电梯, 现实中某些比较坑的电梯存在这样的情况: 10楼有人按了下, 电梯从1楼爬升到5楼的时候又有人在4楼按了下, 调度算法为了节约电, 不会给4楼分配新的电梯, 而是等到10楼下来再"顺便"接上4楼的人, 这是不太人性(为了省电)的调度算法(就如接下来的优先级调度一样), 但是在期望最邻近算法中, 算法会计算每一电梯到达请求楼层的期望时间, 以可能停下楼层的等可能结果计算, 很好地解决了上述问题, 如下是详细计算过程:

不考虑顺向停靠(比如5楼请求上行, 电梯从1楼到5楼这一路上的停靠), 因为按照最邻近调度算法如果顺向停靠被分配, 则是因为电梯资源不够必须被分配, 不能算入一般性概率. 其他情况顺向停靠只会在当前请求被处理完之后才会响应顺向停靠.

不妨设电梯和请求楼层之间距离的绝对值为gEF

$$gEF = |elevator\ \ floor - request\ \ floor|$$

请求楼层和最高层之间的距离为gFB

$$gFB = |request\ \ floor - building\ height|$$

电梯当前楼层和最高层之间的距离为gEB

$$gEB = |elevator\ \ floor - building\ height|$$

1. 电梯静止:
    1. 请求楼层在电梯上面:
        1. 请求向下:

            电梯只可能在请求楼层以上停止, 所以对请求楼层以上停止后的等待时间求期望:

            $$E(x) = (gEF + \sum_{i=1}^{gFB} i \times 2) \div gFB$$

        2. 请求向上:

            由一般性假设, 电梯只可能直接到达:

            $$E(x) = gEF$$

    2. 请求楼层在电梯下面:
        1. 请求向下:

            由一般性假设, 电梯只可能直接到达:

            $$E(x) = gEF$$

        2. 请求向上:

            电梯只可能在请求楼层以下停止, 所以对请求楼层以下停止后的等待时间求期望:

            $$E(x) = (gEF + \sum_{i=1}^{|requestfloor|} i \times 2) \div |requestfloor|$$

2. 电梯正在向上:

1. 请求楼层在电梯上面:

   同1.1

2. 请求楼层在电梯下面:

   1. 请求向下:

      电梯只可能在电梯当前楼层以上停止, 所以对电梯当前楼层以上停止后的等待时间求期望:

      $$E(x) = (gEF + \sum_{i=1}^{gEB} i \times 2) \div gEB$$

   2. 请求向上:

      电梯在电梯当前楼层以上, 请求楼层以下都可能停靠, 对$gEB \times request\ \ floor$种等可能结果的等待时间求期望:

      记在电梯当前楼层以上折返点为第m层, 请求楼层以下折返点为n层, 则此次等待时间可表示为:

      $$per\ wait\ time = ((m - eleFloor) + (reqFloor - n)) \times 2 + gEF$$

      期望和为:

      $$\begin{aligned} p * E(x) &= \sum_{m}\sum_{n} per\ wait\ time \\ &= \sum_{n}(eleFloor * (eleFloor + 1) + 2 * eleFloor * (-eleFloor + reqFloor - n)) + gEF * eleFloor \\ &= reqFloor * (eleFloor * (eleFloor + 1) - 2 * eleFloor^2 + 2 * eleFloor * reqFloor) \\ &\quad - eleFloor * reqFloor * (reqFloor + 1) + gEF * eleFloor * reqFloor \\ &= 2 * eleFloor * reqFloor * (reqFloor - eleFloor) + gEF * eleFloor * reqFloor \\ &= r \times gEB^2 - r^2 \times gEB + r * gEB * (2(r - e) + gEF) \end{aligned}$$

      因为变量`reqFloor`和`eleFloor`较为复杂, 所以看起来比较复杂. 最后一个式子中`eleFloor`简写成`e`, 同理`reqFloor`也是.

      最后再除$gEB \times request\ \ floor$即可.

3. 电梯正在向下:

   1. 请求楼层在电梯下面:

      同1.2

   2. 请求楼层在电梯上面:

      1. 请求向上:

         电梯只可能在电梯当前楼层以下停止, 所以对电梯当前楼层以下停止后的等待时间求期望:

         $$E(x) = (gEF + \sum_{i=1}^{eleFloor} i \times 2) \div eleFloor$$

      2. 请求向下:

         电梯在电梯当前楼层以下, 请求楼层以上都可能停靠, 对$gFB \times elevator\ \ floor$种等可能结果的等待时间求期望:

         记在电梯当前楼层以下折返点为第m层, 请求楼层以上折返点为n层, 则此次等待时间可表示为:

         $$per\ wait\ time = ((eleFloor - m) + (n - reqFloor)) \times 2 + gEF$$

         期望为:

$$E(x) = \sum_{m} \sum_{n} per\ wait\ time$$

代码实现:

```
// 电梯运行在当前时刻停在接下来的任意楼层是等概率的
double building::ele_near(bool reqUp, int reqFloor, int eleFloor, int eleStatus)
{
    if(reqFloor == eleFloor) return 10000;
    int gEF = abs(eleFloor - reqFloor), gFB = FLOOR_NUM - reqFloor, gEB =
FLOOR_NUM - eleFloor;
    if (eleStatus == 0) {
        // 电梯静止
        if (reqFloor > eleFloor) {
            // 如果请求楼层在电梯上面
            if (reqUp == false) {
                // 如果请求向下
                return ex_stand(gEF, gFB);
            }
            else {
                return gEF;
            }
        }
        else {
            if (reqUp == false) {
                return gEF;
            }
            else {
                return ex_stand(gEF, reqFloor);
            }
        }
    }
    else if (eleStatus == 1) {
        // 电梯正在向上
        if (reqFloor > eleFloor) {
            // 如果请求楼层在电梯上面
            if (reqUp == false) {
                // 如果请求向下
                return ex_stand(gEF, gFB);
            }
            else {
                return gEF;
            }
        }
        else {
            // 请求楼层在电梯下面
            if (reqUp == false) {
                return ex_stand(gEF, gEB);
            }
            else {
                double tmp = reqFloor * gEB * gEB - reqFloor * reqFloor * gEB +
reqFloor * gEB * (2 * (reqFloor - eleFloor) + gEF);
                return tmp / (reqFloor * gEB);
            }
        }
    }
    else {
        // 电梯正在向上
```

```
        if (reqFloor < eleFloor) {
            // 如果请求楼层在电梯xia面
            if (reqUp == false) {
                return gEF;
            }
            else {
                return ex_stand(gEF, reqFloor);
            }
        }
        else {
            // 请求楼层在电梯shang面
            if (reqUp == false) {
                return ex_stand(gEF, eleFloor);
            }
            else {
                double tmp = eleFloor * eleFloor * gFB - eleFloor * gFB * gFB +
eleFloor * gFB * (2 * (eleFloor - reqFloor) + gFB);
                return tmp / (eleFloor * gFB);
            }
        }
    }
}
```

优先级调度算法:

确定了两个状态权值表, 通过不同状态的权重来确定调度权重, 权重高的优先调度

1. `distanceRating` 代表电梯位置与请求楼层位置的关系.

电梯位置	`distanRating`
电梯与请求楼层相同	∞
电梯与请求楼层不同	$\frac{\|\|eleFloor - reqFloor\|\|}{FLOOR\_NUM}$

2. `statusRating` 代表电梯状态与请求状态的关系.

电梯状态	请求类型	eleFloor ? reqFloor	statusRating
停止	\		**1.0**
上升	上升	<	**1.0**
		>	**0.2**
	下降	<	**0.6**
		>	**0.4**
下降	上升	>	**1.0**
		<	**0.2**
	下降	>	**0.6**
		<	**0.4**

通过分配权值 `statusRating` 来更新调度的电梯. 对于不同场景(省电等)可以有不同权值配比方式. 最后返回 `distanceRating * 0.8 + statusRating * 0.2;`

```
double building::ele_rate(bool reqUp, int reqFloor, int eleFloor, int eleStatus)
{
    if(reqFloor == eleFloor)
        return 10000;
    double distanceRating = double(abs(eleFloor - reqFloor)) /
double(FLOOR_NUM);
    double statusRating = eleStatus == 0 ? 1.0
        : reqUp ? eleStatus == 1 ? eleFloor < reqFloor ? 1.0 : 0.2
        : eleFloor < reqFloor ? 0.6 : 0.4
        : eleStatus == 2 ? eleFloor > reqFloor ? 1.0 : 0.2
        : eleFloor > reqFloor ? 0.6 : 0.4;
    return distanceRating * 0.8 + statusRating * 0.2;
}
```

全响应调度算法

这个算法相当于同一楼层中电梯按钮不相关, 每次楼层请求即该楼层按钮全部被按下, 全部电梯响应调度.

```
else if (ELE_SELECT_MODE == 3) {
        for(auto i : eles){
            send_request(up, floor, i, true);
            ui->label_bar->setText("已将来自"+ QString::number(floor + 1, 10) +"层
的请求发送至所有电梯.");
        }
}
```

随机调度算法

有时随机响应也不错(狗头)

电梯内按键调度

```
// 绘制电梯按钮的显示框
for(int i = 0; i < FLOOR_NUM; i++){
    QPushButton *btn = new QPushButton(box);
    btn->setGeometry(20+40*(i%5)+210*(i/20), 30+40*(i%20/5), 30, 30);
    btn->setText(QString::number(i+1, 10));
    btn->show();
    connect(btn, &QPushButton::clicked, this, [=] {
        Qbtns[unsigned(i)]->setEnabled(false);
        destsInsider[i] = 1;       // 电梯内按下后, 要去的地方. 这里就是创建一个请求
    });
    Qbtns.push_back(btn);
}
```

电梯内部产生了请求, 程序逻辑不论何时都是主动响应创建请求, 每个请求被`push`进一些数据结构内, 在之后会轮询扫描这些数据结构, 模拟`CPU`时间片轮转的过程, 然后不断刷新当前状态, 模拟多个进程并发执行的过程.

遇到的问题及解决方法

`Qt`真·多坑, 实现并发机制对于`connect`函数的处理还要下一点小心思, 因为触发是随时的! 在这里之前没注意, 以为每个函数在调用结束后逻辑就不对了, 要始终抓住`for`模拟时间片轮转, 刷新状态模拟并发执行的过程, 在对一些细节进行处理, 就能较好地完成模拟.

有些不足的是并发过程是遵从请求时间间隔>0的假设的, 还有调度算法都忽略了现实中开门时间不等的情况.

验收实验

因为`Qt`比较坑, 如果遇到代码复现出问题, 请联系我: `qq`: 645064582, `Tel`:18051988316, 我可以带电脑现场复现.

总结💯

这次实验收获很大, 对进程同步互斥机制和调度算法有了深入的了解. 大概花了50小时完成.

谢谢老师和助教哥的批改~