第一次作业

- 作业简介
- 总体架构
- 同步块和锁
- 调度器设计
- bug分析

作业简介：

模拟**单部多线程电梯**的运行，多楼座均只有一部电梯。

总体架构：

Main类负责所有线程的创建和开启；InputThread线程负责读入数据并将请求放入总等待队列(waitQueue)；Schedule调度线程负责将waitQueue中的请求按照楼座分配给5个不同楼座的等待队列(waitingQueues)；ElevatorThread线程负责完成功能需求，即上下移动、开关门、接乘客、将乘客送到目的地；OutputThread类负责统一输出msg语句；RequestQueue类实例化总等待队列(waitQueue)和各楼座等待队列(waitingQueues)。

- 电梯调度策略

  本次电梯调度(run方法)采用的是指导书给出的ALS策略。

  首先检查是否满足线程结束条件：

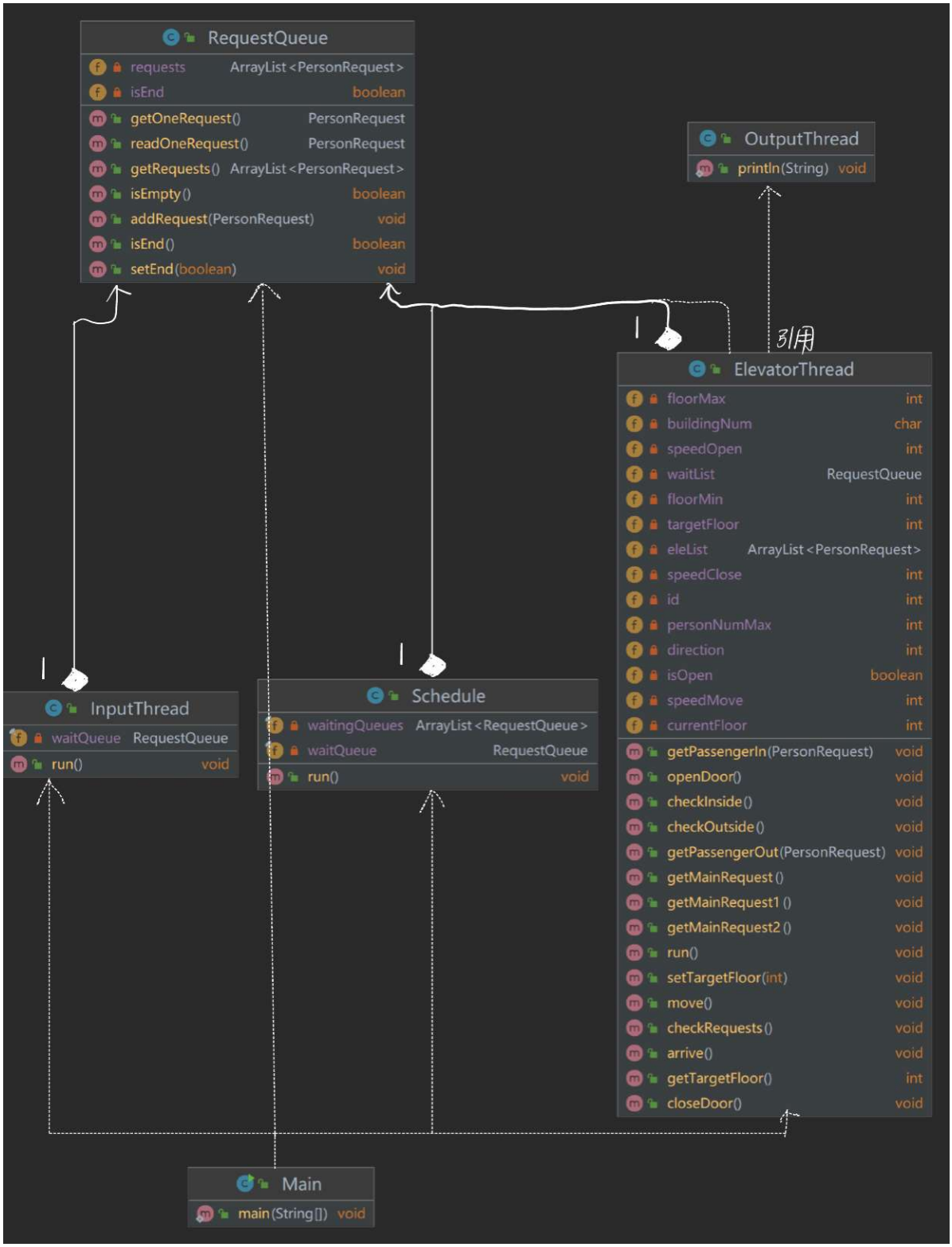
  (IF) 电梯外等待队列(waitList)不会再新增请求 && 电梯外等待队列(waitList)为空 && 电梯内队列(eleList)为空,break出循环即可。

  (ELSE)

  - Step1: checkRequests，分析本层**内**（出电梯？）**外**（捎带？）请求，完成上下人后检查关门。
  - Step2: 如果现在direction=0，即没有方向，刚送完一个人或第一次准备出发时，此时需要获得一个主请求：如果eleList为空，将waitList中第一个请求作为　　主请求，去接人。如果eleList不为空，将eleList第一个请求作为主请求，去送人。如果这二者都不满足，则会wait()直至waitList有新请求加入。
  - Step3：电梯按照direction移动一层，检查此时是否到达targetFloor，若到达，将direction设为0，表明本次主请求已完成，让电梯在下一次循环更新主请求。

*在书写电梯调度部分的时候，指导书其实已经将策略说的很清楚了，但在一些实现细节方面还是de了挺久的bug...例如是先checkRequests还是先 getMainRequest、MainRequest完成后targetFloor和direction该重置成什么值等等。我是先大概写好流程，随便构造几组数据看有没有异常，比如我在一组常规数据发现会出现运行过程中停止输出的情况，最后发现是targetFloor设置有误导致。

- UML类图



同步块和锁：

依照实验代码的思路，把RequestQueue封装成线程安全的类即可，方法使用 synchronized 加锁。RequestQueue作为共享对象分别出现在InputThread和Schedule之间、Schedule和各Elevator之间。

调度器设计：

因为各座只有一部电梯，实际上Schedule调度线程只负责将waitQueue中的请求按照楼座分配给5个不同楼座的等待队列(waitingQueues)，并没有起到调度的作用。

bug分析：

在自行构造数据测试的时候，发现有同一时间点涌入大量乘客请求电梯处理之后不能正常结束的问题，最终发现问题还是出在ElevatorThread**复杂度最高**的run方法中...本次互测中，没有被发现bug。鉴于在群内大佬讨论中发现似乎多线程即使发现了bug也很难在评测机hack到，所以只交了一发简单的数据，hack到了一份代码，问题应该在没有封装专门的线程安全输出类，只需要按照讨论区大佬对于这个问题的分析，就可以避免输出时间戳反复横跳的问题。

第二次作业

- 作业简介
- 总体架构
- 同步块和锁
- 调度器设计
- bug分析

作业简介：

模拟一个多线程环形实时电梯系统。电梯分为楼层电梯和楼座电梯，各自运行，互不干扰。

总体架构：

Main类负责所有线程的创建和开启；

InputThread线程负责读入数据，如果为personRequest，将请求放入总等待队列(waitQueue)；如果为ElevatorRequest，将请求交给ElevatorFactory创建电梯并分配给相应的building；

Schedule调度线程负责将waitQueue中的请求按照楼座分配给5个楼座或10个楼层之一的等待队列(waitingQueue)；

Building线程可以拥有多部电梯，它的任务就是在run方法内不断将请求指派给多部电梯中的一部;

ElevatorThread线程负责完成功能需求，即上下移动、开关门、接乘客、将乘客送到目的地；

OutputThread类负责统一输出msg语句；

RequestQueue类实例化总等待队列(waitQueue)和各楼座等待队列(waitingQueues)。

- 电梯调度策略：

  本次电梯调度(run方法)仍采用的是指导书给出的ALS策略。

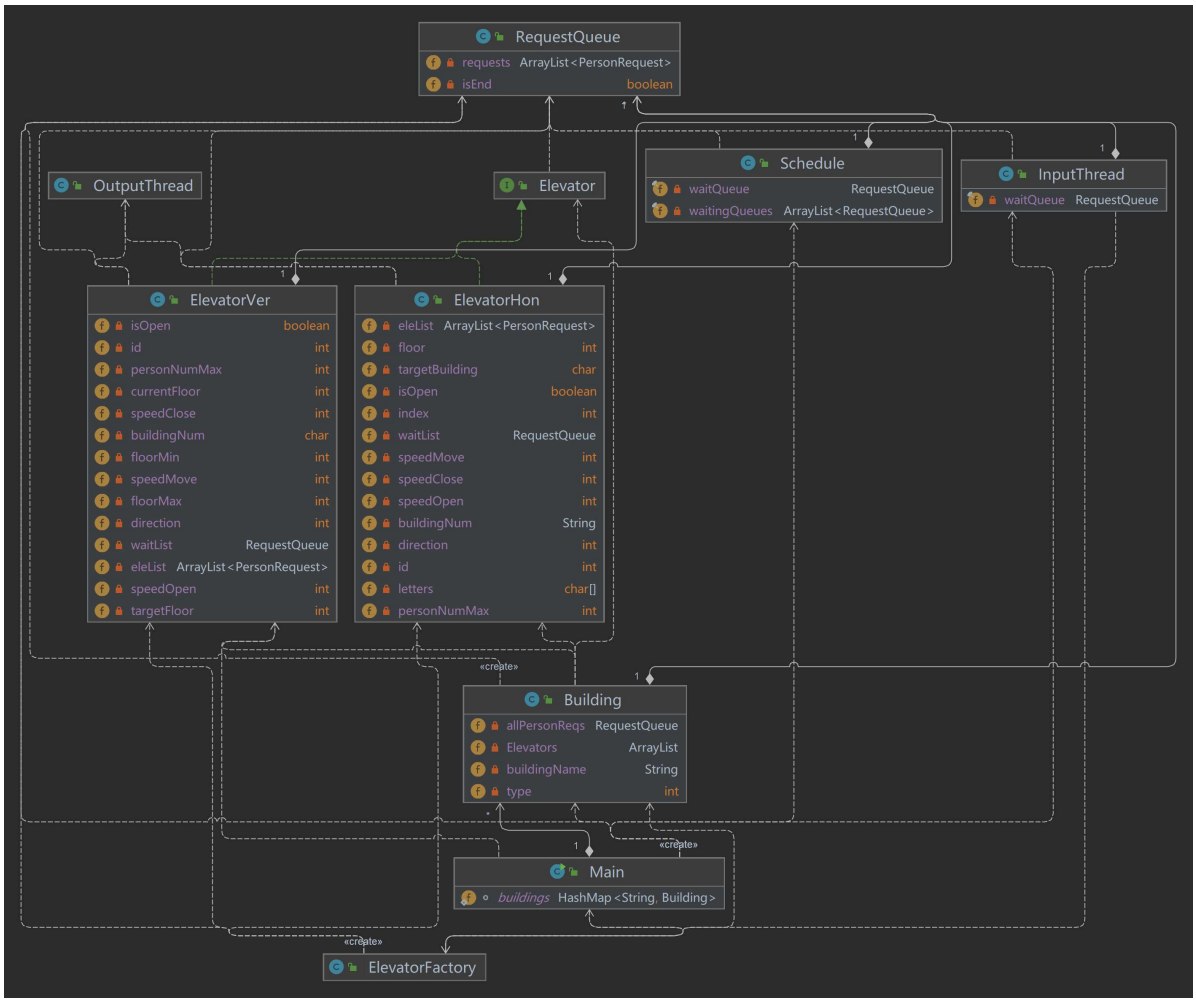
  楼层电梯部分即ElevatorVer类沿用第一次作业的设计，没有改变。新创建了ElevatorHon类，内部采用指导书给出的类似ALS策略。这两个类均实现Elevator接口，便于在building层面统一管理。

- 后续扩展：

本次作业的楼座和楼层之间是相互独立的，不存在类似“A-FLOOR3---------->E-FLOOR7”这样的请求，当面对这样的请求时，可能需要对电梯类内部做出改动，即它在把乘客送到目的地时需要知道这是乘客最终的目的地，还是只是中转站。如果是中转站，需要对乘客请求做修改后丢回schedule队列重新接受分配。

假设电梯增加新属性，不同电梯所能停靠的楼层不同怎么处理？可以维护一个记录是否可达楼层的数组，在每到一层的时候checkRequests之前先判断一下是否需要进行后续工作。

- UML类图：



同步块和锁：

基本没有做改动，因为共享对象统一使用的是RequestQueue，而这个类又是功能完整且线程安全的，所以使用就很方便，不需要另加锁处理。

调度器设计：

本次作业其实就是进行了三次请求分配。第一次在输入阶段将personRequest和ElevatorRequest分流开；第二次在Schedule阶段把personRequest进一步按楼层（楼座）分流；第三次是在楼座（楼层）内部分流给指定一部电梯。

bug分析：

公测和互测中没有被找出bug（不过性能分也挺低就是了...），hack到了一份代码，请求处理完之后不能正常结束造成 REAL_TIME_LIMIT_EXCEED问题。

第三次作业

- 作业简介
- 总体架构
- 度量分析
- 同步块和锁
- 调度器设计
- bug分析

作业简介：

模拟一个多线程环形实时电梯系统。

新增：**扩大乘客的移动需求**，每个乘客的起始位置和目的位置可以在楼层和楼座上都不同。

> **新增电梯定制化功能**，对纵向电梯的运行速度、可容纳人数，横向电梯的运行速度、可容纳人数、可开关

门信息进行**定制**。

总体架构&类功能分析：

整体架构沿用第二次作业，但在诸多细节方面还是做了不少修改。

新增Person类：为personRequest新增了换乘层(transFloor)属性，同时增加setFromBuilding和setFromFloor方法。

Schedule类：需要对FromBuilding和ToBuilding不同的personRequest计算换乘层(transFloor)。如果transFloor == fromFloor，乘客只需要在本层乘坐横向电梯到ToBuilding，如果楼层还不对，就等竖直电梯把他接上即可，所以此时我们把这个乘客丢进FLOORx（某层楼的横向电梯们）。 而如果transFloor ！= fromFloor，证明这个乘客一定会坐三次电梯：坐竖直电梯到transFloor——坐横向电梯到ToBuilding——坐竖直电梯到toFloor。所以我们要记录一下乘客的transFloor，再把这个乘客丢进Buildingx(某座楼的竖直电梯们)。 同时，schedule线程结束的条件也做了修改，在下文阐述。

Building类：在run方法内不断将请求指派给多部电梯中的一部时，需要按照横向or竖直分开处理。因为多部横向电梯可以是可达楼座各不相同的，所以乘客在到达换乘楼层时，需要Building将其分配给一部能到达他的目的地的电梯。

ElevatorHon类：新增了boolean[] path = new boolean[5]数组记录可达楼座，在每次开关门、进出人之前都增加了是否可达的判断，如果不可达，就只是经过，而不进行任何操作。 同时在getPassengerOut时新增判断，如果此时乘客还没到对应楼层，就把他放进本楼座的请求队列中，再坐一次竖直电梯，同时修改掉"需要换乘"这一属性。

ElevatorVer类：同ElevatorHon类，在getPassengerOut时新增判断，如果此时乘客还没到对应楼座，就把他放进本楼层的请求队列中，再坐一次横向电梯，同时修改掉"需要换乘"这一属性。

其他类基本无改动。

- 电梯调度策略：

  同第二次作业，调度策略不变，只在横向电梯中加入“可达”要求。

- 思路分析：

  正如指导书训练目标所言，我觉得本次作业难度在于**线程之间的交互，强化线程之间的协同设计。**不同于上一次作业乘客请求由Input—Schedule—Building—Elevator—Output这样顺序地通畅执行，增加了换乘请求后，乘客最多可以在三部电梯之间反复横跳。这时候以前操作所有线程结束的方法就行不通了：InputThread结束后告诉Schedule没有新请求输入了，你把你手头的请求调度完就行了。Schedule调度完毕之后告诉每座楼，没有新请求会调度到你那儿了，你把你手头的请求分给电梯就完事了。Building分完请求之后，告诉自己手下的全部电梯，没有新请求再分配给你们啦，你们把自己等待队列和电梯队列处理完就行了。**但因为有换乘的存在，Schedule调度完毕之后不能给每座楼setEnd。**虽然采用的是静态调度，在第一次调度之前就计算好了中转楼层，计划好了乘梯路线，即一个personRequest只会进Schedule一次，剩下时间只会在不同Building和不同Elevator间移动。但是Schedule的功能不只有调度请求，还要给每座楼setEnd。

  https://www.cnblogs.com/zxc3wyx/p/14687490.html

  这位学长的博客给了我启发：*“各部电梯之间存在交互，所有电梯线程必须同时停止，因此需引入监听者模式，**每部电梯将自身状态的变化（工作或闲置）报告给输入器，若输入器停止且每部电梯空闲，则电梯线程可以安全停止。**”我感觉挺合理的，只要有一部电梯在工作，其他电梯就都有可能接到新的换乘请求。

  所以最终我选择在Schedule中保存了所有Elevators，新增checkElevators()方法用来检查是不是所有电梯都空闲。只有满足条件**Schedule的请求队列为空 && 不会再给Schedule新请求 && checkElevators()**时，才会给每座楼setEnd。但是还存在一个问题，就是Schedule在checkElevators() == false时会去请求队列中尝试getOneRequest 并调度它，但此时请求队列为空，Schedule就会进入wait()状态，靠什么唤醒呢。最终我选择在每次任一部电梯送出去一个人之后，往Schedule请求队列中addRequest(null)【其实就是告诉Schedule有一部电梯的状态改变了，可能此时所有电梯都空闲了，需要你醒来检查一下...】，这样既可以唤醒Schedule同时不会影响结果。

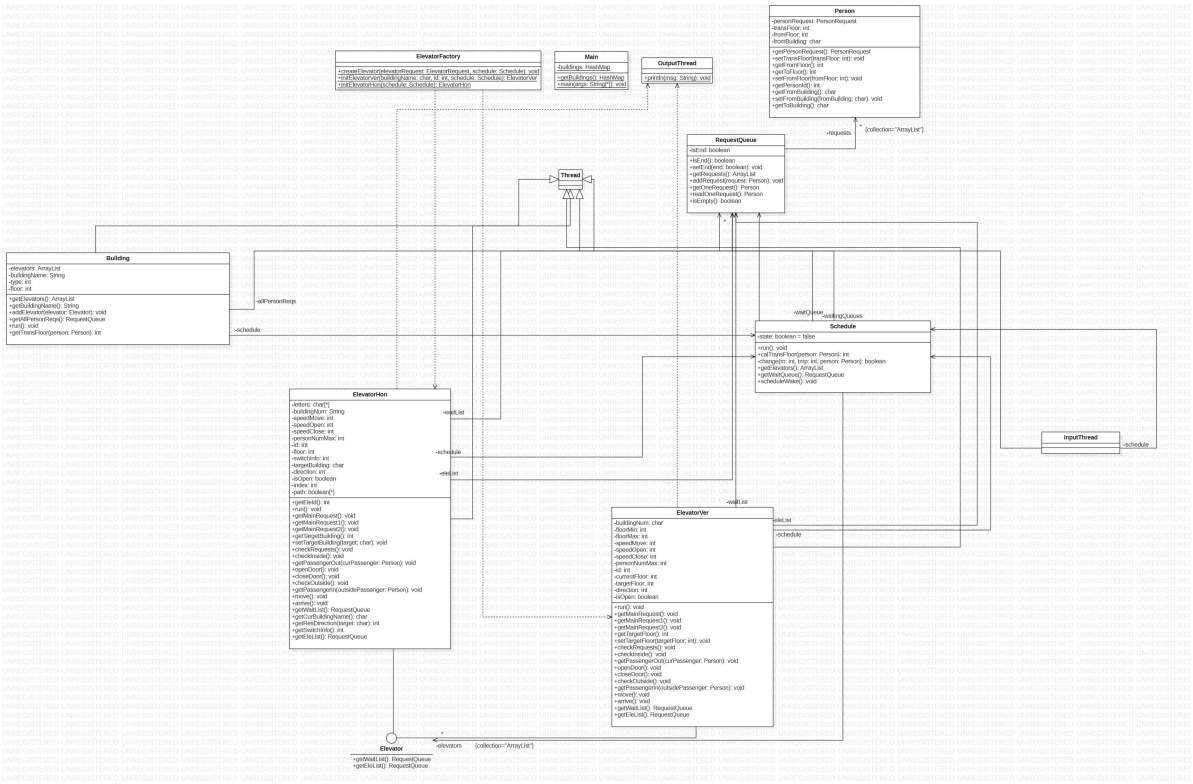
  而这样的实现是有bug的，如果有两个电梯几乎同时结束全部工作，就会出现schedule被连续唤醒的情况，而不是预想的：唤醒——检查所有电梯状态——决定是否结束，被提前唤醒的结果就是无法正常结束，rtle...解决办法就是让“检查所有电梯状态（给出一个属性status）——唤醒（执行run方法里的判断）”一气呵成，保证**只要有一部电梯状态变化，schedule就检查一次。**（但最终证明还是不太对呢...不过现在是会提前结束emmm...）

```
public synchronized void scheduleWake() {
        int i;
        for (i = 0; i < elevators.size(); i++) {
            Elevator elevator = elevators.get(i);
            if ((!elevator.getWaitList().isEmpty()) |
(!elevator.getEleList().isEmpty()))             {
                state = false;
                break;
            }
        }
        if (i == elevators.size()) state = true;
        waitQueue.addRequest(null);     //唤醒wait中的schedule线程
    }
// 接着执行run方法里的判断
         if (waitQueue.isEmpty() && waitQueue.isEnd() && state) {
```

```
            //给每座楼请求队列setEnd
            //结束schedule线程
        }
```

度量分析:

- UML类图



- 代码复杂度分析

一些复杂度超标的方法...

Schedule.Schedule(RequestQueue, ArrayList<RequestQ	0	1	1	1
Schedule.run()	28	4	18	19
Schedule.getWaitQueue()	0	1	1	1
Schedule.getElevators()	0	1	1	1

run方法不需要分配请求，还要承担判断可达，检查电梯状态等工作，与其他类耦合更多了。

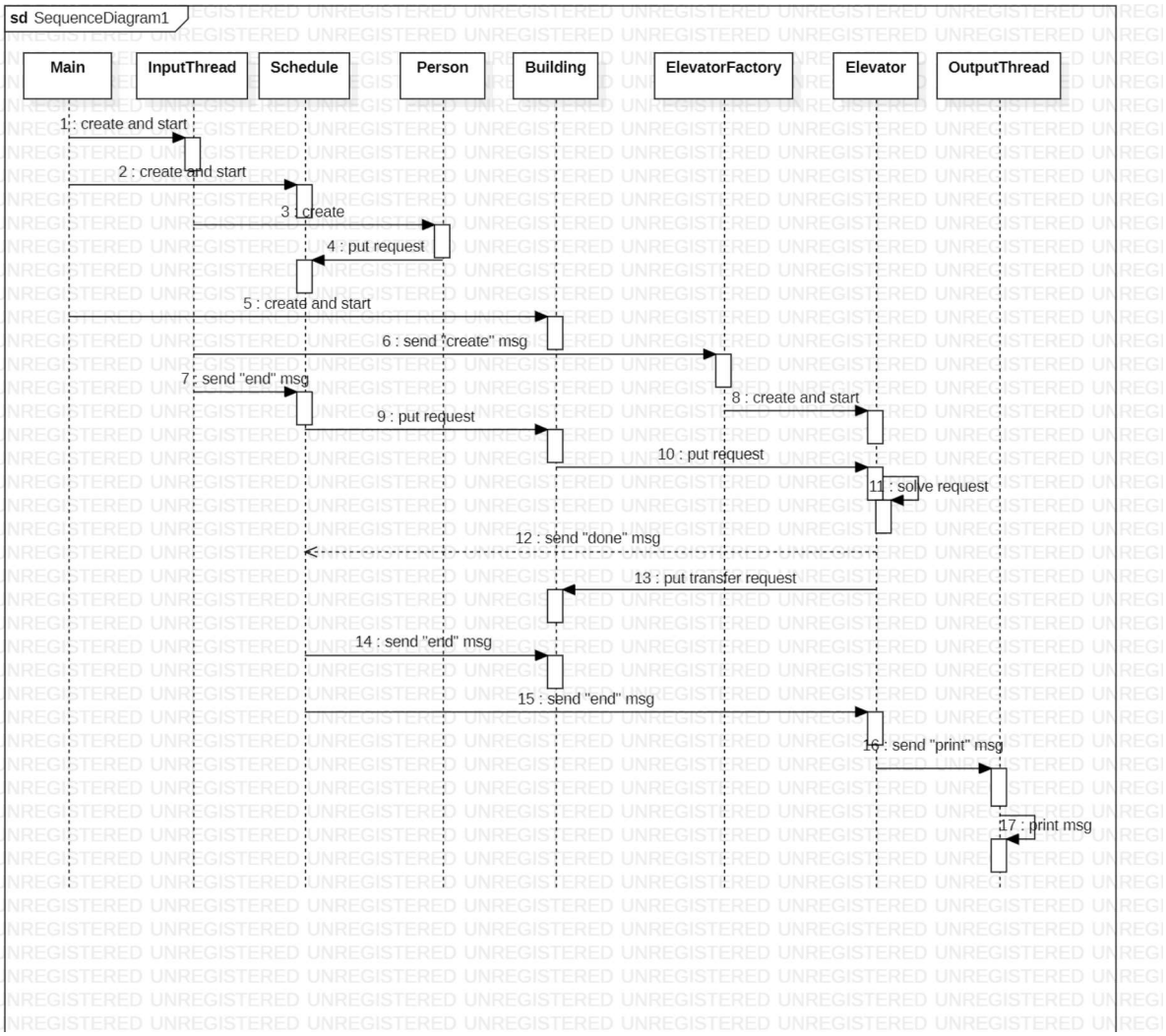
ElevatorVer.setTargetFloor(int)	0	1	1	1
ElevatorVer.run()	18	5	9	11
ElevatorVer.openDoor()	3	1	3	3

一如既往辛苦的电梯（。

Building.run()	28	8	10	12
Building.getTransFloor(Person)	10	5	4	5
Building.getElevators()	0	1	1	1

楼座（层）要承担判断可达后分配，替schedule计算换乘层等工作。和schedule有同样的问题存在。

- 时序图:



同步块和锁：

将Schedule里新实现的方法全部用synchronized加锁，同时为了保证使用方法时是对同一个对象加锁，把schedule作为共享对象也传到了每部电梯里。

调度器设计：

总体流程没有改变，同第二次作业。只是在Schedule内部增加了关于换乘和可达性的计算部分。因为最坏的情况也就是三次换乘，并不是不可预知次数的换乘，所以我把这部分工作交给电梯去做了，并没有给调度器增加额外负担。

心得体会

线程安全：前两次作业中需要访问共享对象的地方比较单一且明确清晰，故不太容易出问题。第三次作业需要不同共享对象之间交互传递信息时，就需要好好考虑访问的次序和怎么加锁才能确保访问修改的结果和我们想要的结果是一样的。

层次化设计：主要就是一个请求怎么被分配到一部电梯，这个分配过程类似流水线处理，各个执行单元各司其职，做完自己的工作就把这个请求传给下一个执行单元。