# Лабораторная работа №4: Многопоточность

Проводим моделирование процесса с общими ресурсами, которые необходимо использовать по очереди.

Моделируем посетителей, приходящих в бизнес-центр.

Каждый посетитель заходит в здание, проходит через проходную (одновременно 1 человек), вызывает лифт, едет на свой этаж, некоторое время там находится, потом снова вызывает лифт и выходит.

1. Создайте класс «Visitor» (не путать с шаблоном проектирования)

|  |
| --- |
| **Visitor** |
| - place : BusinessCenter  - totalCount : int  - num : int  - floor: int |
| **<<constructors>>**  + V**isitor**( p : BusinessCenter)  **<<methods>>**  + run()  - enterBuilding()  - goUp()  - doSomeWork()  - goDown()  + toString() : String |

totalCount – счётчик посетителей, статическое поле, используемое в конструкторе для при присвоении номера

floor­ – на какой этаж идёт посетитель (случайное число 1...10)

num – номер посетителя, инициализируется в конструкторе с использованием totalCount, используется в toString()

Реализуйте интерфейс Runnable в классе Visitor. Метод run просто по очереди вызывает остальные методы, которые пока что можно оставить пустыми (заглушками).

Класс Main cоздаёт 1 бизнес-центр и n посетителей с некоторой задержкой, запускает для каждого из посетителей отдельный поток и ждёт их завершения.

Класс «Visitor» (посетитель) не отвечает за синхронизацию посетителей в здании. Его методы могут вызываться, например, по приходящим запросам в многопользовательской игре.

В данном приложении мы отказываемся также от распространённой архитектуры, где все запросы клиентов добавляются в очередь, а ответные действия происходят по сообщениям, приходящим в ответ через некоторое время и реализуем всю синхронизацию в методах класса «BusinessCenter».

1. Создайте класс «BusinessCenter». Он отвечает за всё «обслуживание» посетителей – лифт, проходную и т.д.

|  |
| --- |
| **BusinessCenter** |
| - liftFloor : int  - visitorAtControl : Visitor  - visitorInLift : Visitor  - liftFree : boolean |
| **<<constructors>>**  + BusinessCenter()  **<<methods>>**  + enterControl(v: Visitor): boolean  + passControl(v: Visitor)  + callLift(v: Visitor) : boolean  + moveLift(v: Visitor, targetFloor: int)  + enterLift(v: Visitor)  + exitLift(v: Visitor) |

Для простоты считаем, что в лифте и проходной *одновременно может находиться один человек*. На этаже посетители могут находиться одновременно.

Весь общий доступ контролируется только в этом классе. Отдельные потоки для лифта и проходной **не создаём,** а рассматриваем их как общие разделяемые ресурсы (типа файлов или сетевых подключений), иначе всё вместо сведётся к несложному обмену сообщениями между потоками через одну или несколько очередей.

Например, метод enterBuilding()класса Visitor вызывает сначала enterControl, а затем passControl (если enterControl вернул true).

Метод enterControl:

* Код ожидания освобождения проходной (сначала можно пропустить)
* Занятие проходной (занесение ссылки на посетителя в соответствующее поле visitorAtControl)
* Отладочная печать на всех этапах

Метод passControl:

* Контрольная проверка, что проходная свободна (эта проверка всегда должна проходить при отсутствии ошибок в программе, так что можно в противном случае выбросить исключение)
* Thread.sleep для имитации задержки (в реальном приложении там может быть проверка паспорта клиента по базе данных, например)
* Код освобождения проходной для последующих посетителей
* Отладочная печать на всех этапах

Аналогично работает пара методов callLiftAndWait / exitLift. Функция callLiftAndWait возвращает true не когда лифт приехал, а когда он лифт готов ехать на этаж к посетителю (пустой!), т.е. «зарезервирован». Методы goUp и goDown после callLiftAndWait вызывают moveLift (для пустого лифта), затем enterLift и второй раз moveLift (для поездки на нужный этаж), и в конце exitLift. Задержка в методе moveLift зависит от расстояния между этажами liftFloor и targetFloor, а поле liftFloor в результате вызова moveLift изменяется (можно с «анимацией» в цикле, тогда за зданием можно будет наблюдать в реальном времени).

1. Добавьте базовую синхронизацию с использованием wait/notify.

Так как действия, которые нужно производить в режиме монопольного доступа к проходной или лифту разнесены на несколько методов, обойтись простыми synchronized-блоками не получится (можно было бы сделать synchronized void passControl без enterControl). Кроме того, проходящие проходную посетители не должны задерживать желающих ехать на лифте.

Переменные visitorAtControl и liftFree ограничивают доступ к общим ресурсам из различных потоков. Их можно корректно присваивать только если текущее значение соответствует свободной проходной или лифту (при этом лифт, вызванный посетителем, также считается занятым).

Если функция, enterControl желающая получить доступ к проходной для нового посетителя, заметит, что проходная занята (visitorAtControl !=null), то ей нужно ожидать освобождения с помощью вызова wait(). Метод passControl должен обязательно сбросить visitorAtControl и корректно вызвать notify() или notifyAll(). Поддерживать очередность ожидающих для простоты не требуется, первым проходит любой из ожидавших (иначе придётся заводить блокирующую очередь или использовать Fair Lock).

Аналогично синхронизируется доступ к методам, отвечающим за лифт (ожидаем состояния liftFree==true).

Контрольные вопросы:

* Почему синхронизированные блоки стоят именно в этих местах и с этими аргументами?
* Можно ли вызвать wait или notify вне синхронизированного блока?
* Нужно ли повторно проверять условие после выхода из wait?
* Есть ли гарантия в нашем коде, что поток, вызвавший wait, будет кем-то разбужен?

1. **Дополнительные усложнения**
   1. Порождайте потоки для посетителей с использованием пула потоков (Executors.newFixedThreadPool) и ожидайте их выполнения с помощью Future::get.  
      **(очень желательно)**
   2. Вместо метода run вызовите по очереди методы создаваемых объектов Visitor через CompletableFuture.

<http://2015.jpoint.ru/presentations/JPoint2015_Chuyko.pdf>

<http://www.baeldung.com/java-completablefuture>

**(усложнённая альтернатива 4.1)**

* 1. Создайте два отдельных объекта (Object) для блокировки проходной и лифта и используйте их для синхронизации

**(достаточно просто)**

* 1. Используёте ReentrantLock и Condition вместо синхронизированных блоков (за счёт этого можно, например, обеспечить более честную очерёдность ожидающих). <https://metanit.com/java/tutorial/8.10.php>

**(усложнённая альтернатива 4.3)**

* 1. Добавьте следующее поведение: если посетитель ожидает лифта более секунды моделирования, он идёт пешком и лифт не занимает ☺

Замечания по печати:

* Порядок печати в несинхронизированном коде может быть перемешан. Лучше большую часть сообщений печатать внутри методов **BusinessCenter.**
* Можно отсчитывать время от момента создания BusinessCenter (завести ещё одно статическое поле startTime), функция System.currentTimeMillis().

Пример вывода (можно и более подробно):

0 ms: Пришёл Посетитель1.  
1 ms: Пришёл Посетитель2.  
1 ms: Посетитель1 показывает документы.  
330 ms: Посетитель1 показал документы.  
331 ms: Посетитель1 вызывает лифт.  
332 ms: Посетитель2 показывает документы.  
330 ms: Посетитель2 показал документы.  
380 ms: Посетитель1 едет на этаж 7.  
Посетитель2 ждёт лифт.   
Посетитель1 вышел из лифта.  
Посетитель2 ждёт лифт с этажа 5.  
Посетитель1 что-то делает.  
Посетитель2 едет на этаж 7.  
Посетитель2 что-то делает.  
Посетитель1 закончил дела.  
Посетитель1 вызывает лифт.  
Посетитель2 закончил дела.  
Посетитель2 вызывает лифт.  
Посетитель2 ждёт лифт с этажа 7.  
Посетитель1 едет на этаж 1.  
Посетитель1 вышел из лифта.  
Посетитель2 едет на этаж 7.  
Посетитель2 едет на этаж 1.  
Посетитель1 вышел.  
Посетитель2 вышел.