# Grunnatriði stýrikerfa
# Forritunarverkefni 1 - samstilling þráða
# Assignment 1 - Thread Synchronization

February 2018

English follows.

## Afurðin

Afurðin er forrit, byggt á grunninum sem gefinn er með verkefninu, sem stýrir lyftum, hverri lyftu í sínum þræði, til þess að taka við og skila af sér fólki, hverri persónu í sínum þræði, á þann hátt að það birtist rétt í viðmótsforritinu (gefið) og búi fólkið til m.v. skipanir frá testSuite þræðinum sem er settur sjálfkrafa af stað í upphafi keyrslu.

Að auki uppfært skjal, sams konar og í fyrri hluta verkefnis, sem nefnir 1) þær semafórur sem búnar eru til í kerfinu, 2) þá atburði (t.d. lyfta kemur á hæð, lyfta yfirgefur hæð, persóna kemur á hæð) sem gerast í kerfinu og 3) hvaða aðgerðir er kallað á á semafórunum við hvern atburð. Skýringarmyndir og kerfislýsingar eru einnig velkomnar.

Til viðbótar á að setja í skjalið lýsingar á þeim vandamálum sem koma upp vegna aðgengis þráðanna í kerfinu að sameiginlegum minnissvæðum og þeim semafórum sem notaðar eru til að leysa þau.

## Að sækja og skila verkefni

Inni á síðu verkefnisins á myschool er skrá með Java projecti sem er grunnurinn að verkefninu. Sækið, opnið og endurnefnið verkefnið með nafni a.m.k. eins meðlims hópsins ykkar. Þegar forritið er tilbúið er því pakkað í zip/rar/7z skrá og skilað ásamt PDF skrá með skjalinu sem lýsir virkni forritsins. Verkefnið má vinna í allt að þriggja manna hópum.

# Vandamálið

Fólk kemur inn í hús og vill taka lyftu milli hæða. Hver lyfta er útfærð sem þráður og hver persóna er útfærð sem þráður. Á hverjum tímapunkti inniheldur kerfið því þráð fyrir hverja lyftu og þráð fyrir hverja persónu sem hefur komið inn í kerfið og hefur ekki náð að ljúka keyrslu með því að komast inn á hæðina sem hún var á leiðinni á.

Þræðirnir eiga að sjá um að bíða á réttum stöðum eftir að þeir komist áfram í kerfinu og að losa um þá lása sem eðlilegt er að losa á hverjum tímapunkti til þess að aðrir þræðir komist einnig áfram í kerfinu.

Í þessari hönnunarvinnu má gera ráð fyrir að lyftuþræðirnir séu tilbúnir og byrjaðir að keyra og að utanaðkomandi áhrif setji í gang persónuþræðina og velji á hvaða hæð þeir byrja og á hvaða hæð þeir ætla. Þið þurfið þó að hanna alla virkni þráðanna í keyrslu.

Mismunandi flækjustig kerfisins eru eftirfarandi:

- **50%** - Ein lyfta, tvær hæðir, allir koma inn á neðri hæð og yfirgefa lyftuna á efri hæð.
- *(Millistig)* Ein lyfta, óákveðinn fjöldi hæða, allir koma inn á neðri hæð og yfirgefa lyftuna á einhverri hæð. Persónuþráðurinn veit frá upphafi á hvaða hæð hann vill yfirgefa lyftuna.
- *(Millistig)* Ein lyfta, tvær hæðir, persónur geta komið inn á hvorri hæðinni sem er og yfirgefa hana á hinni hæðinni.
- **30%** - Ein lyfta, óákveðinn fjöldi hæða, persónur geta komið inn á hvaða hæð sem er og yfirgefið lyftun á hvaða hæð sem er. Persónuþráður veit frá upphafi á hvaða hæð hann vill yfirgefa lyftuna.
- **20%** - Óákveðinn fjöldi lyfta, óaákveðinn fjöldi hæða, persónur koma inn og yfirgefa lyftuna á hvaða hæð sem er. Persóna kemur inn á hæð og getur notað hvaða lyftu sem er til að ferðast.

Hvert kerfi má útfæra þannig að pláss sé fyrir eina manneskju í lyftunni eða þannig að pláss sé fyrir 6 manneskjur í lyftunni. Þið megið velja hvenær þið bætið þessu flækjustigi inn, en á endanum eiga öll stigin að gera ráð fyrir 6 manneskjum í lyftu í einu.

Endilega bætið við millistigum flækju ef það hjálpar ykkur að skilja vandamálin. Einnig má sleppa stigum í lokaskjalinu ef ykkur þykja þau vera innifalin í öðrum stigum, en bætið samt ekki inn of miklu í einu til að hafa betri yfirsýn yfir hvað veldur villum og vandamálum í kerfinu.

Bónus stig:

- **5%** - Leysið test case 5 í **TestSuite** án þess að svelta ákveðnar hæðir þangað til aðrar hæðir eru tómar.
- **5%** - Útfærið hugmyndina um upp/niður takka og látið fólk einungis fara inn í lyftur sem eru á leiðinni í rétta átt.

- **5%** - Gerið allt kerfið **thread safe**, með því að passa upp á **mutual exclusion** á öllum sameiginlegum breytum.

Comment í kóðanum lýsa því ágætlega hvað þarf að gera hvar og hvað má gera, en rennið einnig yfir vinnufyrirkomulagið að neðan til að fá nánari útskýringar á verkefninu.

## Vinnufyrirkomulag

Áður en hafist er handa við að útfæra kerfið fyrir tiltekið flækjustig skal fara gegnum vinnufyrirkomulag fyrri hluta verkefnis fyrir það flækjustig. Að auki er gott að kynna sér dæmatímaverkefni 4 sem lýsir Java klasanum **Semaphore**. Einnig er gott að kynna sér fallasafnið **java.util.concurrent**, ef þið viljið nota aðrar aðferðir en venjulegar teljarasemafórur.

Í upphafi skuluð þið kynna ykkur forritið sjálft. Í java projectinu eru 4 klasar. Ykkar aðalaðgangur að kerfinu er í gegnum klasann **ElevatorScene**. Klasann **ElevatorGraphics** ættuð þið að láta alveg eiga sig, enda er hann þarna einungis til þess að birta framgang forritsins ykkar á skjá. Klasann **TestSuite** þurfið þið ekki að eiga neitt við, en ef þið viljið búa til ný test-case megið þið breyta föllunum **initScene**, sem segir til um hve margar hæðir og lyftur eiga að vera í senunni, og **runScene** sem er keyrt í þræðinum sem stjórnar því hvar og hvenær fólk kemur inn á hæðirnar og hvert það vill fara. Við yfirferð verður þessum föllum aftur skipt út fyrir ný.

Klasinn **ElevatorMainProgram** segir sig sjálfur. Þið megið gera hvað sem þið viljið þar svo fremi það þoli að því sé öllu skipt út fyrir okkar eigin **main()** fall þegar við förum yfir.

Aftur að klasanum **ElevatorScene**. Þar eru ýmis föll sem kerfið kallar á við ákveðnar aðstæður. Yfirskriftirnar á þessum föllum mega ekki breytast en þið megið eiga við kóðann inni í þeim eins og ykkur sýnist. Einnig má bæta við föllum í þennan klasa og að sjálfsögðu má bæta við klösum alveg eins og ykkur hentar.

Mikilvægustu föllin eru tvö. 1) **restartScene** þar sem þið búið til senuna ykkar, búið til nýjan þráð fyrir hverja lyftu, búið til og endurstillið þær semafórur sem þarf og undirbúið allt sem þarf til að kerfið sé tilbúið að taka við fólki. 2) **addPerson** þar sem búinn er til nýr persónuþráður með persónu sem byrjar á ákveðinni hæð og vill fara út úr lyftu á ákveðinni hæð. Þessar hæðir koma inn í fallið **addPerson** sem færibreytur.

Að auki eru nokkur föll sem þarf að gera breytingar á til þess að þau skili réttum gildum. Skilagildin eru t.d. harðkóðuð fyrir föllin **getCurrentFloorForElevator** og **getNumberOfPeopleInElevator** sem bæði taka inn færibreytu sem segir til um fyrir hvaða lyftu er verið að biðja um gildi. Þið megið útfæra þessi föll eins og ykkur best hentar en integer gildin sem eru send inn eru indexar frá 0 upp í fjölda lyfta - 1 (0 í öllum flækjustigum nema því síðasta).

Skoðið afganginn af föllunum í klasanum **ElevatorScene** og takið afstöðu til þess hvort þeim þarf að breyta. T.d. er notaður listinn **peopleCount** sem

inniheldur heiltölu fyrir hverja hæð. Ef þið viljið nota þennan lista áfram þurfið þið að ganga úr skugga um að hann sé uppfærður rétt þegar fólk fer inn í lyftur og þegar nýtt fólk bætist við á hæð. Þið megið halda utan um þetta á þann hátt sem þið viljið en það verður að passa upp á að öll föllin, eins og t.d. **getNumberOfPeopleWaitingAtFloor** skili réttu gildi svo að grafíska viðmótið birti allt rétt.

Athugið einnig að fleiri en einn þráður munu þurfa að kalla á sum þessara falla. Passið upp á mutual exclusion fyrir hverja breytu sem þetta gæti gilt um, þ.e. að á meðan einn þráður er að skrifa í breytuna eða bæta við lista eða þess háttar, sé engin annar þráður að lesa breytuna eða skrifa. Ekki þarf að gera ráð fyrir multiple readers. Bara alltaf einn í einu að lesa og/eða skrifa er fínt.

## Virkni þráðanna

Til að útfæra þráð er fínt að búa til klasa, t.d. Elevator eða Person, sem útfærir (implements) **Runnable** skilin (interface). Síðan er hægt að smíða nýtt tilvik af **Thread** sem tekur inn nýtt tilvik af þessum klasa, og kalla á **start()** á **Thread**. Virkni þráðarins er þá útfærð í **run()** fallinu á klasanum sem útfærir **Runnable**.

Ef einhver klasi þarf að geta talað við **ElevatorScene** til að uppfæra breytur þar má búa til smið (**constructor**) á klasann sem tekur inn **ElevatorScene** sem færibreytu og setur það gildi á sitt eigið tilvik (sem er þá **reference** á aðalsenuna). Síðan má senda **this** inn í þann smið til að það sé tilvikið sem notað er. Að öðrum kosti mætti líka búa til einhvers konar globally accessible tilvik af **ElevatorScene** sem er gefið þetta sama gildi. Gangið bara úr skugga um að einungis eitt tilvik af **ElevatorScene** sé í gangi, og að það sé tilvikið sem allir klasar hafa aðgang að sem þurfa.

***Þræðir fyrir lyftur*** þurfa að vera til í kerfinu frá því kallað er á **restartScene** og þar til aftur er kallað á **restartScene**. Gott væri í upphafi þess falls að taka saman þá lyftuþræði sem eru lifandi í kerfinu (ef einhverjir eru), gefa þeim merki um að stöðva og **join**-a þá áður en gengið er í það verk að búa til þræði fyrir næstu senu. Lyftuþræðirnir þurfa að færa sig á milli hæða, upp eða niður eina hæð í einu, og stýra þeim höftum sem ráða því hvort persónuþræðir geti yfirgefið röðina sína og hvort þeir geti yfirgefið lyftuna (á réttri hæð). *Í hvert skipti sem lyfta skiptir um hæð, eftir að hafa lokið þeim aðgerðum sem hún þarf að framkvæma þegar hún kemur inn á þessa nýju hæð, verður hún að bíða í* ***VISUALIZATION_WAIT_TIME*** *millisekúndur. Þetta er gert með* ***Thread.sleep(ElevatorScene.VISUALIZATION_WAIT_TIME)*** *eins og sjá má í TestSuite klasanum. Eftir þessa bið framkvæmir lyftan þær aðgerðir sem hún þarf að framkvæma áður en hún yfirgefur hæð og skiptir síðan aftur um hæð (í þá átt sem ykkur finnst henta hverju sinni).* Þessibið er tíminn sem persónuþræðir hafa til að fara inn í lyftuna. Persónuþræðir þurfa ekki að útfæra neinn svefn sjálfir.

***Þræðir fyrir persónur*** þurfa að vera til frá því kallað er á **addPerson** og þangað til þeir hafa komist bæði inn í lyftu á hæðinni þar sem persónan byrjar og út úr lyftinni á hæðinni sem hún vill fara á. Keyrslu persónuþráðanna má einfaldlega ljúka þá. Best er að skila **Thread** tilvikinu sem keyrir persónuna í

fallinu addPerson (þegar þráðurinn er búinn til), því þá sér **TestSuite** um að bíða eftir að allir persónuþræðir ljúki keyrslu. Annars gæti næsta sena dottið beint í gang áður en allir eru komnir á sinn stað, ef verið er að keyra nokkrar senur í röð.

Persónuþráður þarf einfaldlega að bíða eftir að pláss sé fyrir hana í lyftu, bíða svo eftir að rétt lyfta hleypi henni út á réttri hæð og sjá svo um að ganga frá eftir sig, t.d. tilkynna um plássið sem losnar eða þess háttar, en það fer eftir útfærslu ykkar á semafórunum og því hver kallar á aðgerðirnar á þeim og hvenær.

***Persónuþráður verður að kalla á personExitsAtFloor(floor) með númeri hæðarinnar þar sem hann fer út úr lyftunni, um leið og honum er hleypt út úr lyftunni, áður en þráðurinn klárar keyrslu. Þannig getur viðmótið birt réttan fjölda fólks vinstra megin við lyfturnar til að sýna hverjir hafa yfirgefið þær.***

Þið ættuð að velta fyrir ykkur hvernig mismunandi partar af þessum keyrslum passa við vandamálin sem við höfum rætt í fyrirlestrum, t.d. **Mutual Exclusion** og **Producer-Consumer**, og þá hver er producer og hver er consumer og hvaða semafórur þarf til að passa upp á aðgengi að öllum aðföngum (og hver aðföngin (resources) eru). Einnig skuluð þið passa vel upp á að leysa öll mutual exclusion vandamál sem koma upp sem hliðarvandamál við aðalvandamálið, vegna aðgengis margra þráða að upplýsingunum sem notaðar eru (t.d. af grafíska viðmótinu) til að halda utan um stöðu kerfisins hverju sinni.

Athugið einnig að þó að mest af verkefninu sé hægt að leysa einungis með því að stilla semafórur, þá getur einnig gangast að halda utan um ákveðna hluti í sameiginlegum breytum í minni, aukateljurum o.fl. Í einhverjum flækjustigum má líka skoða möguleikann á aukaklösum, jafnvel aukaþráðum, sem sjá um að halda utan um og tengja saman viðeigandi gögn og þræði, aftur með teljurum eða semafórum sem hægt er að nota til að samstilla (synchronize) aðalþræðina. Öllum slíkum lausnum skal síðan lýsa í skjalinu sem fylgir, til að ganga úr skugga um að kennarar komi auga á snilld nemenda við lausn þessara flóknu vandamála.

Góða skemmtun!

# The Product

The product is a program, based on the java project included, that controls elevators, each elevator in a separate thread, that accept and release people, each person in a separate thread, in such a way that it appears correctly in the graphical visualization (included) and creates the people as instructed by the **testSuite** thread that is run automatically at the beginning of the program's execution.

In addition a revised document, the same as in part 1 of the project, that names 1) the semaphores that are made and used in the system, 2) the events

(elevator stops on floor, elevator leaves floor, person enters floor, etc.) that happen in the system and 3) what operations are called on the semaphores when these events occur. Any images and system descriptions are also welcome.

The document should also include descriptions of the problems that arise as a result of multiple threads needing access to shared memory objects and the semaphores used to solve them.

## Getting and returning the assignment

On the project website on myschool there is an archive with a Java project which serves as the base for your program. Fetch, open and rename the project with the name of at least one member of your group. Once the program is ready, archive it (zip/rar/7z) and return along with a PDF with your design document. The project can be done in groups of three (or less).

## The problem

People enter a building and wish to use an elevator to travel between floors. Each elevator is implemented as a thread and each person is implemented as a thread. At any given time the system will include a separate thread for each elevator and each person that has entered the system but not yet finished its run by exiting at the floor it wishes to travel to.

The thread should wait at the appropriate places and unlock certain lock at the appropriate times, to make sure that other threads can also conclude their run correctly.

In the design work you can assume that the elevator threads are already running and that an outside source starts the person threads and lets them know where they are and where they wish to go. You still need to design the running functionality of the threads.

Following are the different levels of complexity:

- **50%** - Single elevator, two floors, everyone enters at the bottom floor and exits at the top floor.
- *(Intermediate)* Single elevator, any number of floors, everyone enters at the bottom floor and exit at any other floor. The person-thread knows from the beginning at shich floor it wishes to exit.
- *(Intermediate)* Single elevator, two floors, persons can enter at either floor and exit at the other floor.
- **30%** - Single elevator, any number of floors, persons can enter and exit at any floor. The person-thread knows from the beginning at shich floor it wishes to exit.

- **20%** - Any number of elevators, any number of floors, persons enter and exit at any floor. Persons enter at a floor and can use any elevator to travel.

Each system can be described either for a single person in an elevator at a time, or for 6 people at a time in an elevator. You can decide when this level of complexity is added, but in the end each system should allow for 6 people in an elevator at a time.

Feel free to add any intermediate levels of complexity if this helps you get your head around the problem. You can also skip levels in the final document if you feel they are included in other levels, but take care not to add too much at a time, so you have better overview over what causes errors and problems in your system.

Bonus points:

- **5%** - Solve test case 5 in **TestSuite** without having to starve certain floors until other floors are empty.
- **5%** - Implement the idea of up/down buttons, and have people only enter elevators when they are going in the correct direction.
- **5%** - Make the entire system **thread safe**, by verifying **mutual exclusion** on all shared variables.

Comments in the project code describe what needs to be done where, and what is allowed, but also look at the following process description for better descriptions of the problem.

# Process

Before you begin work on any level of complexity, start by going through the steps in part 1 of the project for that level of complexity. It is also beneficial to look at lab 4 (dæmatímaverkefni 4) that described the Java class **Semaphore**. Also look at the library **java.util.concurrent**, if you wish to use other methods than only counting semaphores in the more complex parts of the system.

First look at the base program. In the java project there are 4 classes. Your main entry point to the system is through the class **ElevatorScene**. The class **ElevatorGraphics** should be left alone, as it is only there to display the state of your system. You don't need to add anything to the class **TestSuite**, but if you wish to make new test cases you can change the functions **initScene**, which decide how many floors and elevators are in the system, and **runScene** which is run in the thread that adds people and decide where they are added and which floor they wish to exit at. When grading, these functions will be switched out for new ones.

The class **ElevatorMainProgram** is self-explanatory. You can do whatever you want there, as long as it handles being switched out again for our own **main()** function when grading.

Now back to the class **ElevatorScene**. It contains several functions that the system calls at different times. The definitions of those functions can not be changed but you can do anything with the code inside them. You can also add functions to this class and you can of course add classes to the system to your heart's content.

The most important functions are two. 1) **restartScene** is where you make your scene, make new threads for each elevator, make and initialize the semaphores needed and prepare everything for accepting people into the system. 2) **addPerson** is where a new person-thread is made with a person that enters at a specific floor and wishes to exit at another floor. These floors are variables in the **addPerson** function.

In addition there are a few functions that need to be changed so that they return correct values. For example, the return values are hard coded for **getCurrentFloorForElevator** and **getNumberOfPeopleInElevator**, both of which take a parameter stating for which elevator a value is needed. You can implement these functions in way you wish, but the integer values for the parameter are indexes from 0 to number of elevators - 1 (0 in all levels of complexity except the last).

Look at the rest of the functions in the class **ElevatorScene** and decide whether they need to change. The list **peopleCount** holds an integer for each level. If you wish to use this list you must make sure it is updated correctly when people enter elevators and when new people enter floors. You can keep track of this in way you wish, but you must make sure that all the functions, e.g. **getNumberOfPeopleWaitingAtFloor** return the correct value so that the graphical visualization displays everything correctly.

Also realize that more than one thread may call some of these functions. Make sure that mutual exclusion is used where needed, that is, that while one thread is writing to a variable or adding to a list, no other thread is accessing it at the same time. There is no need to accommodate multiple readers. Simply one thread wither reading or writing at a time is fine.

## Functionality of threads

To implement a thread it is best to make a class, e.g. Elevator or Person, that implements the **Runnable** interface. Then you can construct a new instance of **Thread**, which takes a new instance of your class as a parameter, and call **start()** for the thread. The functionality of the thread is then implemented in the **run()** function of the class implementing **Runnable**.

If any class needs to access **ElevatorScene** to update variables there you can make a constructor on that class that takes **ElevatorScene** as a parameter and sets its own instance to that (which is a **reference** to the main scene). You can then send **this** into the constructor. Alternatively you can make a globally accessible instance of **ElevatorScene** that gets that same value. Just make sure that only one instance of **ElevatorScene** is used at any time, and that it is the instance that all classes needing it have access to.

***Threads for elevators*** need to live in the system from the time **restartScene** is called and until **restartScene** is called again. It would be good, in the beginning of that function, to collect all currently living elevator threads (if any), give them a signal to stop and then **join()** them before beginning to build the threads needed for the next scene. The elevator threads must move between floors, up or down one at a time, and control the locks or constraints that make sure people only enter and exit at the correct time (and at the correct floor). *Each time an elevator switches floors, after finishing any operations needed when it enters a new floor, it must wait **VISUALIZATION_WAIT_TIME** milliseconds. This is done by calling **Thread.sleep(ElevatorScene.VISUALIZATION_WAIT_TIME)** as can be seen in the **TestSuite** class. After this wait the elevator performs any operations needed before leaving a floor and then switches floors again.* This wait time is the time person-threads have to enter the elevator. Person-threads do not need to implement any sleeping of their own.

***Threads for persons*** need to exist from the time **addPerson** is called until they have both entered an elevator and exited it at the correct floor. The person-thread can simply finish its run then. You should return the instance of **Thread** that runs the person in the function **addPerson** (when the thread is created). That way **TestSuite** will handle waiting for every thread to finish, so that the next scene doesn't suddenly start before everything was finished in the previous one, in case several scenes are being run sequentially.

A person-thread simply needs to wait for there being room for it in an elevator, then to wait for the correct elevator to let it out at the correct floor and finally call any operations, possibly to let know that the room is free again, but this depends on your implementation of the semaphores, and who calls which operation to lock/unlock them, and when.

***A person thread must call personExitsAtFloor(floor) with the correct value of exit floor when it's let out of the elevator, before it finishes. That way people will be shown after they have exited the elevators.***

You should think about how different parts of this problem fit with the problems we have discussed in lectures, e.g. **Mutual Exclusion** and **Producer-Consumer**, and then what is the producer and what the consumer and what semaphores are needed to control access to all resources (and which resources they are). You should also make sure to solve all mutual exclusion problems that may come up as side effects to the main problem, due to multiple threads needing access to the same information (e.g. the graphical visualization) to keep track of the state of the system at each time.

Take note that even though big parts of the project can be solved by setting semaphores, it can also be beneficial to keep track of certain things in shared variables in memory, extra counters and such. In some levels of complexity you can also look at the possibility of making extra classes, even extra threads, that keep track of and connect certain data and threads, again with counters or semaphores that can be used to synchronize the main threads. All such solutions shall then also be described in the accompanying document, to make sure that teachers spot students' brilliance in solving these complex problems.

Have a good time!