The University of Melbourne
Department of Computing and Information Systems
SWEN90004 Modelling Complex Software Systems

Assignment 2a, 2016

Released: 26 April. Deadline: 14 May at 23:00

Objectives

To provide programming practice in a concurrent programming language and to get a better understanding of safety and liveness issues.

Background and context

There are two parts to Assignment 2. This first part, 2a, is worth 10% of your final mark; the second part, 2b, will be worth 15%.

This first part of the assignment deals with programming threads in Java. Your task is to implement and test a simulator of an automated car park system where cars are moving through a parking space and going into a lift in the process. A lift is a device that connects the Ground and first floors. This lift can be moved up and down using jacks in order to raise cars that want to move to the first floor and lower the ones that want to move to the ground floor. There are gates at either end of the lift.

The system to simulate

The system to be built is a simulator of an automatic car park with a lift. Figure 1 shows the layout of the automated car park system. The parking space is somehow at the appropriate level by a mechanism that we don't worry about. It only allows one-way traffic and is divided into a number of sections (six in the example). A car park section can not be occupied by more than one car at a time. A car arrives from the ground floor at the lift and, provided the lift is unoccupied and it is at the ground level, the car can enter the lift and be raised, allowing it to then travel through the sections 1, 2, 3,... Eventually it will arrive back at the lift and, provided the lift is unoccupied and it is at level one, the car can enter the lift and be lowered, allowing it to depart the system. Figure 1 shows a snapshot: car 9 occupies section 2 and car 8 occupies section 6, where it is waiting for the lift to become unoccupied, so that it can depart. However, car 10 is currently in the lift; the sub-index "in" indicates that car 10 is inbound, being raised, so that it can enter section 1 of the car park.

Cars arrive from the ground level, at random times, under their own speed. However, they need to be towed from the lift into the first section, from one section to the next, and from the last section back into the lift. For this, three types of vehicles are used. A special "launch" vehicle transfers cars from the lift to the first car park section. A special "return" vehicle transfers cars from the last section back to the lift. A number of ordinary vehicles tow cars from one section to the successor section. Each ordinary vehicle is committed to serving the same pair of sections continually, picking up from section k and delivering to section $k + 1$.

In the snapshot of Figure 1, once car 10 has entered the car park, the lift is at the right level for car 8 to enter the lift and be lowered. However, if there was no car 8 waiting to depart, and

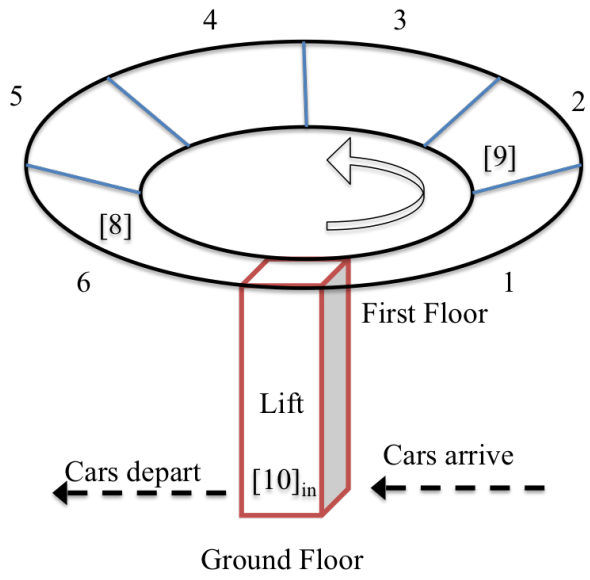

Figure 1: An automated car park system, floors connected by a lift

instead more cars were arriving, the lift should be lowered to allow the arriving cars to enter the lift. Hence it is sometimes necessary to move the lift up or down even if there are no cars in the lift. It is the task of an "operator" to occasionally move the lift up and down like that.

The task

Your task is to implement a simulator for the car park/lift system. It should be suitably parameterised, so that

- timing assumptions can be varied, and
- the number of sections can be varied.

The simulator should produce a trace of events, similar to the example below. Note that after the event "`[10] enters lift to go up`" we have Figure 1's snapshot.

```
:
[9] enters section 2
[10] enters lift to go up
[10] leaves the lift
[8] enters lift to go down
[8] departs
[9] leaves section 2
[10] enters section 1
[10] leaves section 1
[9] enters section 3
[9] leaves section 3
[10] enters section 2
[10] leaves section 2
[9] enters section 4
[9] leaves section 4
lift goes up
[10] enters section 3
[10] leaves section 3
[9] enters section 5
[10] enters section 4
[9] leaves section 5
[10] leaves section 4
[9] enters section 6
[9] leaves section 6
[10] enters section 5
[10] leaves section 5
lift goes down
[11] enters lift to go up
[11] leaves the lift
[9] enters lift to go down
[9] departs
lift goes up
[10] enters section 6
[10] leaves section 6
[11] enters section 1
[11] leaves section 1
[11] enters section 2
[11] leaves section 2
[10] enters lift to go down
[10] departs
[12] enters lift to go up
:
```

A possible design and suggested components

In the Java context it makes sense to think of each section as a monitor, and similarly to consider the lift as a monitor. As possible set of active processes would then be:

`Producer`: Generates new cars that want to enter; hands them to the lift, subject to the constraint that the lift is unoccupied and it is also at ground level. The times between arrivals should vary.

`Consumer`: Removes cars that are ready to depart, that is, the lift has an outbound car and it is at ground level. The times between departures should vary.

`Launch_vehicle`: Picks up cars from the lift and tows them to the first section, subject to the constraints that there is an inbound car in the lift and the first section is unoccupied. The launch vehicle may need to wait for these conditions to be satisfied.

`Vehicle[i]`: Picks up cars from section `i` and delivers them to the successor section, subject to the obvious constraints.

`Return_vehicle`: Picks up cars from the last section and tows them to the lift, subject to the constraints that the last section is occupied, the lift is unoccupied, and it is at Level 1.

`Operator`: Inspects the lift at random intervals and, once the lift is unoccupied, changes it's level, that is, ground or first floor.

You do not need to follow this design sketch. We have made some scaffolding available on the LMS, and that does assume the design outlined above, but you can adapt it as you see fit (or not use it at all). The components that we have provided are:

`Car.java`: Cars can be generated as instances of this class.

`Param.java`: A class which, for convenience, gathers together various system-wide parameters, including time intervals.

`Main.java`: The overall driver of the simulation.

Timing parameters

The class `Param.java` assumes the following time parameters are of interest:

`arrivalLapse()`: The time lapsed between two successive arrivals (varies).

`departureLapse()`: The time lapsed between two successive departures (varies).

`operateLapse()`: The time lapsed between two successive moving up/down operations (varies).

`OPERATE_TIME`: The time it takes to move the lift to ground or first floor.

`TOWING_TIME`: The time it takes a vehicle to take a car from its source to its destination.

Varying these parameters will give different system behaviours—you are encouraged to experiment with the settings.

Extension tasks

A number of optional extensions are suggested, ranging from the simple to the more ambitious:

1. Time between arrivals and departures should arguably follow an exponential, rather than uniform, distribution.

2. To complement the trace output, it would be useful to have a sequence of (textual) snapshots of the whole system, showing the content of the lift and each section (and any other relevant information) after each change. This could be generated from a trace, by post-processing. A greater challenge is to have the simulator output these snapshots, rather than the trace, as the simulation is happening.

3. As a more ambitious variant of the previous extension, graphical output could be produced, so that a diagram similar to Figure 1 is kept updated while the simulator is running.

4. Provide a second implementation in go and write a short essay that compares Java and go for the task.

Procedure and assessment

The project should be done by students individually. On the LMS you will find a zip file containing the bits of scaffolding mentioned above.

The submission deadline is Saturday 14 May at 23:00. A late submission will attract a penalty of 1.5 marks for every calendar day it is late. If you have a reason that you require an extension, email Lida *well before the due date* to discuss this. Submit a single zip file via LMS. The file should include

- All Java source files needed to create a file called `Main.class`, such that '`java Main`' will start the simulator.

- A makefile that will generate `Main.class` (it may be very simple; perhaps just containing the action "`javac *.java`").
- A file called `reflection.txt` with 400–600 words evaluating the success or otherwise of your solution, identifying critical design decisions or problems that arose, and summarising any insights from experimenting with the simulator.

I encourage the use of the LMS's discussion board for discussions about the project. However, all submitted work is to be your own individual work.

This project counts for 10 of the 50 marks allocated to project work in this subject. Marks will be awarded according to the following guidelines:

Criterion	**Description**	**Marks**
Design	The code is well designed and potentially extensible, and shows understanding of concurrent programming concepts and principles.	3 marks
Correctness	The code runs and generates output that is consistent with the specification.	4 marks
Structure & style	The code is well structured and readable.	1 marks
Layout	The code adheres to the code format rules (Appendix A) and in particular is well commented and explained.	1 marks
Reflection	The reflection document demonstrates engagement with the project.	1 marks
Total		10 marks

Harald Søndergaard and Lida Rashidi
26 April 2016

A Code format rules

Your implementation must adhere with the following simple code format rules:

- Every Java class must contain a comment indicating its purpose.
- Every method must contain a comment at the beginning explaining its behaviour. In particular, any assumptions should be clearly stated.
- Constants, class, and instance variables must be documented.
- Variable names must meaningful.
- Significant blocks of code must be commented.

  However, not every statement in a program needs to be commented. Just as you can write too few comments, it is possible to write too many comments.
- Program blocks appearing in if-statements, while-statements, etc., must be indented consistently. They can be indented using tabs or spaces, and can be indented 2, 4, or 8 spaces, as long as it is done consistently.
- Each line should contain no more than 80 characters.