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

Assignment 1a, 2019

Released: Tuesday 12 March, 2019.
Deadline: 23:59, Thursday 28 March, 2019

Objectives

The objectives of this assignment are: to convert a description of a system into a simulation model of that system; to implement that simulation in a shared memory concurrent programming language; to use the implemented simulation to explore the behaviour of a complex system; to gain a better understanding of safety and liveness issues in concurrent systems.

Background and context

There are two parts to Assignment 1 (a and b), each worth 10% of your final mark. This first part of the assignment deals with programming threads in Java. Your task is to implement a concurrent simulation of cargo ships docking at a spaceport.

The system to simulate

The ***USS Emafor*** is a space station located near Ceres in the asteroid belt between Mars and Jupiter. The Emafor serves as a hub for the processing and distribution of ore mined from nearby asteroids. The raw ore is delivered by **cargo ships** that arrive at an **arrival wait zone**, approximately ten kilometres from the Emafor. At this point, a local **pilot** is transported to the cargo ship to safely handle the approach, docking, unloading of cargo, undocking and departure.

Docking and undocking of cargo ships at the USS Emafor's **berth** require the assistance of several smaller spacecraft known as **space tugs**. The berth can only hold one cargo ship at a time. Before they commence docking a cargo ship, a pilot must have engaged three tugs. These tugs are used during the docking process, and released once unloading of cargo commences. That is, while cargo is being unloaded the tugs may be utilised by other ships. Before commencing undocking, a pilot must have engaged two tugs (undocking is a simpler procedure), which are released once undocking is complete.

Following undocking, the local pilot is transported off the cargo ship, the cargo ship returns to a **departure wait zone**. The pilot is then transported off and the cargo ship departs the region of the USS Emafor.

The USS Emafor employs a tug controller, whose job it is to coordinate requests from pilots and allocate the requested number of tugs once they are available. A pilot can therefore contact the controller *once* to acquire or release the services of *multiple* tugs.

One hazard associated with mining in the asteroid belt is the risk of collisions with space debris. The USS Emafor's berth is equipped with a **shield** which can be activated to protect the berth at times of high risk. While the shield is activated, no cargo ships can start docking or undocking from the berth until after the shield is deactivated. Ships that have already started the docking or undocking process when the shield is activated can safely complete their action. The shield is activated and deactivated by a shield operator.

The full system is shown in Figure 1.

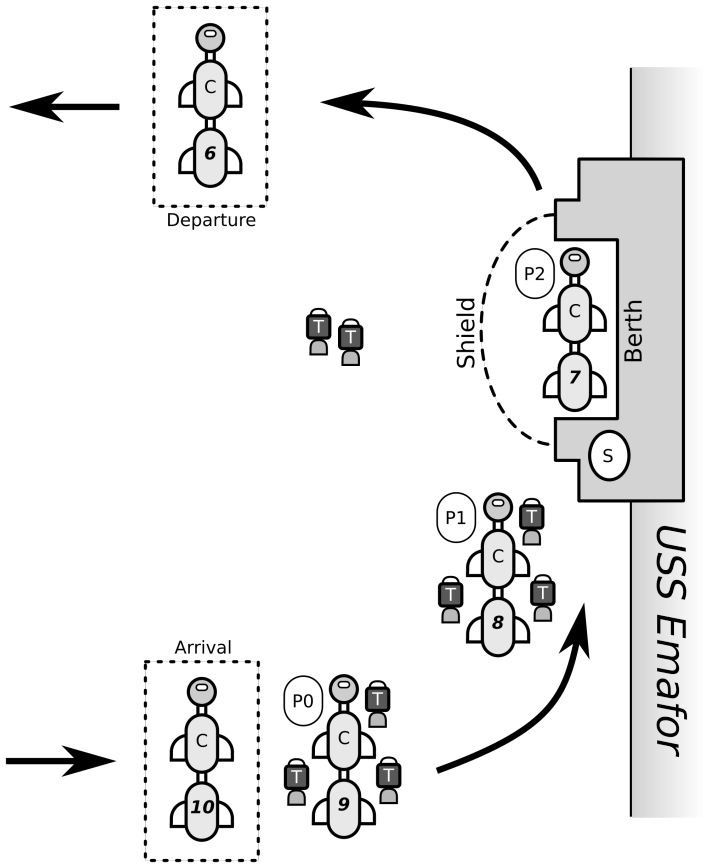

Figure 1: The cargo ship docking system described above. At this point (corresponding to the point in time "`ship [7] being unloaded`" in the trace below) ship 7, under control of pilot 2 is docked in the berth and currently being unloaded. Ship 6 has already been unloaded, returned to the departure zone, and is now ready to leave. Ships 8 and 9 have been acquired by pilots 1 and 0 respectively, each of whom has also acquired the 3 tugs necessary to dock their ships, once the berth becomes available. Ship 10 has recently arrived in the arrival zone and is awaiting a pilot. The shield has been activated, meaning that no ships can dock or undock until it is deactivated.

Your tasks

Your task is to implement a simulator for the cargo ship docking system described above. It should be suitably parameterised such that the timing assumptions can be varied, and the number of pilots and tugs can be varied. You shuold use your simulator to explore the relationship between the number pilots and tugs to identify any potential problems with the system.

You should assume:

- that having engaged some number of tugs, a pilot will only release them once they have completed their current docking/undocking procedure;
- that a pilot will never request or engage the services of more tugs than it requires to complete its current docking/undocking procedure;
- that having completed their current docking/undocking procedure, a pilot will release the services of all tugs that it has engaged.

The simulator should produce a trace of events matching that below. Note that Figure 1 corresponds to the state of the system immediately after "`ship [7] being unloaded`", near the bottom of the right column.

```
     :
pilot 2 acquires ship [7].
pilot 2 acquires 3 tugs (2 available).
ship [8] arrives at arrival zone
Shield is activated.
ship [5] being unloaded.
pilot 1 acquires 2 tugs (0 available).
Shield is deactivated.
ship [5] undocks from berth.
ship [6] docks at berth.
pilot 0 releases 3 tugs (3 available).
pilot 1 releases ship [5].
pilot 1 releases 2 tugs (5 available).
ship [5] departs departure zone
pilot 1 acquires ship [8].
pilot 1 acquires 3 tugs (2 available).
ship [9] arrives at arrival zone
ship [6] being unloaded.
pilot 0 acquires 2 tugs (0 available).
ship [6] undocks from berth.
ship [7] docks at berth.
pilot 2 releases 3 tugs (3 available).
pilot 0 releases ship [6].
pilot 0 releases 2 tugs (5 available).
pilot 0 acquires ship [9].
ship [6] departs departure zone
ship [10] arrives at arrival zone
pilot 0 acquires 3 tugs (2 available).
Shield is activated.
ship [7] being unloaded.
pilot 2 acquires 2 tugs (0 available).
Shield is deactivated.
ship [7] undocks from berth.
      :
```

A possible design and suggested components

In the context of Java, it makes sense to think of each location (wait zones and berths) as a monitor, and to think of the tugs (collectively) as a monitor. A possible set of active processes would then be:

`Producer:` Generates new cargo ships arriving at the USS Emafor wait zone, subject to room being available. The times between arrivals should vary.

`Consumer:` Removes cargo ships who have finished unloading, once they have undocked and returned to the wait zone. Departing ships leave the wait zone immediately.

`Pilot:` Acquires a newly arrived cargo ship, acquires the required number of tugs to dock the ship. After the ship is unloaded, acquires tugs for undocking.

`Operator:` Periodically activates the shield to protect the space station from space debris.

We have made some scaffold code available on LMS that follows this outline described above. The components we have provided are:

`Producer.java` **and** `Consumer.java:` as described above.

`Ship.java:` Cargo ships can be generated as instances of this class.

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

`Main.java:` The overall driver of the simulation. Note that this won't compile until you have defined some additional classes.

System parameters:

The class `Params.java` contains the system parameters, including the number of pilots and tugs, the numbers of tugs required for docking and undocking procedures, and a number of timing parameters. Varying these parameters will give you different system behaviours. Once you have a working simulator, you should experiment with the parameter settings.

Procedure and assessment

- The project should be done by students **individually**.
- Late submissions will attract a penalty of 1 mark for every *calendar* day it is late. If you have a reason that you require an extension, email Nic *well before the due date* to discuss this.
- You should submit a single zip file via LMS. The zip file should include:
  1. A single directory containing all Java source files needed to create a file called `Main.class`, such that "`javac *.java`" will generate `Main.class`, and "`java Main`" will start the simulator.
  2. A plain text file `reflection.txt` should contain approximately 500 words that discusses the potential problems that can arise in this system, drawing on observations of your simulator behaviour. You could also use this text to evaluate the success or otherwise of your solution, identify critical design decisions or problems that arose, and summarise any other insights from experimenting with the simulator. Please ensure that this is a *plain text* file; ie, not a `doc`, `docx`, `rtf`, or other file type that requires specific software to read.

  All source files and your text file should contain, near the top of the file, your name and student number.
- We encourage the use of the LMS discussion board for discussions about the project. **However, all submitted work must be your own individual work.**
- This project counts for 10 of the 40 marks allocated to project work in this subject. Marks will be awarded according to the following guidelines:

Criterion	**Description**	**Marks**
Understanding	The submitted code is evidence of a deep understanding of concurrent programming concepts and principles.	3 marks
Correctness	The code runs and generates output that is consistent with the specification.	2 marks
Design	The code is well designed, potentially extensible, and shows understanding of concurrent programming concepts and principles.	2 marks
Structure & style	The code is well structured, readable, adheres to the code format rules (Appendix A), and in particular is well commented and explained.	2 marks
Reflection	The reflection document demonstrates engagement with the project.	1 marks
Total		10 marks

Nic Geard
11 March 2019

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 must contain no more than 80 characters.