School of Electrical Engineering and Computing
COMP2240/COMP6240 - Operating Systems
Assignment 2 (15%)

Submit using Blackboard by **23:59, 12th October (Friday), 2018**

Problem 1: Sharing the Bridge

A new single lane bridge is constructed to connect the North Island of New Zealand to the South Island of New Zealand. Farmers from each island use the bridge to deliver produce to the other island, return back to their island and this is repeated indefinitely. It takes a farmer carrying produce 20 steps to cross the bridge. Once a farmer (say a North Island farmer identified as `N_Farmer1`) crosses the bridge (from North Island to South Island) he just attempts to cross the bridge in the opposite direction (from South Island to North Island) and so on. Note that the ID of the farmer does NOT change.

The bridge can become deadlocked if a northbound and a southbound farmer are on the bridge at the same time (New Zealand farmers are stubborn and will not back up). The bridge has a large neon sign above it indicating the number of farmers that have crossed it in either direction. The neon sign counts multiple crossing by the same farmer.

Using **semaphores**, design and implement an algorithm that prevents deadlock. Use **threads** to simulate multiple/**concurrent** farmers and assume that the stream of farmers are constantly attempting to use the bridge from either direction. Your program should input parameters at runtime to initialise the number of farmers from each direction. For example `[N=5, S=5]` would indicate a constant stream of 5 farmers from each direction wanting to use the bridge. You also make sure that the solution is starvation-free (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa should not occur).

Sample Input/output for Problem 1.

The input will be as follows:

```
N=2, S=2
```

The input indicates the program is initialized with 2 farmers from each direction who will constantly attempt to use the bridge to go from one island to other.

The output (partial) from one execution is as follows:

```
N_Farmer1: Wating for bridge. Going towards South
S_Farmer2: Wating for bridge. Going towards North
S_Farmer1: Wating for bridge. Going towards North
N_Farmer2: Wating for bridge. Going towards South
N_Farmer1: Crossing bridge Step 5.
N_Farmer1: Crossing bridge Step 10.
N_Farmer1: Crossing bridge Step 15.
N_Farmer1: Across the bridge.
NEON = 1
S_Farmer2: Crossing bridge Step 5.
N_Farmer1: Wating for bridge. Going towards North
S_Farmer2: Crossing bridge Step 10.
S_Farmer2: Crossing bridge Step 15.
S_Farmer2: Across the bridge.
NEON = 2
N_Farmer2: Crossing bridge Step 5.
N_Farmer2: Crossing bridge Step 10.
N_Farmer2: Crossing bridge Step 15.
S_Farmer2: Wating for bridge. Going towards South
N_Farmer2: Across the bridge.
NEON = 3
S_Farmer1: Crossing bridge Step 5.
N_Farmer2: Wating for bridge. Going towards North
S_Farmer1: Crossing bridge Step 10.
S_Farmer1: Crossing bridge Step 15.
S_Farmer1: Across the bridge.
NEON = 4
N_Farmer1: Crossing bridge Step 5.
N_Farmer1: Crossing bridge Step 10.
S_Farmer1: Wating for bridge. Going towards South
N_Farmer1: Crossing bridge Step 15.
N_Farmer1: Across the bridge.
NEON = 5
…
…
…
```

NOTE: For the same input the output may look somewhat different from run to run.

Problem 2: Ice-cream time

A new ice-cream parlour has been opened at Shortland Student Hub. The parlour does not offer take away service but there is only five seats in the parlour to eat in. A customer may arrive to the parlour at any time and may take a certain time to finish his ice-cream. The manager chooses a peculiar rule to serve his customers. If a customer arrives when there is an empty seat, then the customer can immediately take a seat. However, if all the five seats are occupied (i.e. there are five customers enjoying their ice-creams at any instant) then all the arriving customers have to wait for the entire party (all current customers) to leave before they can get their seats.

Using **semaphores** design and implement an algorithm that manages the customers entering and leaving the ice-cream parlour in line with manager's rules. Use **threads** to simulate multiple/**concurrent** customers. Your solution must be fair - starvation free. Assume no time is wasted in taking seat, serving/starting eating ice-cream and leaving the parlour.

The input will be as follows:

```
0 C1 5
0 C2 7
0 C3 8
2 C4 5
3 C5 5
4 C6 3
7 C7 5
10 C8 5
END
```

Where each line, except the last, contains information regarding a customer
Arrival-time Customer-ID Ice-cream-eating-time
Input ends with a line containing the word END.

You can assume that in the input the customers will be sorted in order of their arrival times.

The output should be as follows:

```
Customer    arrives      Seats    Leaves
C1              0          0         5
C2              0          0         7
C3              0          0         8
C4              2          2         7
C5              3          3         8
C6              4          8        11
C7              7          8        13
C8             10         10        15
```

Output shows information of each customer in a separate line. Each line contains Customer-ID, Arrival-time, time when the customer seats in the ice-cream parlour and time when the customer leaves the parlour.

Problem 3 : Hot or Iced Coffee?

School of EEC, UoN bought a coffee machine that can serve both iced and hot coffee for the staffs. The coffee machine has three dispensing heads which can serve three clients (staffs) in parallel. We call a staff Hot Client (H) if (s)he is looking for hot coffee and a Cold Client (C) if (s)he is looking for cold coffee. However, the machine can operate in either of its two modes (hot or cold drink) at a time. If a Hot Client has started to make coffee in the machine then the other two vacant dispersers can serve hot coffee only – a Cold Client must wait. A client (Hot or Cold) can choose the brew strength by choosing the brew time at the beginning. So the assumptions in operating the coffee machine are

- Hot and Cold clients cannot use the machine at the same time.
- No more than three clients can use the machine simultaneously.
- Each client can choose different time to brew his coffee.

Using **monitor**, design and implement an algorithm that ensures the operation of the coffee machine according to the above characteristics. Use **threads** to simulate multiple/**concurrent** clients. Your solution should be fair – stream of Hot Clients should not prevent a Cold Client from using the coffee machine or vice versa. A Hot Client with ID y (i.e. Hy) should not be served before a HOT Client with ID x (i.e. Hx) where x < y. And the same for the cold clients.

The input will be as follows:

```
9
H1 4
H2 5
H3 3
C1 5
C2 3
C3 2
C4 2
H4 3
H5 2
```

Where the first line contains the number of clients looking for coffee and each line contains information about each client of the form

Client-ID Brew-Time

Client-ID: The first character is H or C indicating hot (H) or cold (C) coffee client, a number (without no blank in-between) indicating the client ID in each group. Client-IDs are unique.

Brew-Time: The time to brew his coffee.

The order of clients in the input file indicates the order in which they arrived for coffee.

The output should be as follows:

```
(0) H1 uses dispenser 1 (time: 4)
(0) H2 uses dispenser 2 (time: 5)
(0) H3 uses dispenser 3 (time: 3)
(5) C1 uses dispenser 1 (time: 5)
(5) C2 uses dispenser 2 (time: 3)
(5) C3 uses dispenser 3 (time: 2)
(7) C4 uses dispenser 3 (time: 2)
(10) H4 uses dispenser 1 (time: 3)
(10) H5 uses dispenser 2 (time: 2)
(13) DONE
```

Each line contains information about the usage of the coffee machine by a client. First, the time the client starts using the coffee machine in parenthesis. Then his Client-ID the disperser number in which he is operating and his brew time in parenthesis.

Last line shows the time to serve all the clients.

Problem 4: Share the Bridge in Pair [ONLY for COMP6240 students]

The single lane bride in Problem 1 has been upgraded to two lanes allowing two farmers to travel simultaneously. In order to share the toll, farmers cross the bridge only in pairs – a farmer will never cross the bridge alone. In order to avoid possible conflicts, either two North Island farmers or two South Island farmers can cross the bridge in the same direction. Farmers from each island use the bridge in pairs to deliver produce to the other island. As before, it takes a farmer 20 steps to cross the bridge. Once two farmers (say two North Island farmers identified as `N_Farmer3` and `N_Farmer4`) cross the bridge (from North Island to South Island) they will attempt to cross the bridge in the opposite direction (from South Island to North Island), however, the farmers may pair with other North Island farmers (e.g. `N_Farmer3` may pair up with `N_Farmer2` if `N_Farmer2` was waiting in the South Island for the bridge/pair）.

The bridge can become deadlocked if two northbound and two southbound farmers are on the bridge at the same time as the stubborn farmers will not back up. The large neon sign indicates the number of farmers that have crossed it in either direction and will count multiple crossing by the same farmer.

Using **semaphores**, design and implement an algorithm that prevents deadlock. Use one **thread** to represent each farmer and simulate multiple/**concurrent** farmers and assume that the stream of farmers attempt to use the bridge in pairs from both directions. Your program should input parameters at runtime to initialise the number of farmers from each direction. For example, the input `[N=5, S=5]` would indicate a group of 5 farmers from each direction want to use the bridge. You also make sure that the solution is starvation-free (i.e. the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa should not occur).

Sample Input/output for Problem 4.

The input will be as follows:

```
N=2, S=2
```

The input indicates the program is initialized with 2 farmers from each direction who will constantly attempt to use the bridge to go from one island to other.

The output (partial) from one execution is as follows:

```
N_Farmer1: Wating for bridge. Going towards South
S_Farmer2: Wating for bridge. Going towards North
S_Farmer1: Wating for bridge. Going towards North
N_Farmer2: Wating for bridge. Going towards South
S_Farmer1: Crossing bridge Step 5.
S_Farmer2: Crossing bridge Step 5.
S_Farmer2: Crossing bridge Step 10.
S_Farmer1: Crossing bridge Step 10.
S_Farmer2: Crossing bridge Step 15.
S_Farmer2: Across the bridge.
NEON = 1
S_Farmer1: Crossing bridge Step 15.
S_Farmer1: Across the bridge.
NEON = 2
N_Farmer2: Crossing bridge Step 5.
N_Farmer1: Crossing bridge Step 5.
N_Farmer1: Crossing bridge Step 10.
N_Farmer1: Crossing bridge Step 15.
N_Farmer1: Across the bridge.
NEON = 3
N_Farmer2: Crossing bridge Step 10.
N_Farmer2: Crossing bridge Step 15.
N_Farmer2: Across the bridge.
NEON = 4
…
…
```

NOTE: For the same input the output may look somewhat different from run to run.

Programming Language:

The preferred programming language is Java.
If you wish to use any language other than the preferred programming language, you must first notify the course demonstrator.

User Interface:

There are no marks allocated for using or not using a GUI – the choice is yours.

Input and Output:

Your program should accept data from an input file of name specified either as a command line argument (for non-GUI solutions) or using a file dialogue (for GUI solutions). The sample inputs are shown above to demonstrate the expected formatting for inputs. Your submission will be tested with the above data and with other input files.

Your program should output to standard output (for non-GUI solutions) or to a text area (for GUI solutions). Output should be **strictly** in the shown output format.

Deliverable:

1. For each problem, the program source code and a README file containing any special instructions required to compile and run the source code. If programmed in Java, your main class should be c9999999A2Px (where c9999999 is your student number, A2 for assignment 2 and Px is the problem number i.e. P1, P2 or P3) i.e. your program for the first problem can be executed by running "java c9999999A2P1". If programming in other languages, your code should compile to an executable named "c9999999A2Px".

2. Brief 1 page (A4) review (report = 10%) of the how you tested your programs to ensure they enforced mutual exclusion and are deadlock and starvation free.

Please place all files and the README file for each problem in a separate folder. There should be three folders for the solutions of three problems. Submit all folders, files and the 1 page report plus a copy of the official assignment cover sheet in a ZIPPED folder through Blackboard. The folder name should be "c9999999A2" and the zip file should be name as c9999999A2.zip (where c9999999 is your student number).

NOTE: Assignments submitted after the deadline **(11:59 pm Friday 12th October 2018)** will have the maximum marks available reduced by 10% per 24 hours.

Mark Distribution:

Mark distribution can be found in the assignment feedback document (Assign2Feedback2240.pdf/ Assign2Feedback6240.pdf). There are no marks allocated for using or not using a GUI – the choice is yours.

Threads:

You will use threads to simulate the concurrent elements. To get you started, have a look at the following tutorials:

For Java: http://docs.oracle.com/javase/tutorial/essential/concurrency/
For C/C++: https://computing.llnl.gov/tutorials/pthreads/