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

Submit using Blackboard by **23:59, 6th October (Friday), 2017**

Problem 1: Sharing the Bridge (30%)

A single lane bridge connects 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 streams of farmers are constantly attempting to use the bridge 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 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: Share the Bridge in Pair (25%)

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 carrying produce 20 steps to cross the bridge. Once two farmers (say two North Island farmers identified as `N_Farmer1` and `N_Farmer2`) cross the bridge (from North Island to South Island) they will NOT attempt to cross the bridge in the opposite direction (from South Island to North Island).

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.

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 only once to go to the other island. 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 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 2.

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 attempt to use the bridge to go from one island to other only once.

The output 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.

Problem 3 : Colour or Monochrome Print? (20%)

School of Electrical Engineering and Computing, UoN bought a new multi-printer that can print both in colour and in monochrome. The multi-printer has three printing heads which can print up to three jobs in parallel. We classify a job as either Monochrome (M) or Colour (C) based on its mode of printing. However, the printer can operate in either of its two modes (Monochrome or Colour) at a time. If a Monochrome job is printing in the printer then the other two vacant printing heads can be used for Monochrome printing only – a Colour printing job must wait. A printing job (Monochrome or Colour) must specify beforehand the number of pages it will be printing. So the assumptions in operating the multi-printer are

- Monochrome and Colour jobs cannot be printed at the same time.
- No more than three jobs can use the printer simultaneously.
- Printing a single page takes the same time (1 sec) in all jobs.
- Each job can have different pages to print, therefore, can take different time to print.
- A Monochrome job with ID y (i.e. My) should NOT be served before a Monochrome job with ID x (i.e. Mx) where x < y. And the same for the Colour job.

Using **monitor**, design and implement an algorithm that ensures the operation of the multi-printer according to the above characteristics. Use **threads** to simulate multiple/**concurrent** printing jobs. Your solution should be fair – stream of Monochrome printing jobs should not cause the Colour Printing jobs wait forever or vice versa.

The input will be as follows:

```
9
M1 4
M2 5
M3 3
C1 5
C2 3
C3 2
C4 2
M4 3
M5 2
```

Where the first line contains the number of jobs to be processed and each line contains information about each job of the form
Job-ID Number-of-Pages
Job-ID: The first character is M or C indicating monochrome (M) or colour (C) job, a number (without no blank in-between) indicating the job ID in each job-group. Job-IDs are unique.
Number-of-Pages: The number of pages to print in that job. It is the same as the time in seconds to print this job.

The output should be as follows:

```
(0) M1 uses head 1 (time: 4)
(0) M2 uses head 2 (time: 5)
(0) M3 uses head 3 (time: 3)
(5) C1 uses head 1 (time: 5)
(5) C2 uses head 2 (time: 3)
(5) C3 uses head 3 (time: 2)
(7) C4 uses head 3 (time: 2)
(10) M4 uses head 1 (time: 3)
(10) M5 uses head 2 (time: 2)
(13) DONE
```

Each line contains information about the usage of the printer by a job. First, the time the job starts in the printer is shown in parenthesis. Then follows the Job-ID, the printer head number in which the job is printed and its required time (Number-of-pages) in parenthesis.
Last line shows the time to finish all the jobs.

Programming Language:

The preferred programming language are Java, C (gcc), C++ (g++).
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. **If output is not generated in the required format then your program will be considered incorrect.**

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 6th October 2017)** 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). There are no marks allocated for using or not using a GUI – the choice is yours.

Threads/Concurrency:

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/
Concurrency: http://greenteapress.com/wp/semaphores/