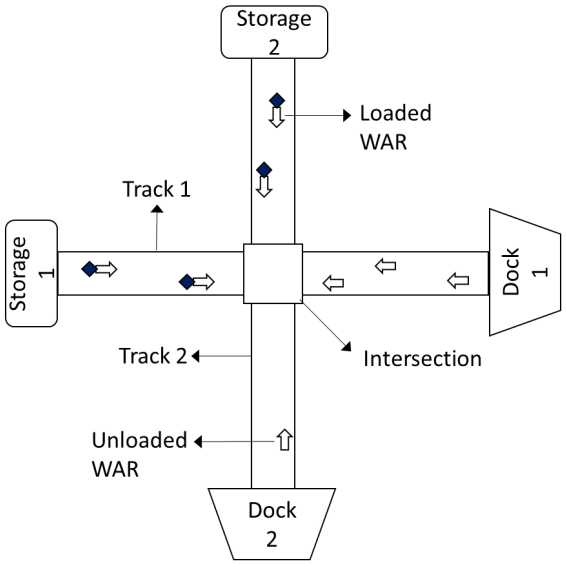
School of Information and Physical Sciences
COMP2240/COMP6240 - Operating Systems
Assignment 2 (15%)

Submit using Blackboard by **23:59, 17th September (Friday), 2021**

Problem 1: WAR controlling:

Daintree warehouse uses small robots called WARs (Warehouse Assistance Robot) for carrying its goods between storage and docks for loading and unloading. The partial layout of the warehouse with two outbound docks and two storage sections is shown below. Track 1 and Track 2, which connect Storage 1 with Dock 1 and Storage 2 with Dock 2, respectively, intersect with each other.



WARs are simple robots which repeatedly travel back and forth in the same track, being 'Loaded' in their trip from Storage to Dock and return 'Unloaded' in the return trip. Each WAR has a unique number for identification but its status changes between 'Loaded' and 'Unloaded' from trip to trip. After initial setup, it was immediately identified that WARs operating in both tracks can collide in the intersection, therefore, Daintree employed you to solve the problem. The assigned task is as follows:

1. The intersection can become deadlocked if two WARs from different directions enter the intersection (WARs can only go forward). Collison is evident if more than one WAR enter the intersection simultaneously. So, for safety purpose, not more than one WAR is allowed to enter the intersection.
2. The solution should be starvation free. I.e. a stream of WARs from any one direction should not prevent WARs from other directions to cross the intersection.
3. There are three checkpoints (sensors) in the intersection to which each WAR reports. Add 200 ms delay after each checkpoint to simulate the time to pass the checkpoint. The system keeps track of WARs passing in each track (i.e. counts every time a WAR in a track crosses the intersection).
4. Loading and unloading of goods happen instantly (i.e. the assumption is no time is required) after a WAR crosses the intersection.

Using **semaphores**, design and implement an algorithm that prevents deadlock. Use **threads** to simulate multiple/**concurrent** WARs and assume that the group of WARs are constantly attempting to use the intersection from all four directions (N, S, W, E). Your program should input parameters at runtime to initialise the number of WARs from each direction. For example, the input "`N=3 S=1 E=1 W=1`" would indicate that the simulation should start with 3, 1, 1 and 1 WARS starting from north, south, east and west direction respectively. WARs should be numbered continuously starting from 1 (e.g. WAR-1, WAR-2 etc.) You may choose how to number the WARs but the total number of WARs from each direction (N, S, E, W) must match the input. Depending on whether a WAR is going towards the Dock or towards the Storage it will be "Loaded" or "Unloaded", respectively.

Sample Input/output for Problem 1:

The input will be as follows:

```
N=3 S=1 E=1 W=1
```

The input indicates the program is initialized with 3 'Loaded' and 1 'Unloaded' WARs going from Storage 2 to Dock 2 and from Dock 2 to Storage 2, respectively, in Track 2 and 1 'Unloaded' and 1 'Loaded' WARs going from Dock 1 to Storage 1 and from Storage 1 to Dock 1, respectively, in Track 1. This set of 6 WARs will constantly attempt to use the intersection to travel between Docks and Storages in their respective tracks delivering stocks from storages to docks.

Termination Criteria:

You will terminate your simulation once the **Total crossed Track1** and **Track2** have both reached **150**.

The (**partial**) output from the beginning in one execution is as follows:

```
WAR-2 (Loaded): Wating at the Intersection. Going towards Dock2
WAR-6 (Loaded): Wating at the Intersection. Going towards Dock1
WAR-5 (Unoaded): Wating at the Intersection. Going towards Storage1
WAR-4 (Unloaded): Wating at the Intersection. Going towards Storage2
WAR-1 (Loaded): Wating at the Intersection. Going towards Dock2
WAR-3 (Loaded): Wating at the Intersection. Going towards Dock2
WAR-2 (Loaded): Crossing intersection Checkpoint 1.
WAR-2 (Loaded): Crossing intersection Checkpoint 2.
WAR-2 (Loaded): Crossing intersection Checkpoint 3.
WAR-2 (Loaded): Crossed the intersection.
Total crossed in Track1: 0 Track2: 1
WAR-6 (Loaded): Crossing intersection Checkpoint 1.
WAR-6 (Loaded): Crossing intersection Checkpoint 2.
WAR-2 (Unloaded): Wating at the Intersection. Going towards Storage2
WAR-6 (Loaded): Crossing intersection Checkpoint 3.
WAR-6 (Loaded): Crossed the intersection.
Total crossed in Track1: 1 Track2: 1
WAR-5 (Unoaded): Crossing intersection Checkpoint 1.
WAR-5 (Unoaded): Crossing intersection Checkpoint 2.
WAR-6 (Unloaded): Wating at the Intersection. Going towards Storage1
WAR-5 (Unoaded): Crossing intersection Checkpoint 3.
WAR-5 (Unoaded): Crossed the intersection.
Total crossed in Track1: 2 Track2: 1
WAR-4 (Unloaded): Crossing intersection Checkpoint 1.
WAR-5 (Loaded): Wating at the Intersection. Going towards Dock1
WAR-4 (Unloaded): Crossing intersection Checkpoint 2.
WAR-4 (Unloaded): Crossing intersection Checkpoint 3.
WAR-4 (Unloaded): Crossed the intersection.
Total crossed in Track1: 2 Track2: 2
WAR-1 (Loaded): Crossing intersection Checkpoint 1.
WAR-4 (Loaded): Wating at the Intersection. Going towards Dock2
WAR-1 (Loaded): Crossing intersection Checkpoint 2.
WAR-1 (Loaded): Crossing intersection Checkpoint 3.
WAR-1 (Loaded): Crossed the intersection.
Total crossed in Track1: 2 Track2: 3
WAR-3 (Loaded): Crossing intersection Checkpoint 1.
WAR-3 (Loaded): Crossing intersection Checkpoint 2.
...
...
```

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

Problem 2 : Monitor Colour and Monochrome Printing:

The newly established School of Information and Physical Sciences (SIPS) at 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 currently 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 number of 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 jobs.
- No time is wasted in job selection and dispatching.

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 any space 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 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.

Problem 3 : Colour and Monochrome Printing with semaphore:

You will need to implement a solution for Problem 2 (Colour and Monochrome Printing) using semaphore.

Using **semaphore**, 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 sample input/output will be the same as shown in Problem 2.

Additional Requirements:

Programming Language:

The programming language is Java, versioned as per the University Lab Environment (**currently a subversion of Java 11.0.10**). You may only use standard Java libraries as part of your submission.

Threads:

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

http://docs.oracle.com/javase/tutorial/essential/concurrency/

User Interface:

The output should be printed to the console, and strictly following the output samples shown above (also given in the assignment package as separate files). While there are no marks allocated specifically for the Output Format, there will be a deduction when the result format varies from those provided.

Input and Output:

Your program will accept data from an input file of name specified as a command line argument. The sample files `P1-1in.txt` and `P2-1in.txt` (containing inputs for Problem 1 and 2/3 respectively) are provided to demonstrate the required input file format.

Your submission will be tested with the above data and will also be tested with other input files.

Your program should output to standard output (*this means output to the Console*). Output should be strictly in the given format (*see the output files below*).

The sample files `P1-1out.txt` and `P2-1out.txt` (*containing output for* `P1-1in.txt` *and* `P2-1in.txt` *respectively*) are provided to demonstrate the required output (and input) format which **must be strictly maintained.**

If output is not generated in the required format then your program will be considered incorrect.

Mark Distribution:

A general mark distribution can be found in the assignment feedback document (`Assign2Feedback2240.pdf` and `Assign2Feedback6240.pdf`); Note that this is a draft distribution and subject to change.

Additional Task for COMP6240 Students: [NOT for COMP2240 students]:

In the lecture, two algorithms, namely Dekker's algorithm and Peterson's algorithm, were discussed as software approaches to mutual exclusion. A couple of other algorithms exist in the literature for the same purpose. You are asked to perform a survey on the software approaches to mutual exclusion and prepare a report on that. Your survey should include at least four different algorithms (may and may not include the above two). You should discuss the design principle, suitability of generalisation for n processes, suitability for multiprocessor/multi-core environment, relative advantages/disadvantages.

Your survey report should be **between 4 and 6 pages** (single space) of content, excluding cover and references.

Submission Information for COMP2240 and COMP6240:

Deliverable:

1. Your submission will contain your program source code (*for all three programs*) and a **readme.txt** (containing any special instructions – *this does not mean you are permitted to violate the following naming and running conventions*) in the root of the submission. These files will be zipped and submitted in an archive named c12345678.zip (where c12345678 is your student number) - do not submit a .rar or a .7z etc.

2. Your **main classes** for different problems should be P1.java, P2.java and P3.java and your program will compile with the command lines javac P1.java, javac P2.java and javac P1.java respectively. Your program will be executed by running java P1 input.txt, java P2 input.txt and java P3 input.txt respectively.

3. Brief **1 page** (A4) report of the how you tested your programs to ensure they enforced mutual exclusion and are deadlock and starvation free. Specifically, your report should include discussion on edge cases you considered and behaviour of your algorithm on those cases, any specific trick/technique you applied and did you face any specific issue.

4. **COMP6240** students will also submit their survey report on additional task as a PDF document, called SurveyReport.pdf.

5. Completed Assignment Coversheet.

6. Any Adverse Circumstances Extension information you may have received.

7. If you are requesting your COVID extension, you must firstly contact **Nasim** or **Dan** and inform them of this – in your submission, you will include a text file called CovidExtension.txt, containing your email and reply. You should also note your extension on your Coversheet.

Notes:

1. Assignments submitted after the deadline (**11:59 pm 17th September 2021)** will have deducted 10% of the maximum marks possible, per day late, in line with UoN Policy. This means (*for example*) if you submit **two days late**, and score **80%** in the assignment, you mark will be **80** (mark) **– 20** (2 * 10% maximum possible mark (100%)) **= 60**.
2. If your assignment is not prepared and submitted following above instructions then you will lose most of your marks despite your algorithms being correct – *the specifications are not-negotiable!*

Dan and Nasim
27AUG21 v1.0