School of Information and Physical Sciences
COMP2240/COMP6240 - Operating Systems

Assignment 2 (15%)

Submit using Canvas by **11:59 pm, Sunday 17th September 2023**

1. Tasks:

Problem 1, 2 and 3 for both COMP2240 & COMP6240 students.
Problem 4 is only for COMP6240 students.

Problem 1: Travelling Through The Wormhole

Not too far in the future, the first ever Einstein-Rosen wormhole has been established between earth and Proxima Centauri planet Proxima-b by SpaceX. Humans from Earth and aliens from Proxima-b can use the wormhole to travel to the other planet and return to their own planet and this is repeated for a fixed number of times. Once a space-traveller, say a human from Earth identified as `E_H1` (or an alien from Proxima-b identified as `P_A1`) passes through the wormhole from Earth to Proxima-b (or from Proxima-b to Earth) (s)he immediately attempts to travel back through the wormhole in the opposite direction and so on. Note that each space-traveller must travel with a unique ID which does NOT change based on their travelling direction.

The wormhole can become deadlocked if a space traveller from Earth to Proxima-b and one from Proxima-b to Earth attempts to travel at the same time and once deadlocked the wormhole may collapse. SpaceX need to keep a record of travels in any direction through the wormhole and will count multiple crossing by the same space-traveller. SpaceX will also supervise the travels of each traveller tracking their loading progress.

You have been tasked to prevent collapsing of the wormhole by preventing any deadlocks.

Using **semaphores**, design and implement an algorithm that prevents deadlock in the wormhole. Use **threads** to simulate **multiple concurrent** space-travellers and assume that the stream of travellers are attempting to use the wormhole from either direction.

Your program should input parameters at runtime to initialise the number of space travellers from each direction. For example `[E=5, P=4, N=3]` would indicate `5` humans from Earth and `4` aliens from Proxima-b from each direction wanting to use the wormhole for `3` times. You also make sure that the solution is starvation-free (the situation in which Proxima-b-bound travellers prevent Earth-bound travellers from using the wormhole, or vice versa should not occur).

Add 50 ms delay between every 25% loading of the travellers.

Sample Input/output for Problem 1.

The input will be as follows:

```
E=2, P=2, N=2
```

The input indicates the program is initialized with 2 travellers from each direction (`E=2, P=2`) who will attempt to use the wormhole to go from one planet to other for two times (`N=2`).

The output from one execution is as follows:

```
P_A1: Waiting for wormhole. Travelling towards Earth
E_H2: Waiting for wormhole. Travelling towards Proxima-b
E_H1: Waiting for wormhole. Travelling towards Proxima-b
P_A2: Waiting for wormhole. Travelling towards Earth
P_A1: Crossing wormhole Loading 25%.
P_A1: Crossing wormhole Loading 50%.
P_A1: Crossing wormhole Loading 75%.
P_A1: Across the wormhole.
COUNT = 1
E_H2: Crossing wormhole Loading 25%.
E_H2: Crossing wormhole Loading 50%.
P_A1: Waiting for wormhole. Travelling towards Proxima-b
E_H2: Crossing wormhole Loading 75%.
E_H2: Across the wormhole.
COUNT = 2
E_H1: Crossing wormhole Loading 25%.
E_H2: Waiting for wormhole. Travelling towards Earth
E_H1: Crossing wormhole Loading 50%.
E_H1: Crossing wormhole Loading 75%.
E_H1: Across the wormhole.
COUNT = 3
P_A2: Crossing wormhole Loading 25%.
E_H1: Waiting for wormhole. Travelling towards Earth
P_A2: Crossing wormhole Loading 50%.
P_A2: Crossing wormhole Loading 75%.
P_A2: Across the wormhole.
COUNT = 4
P_A1: Crossing wormhole Loading 25%.
P_A2: Waiting for wormhole. Travelling towards Proxima-b
P_A1: Crossing wormhole Loading 50%.
P_A1: Crossing wormhole Loading 75%.
P_A1: Across the wormhole.
P_A1 Finished.
COUNT = 5
E_H2: Crossing wormhole Loading 25%.
E_H2: Crossing wormhole Loading 50%.
E_H2: Crossing wormhole Loading 75%.
E_H2: Across the wormhole.
E_H2 Finished.
COUNT = 6
E_H1: Crossing wormhole Loading 25%.
E_H1: Crossing wormhole Loading 50%.
E_H1: Crossing wormhole Loading 75%.
E_H1: Across the wormhole.
E_H1 Finished.
COUNT = 7
P_A2: Crossing wormhole Loading 25%.
P_A2: Crossing wormhole Loading 50%.
P_A2: Crossing wormhole Loading 75%.
P_A2: Across the wormhole.
P_A2 Finished.
COUNT = 8
```

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

Problem 2: Cluster Management

In a cluster, a dedicated server schedules all tasks by assigning each task to a group of dedicated processors. Each task has an associated input file containing data to be processed by multiple threads. The threads linked to a task will share the group of processors allocated to that task but each thread will run exclusively on each processor (*i.e. uniprogramming mode*). Before finishing, a thread records the processing details in a system-wide common log shared by all threads of all tasks.

You need to develop a multi-threaded scheduling program for creating multiple threads for each task and assigning them to a group of processors.

Your goal is to implement a solution that uses semaphores to ensure proper synchronisations of threads avoiding possible starvation and deadlocks.

Input: The scheduler reads all the tasks from an input file; e.g. `task_list.txt` (*taken via command line argument*) which contains the input file name (*e.g.* `task1.txt`) and the number of processors requested for a task (*e.g.* `2`) in each line.

`task_list.txt`

```
task1.txt  2
task2.txt  3
task3.txt  2
```

The task input files (*e.g.* `task1.txt`) contain some integers each on a different line. Each integer represents an input to be processed by a separate thread.

`task1.txt`

```
5
8
12
```

Scheduler Thread: Your task scheduler needs to be a multi-threaded program that will create one thread for preparing each task in the input file (*e.g.* `task_lits.txt`) for running.

This scheduler thread will create multiple threads for each task, one for each integer in the task input file (*e.g.* `task1.txt`), and allocate the requested number of processors for executing all threads for that task.

Task Thread: Each thread of all tasks receives an integer, `intgr`, (*e.g.* `12`) as input and calculates it's square (`intgr*intgr`) as output, and simulates a delay, proportional to `intgr`, for calculation. You can simply do that by putting the thread to *sleep* for `intgr*100` ms.

All the threads linked to a task will compete to get access to the allocated processors to that task. As soon as a processor become available (*i.e. idle*), one of the waiting threads will start running in that processor. When all processors allocated to a task are running threads, the left-over threads wait for processor access.

It is assumed no time is wasted in running the dispatcher or switching the threads to processors.

Output: Each task thread should add the following information to a common log (*to be displayed as console output*) before termination.

Time/Date of finishing, Task input file, Thread No (of the task), input, result

```
Server initialised....
Time: Fri Aug 25 10:52:00 EST 2023, Task: task3.txt, Thread No: 0, Input: 2, Result: 4
Time: Fri Aug 25 10:52:00 EST 2023, Task: task2.txt, Thread No: 0, Input: 3, Result: 9
Time: Fri Aug 25 10:52:00 EST 2023, Task: task3.txt, Thread No: 1, Input: 4, Result: 16
Time: Fri Aug 25 10:52:00 EST 2023, Task: task1.txt, Thread No: 0, Input: 5, Result: 25
Time: Fri Aug 25 10:52:00 EST 2023, Task: task2.txt, Thread No: 1, Input: 7, Result: 49
Time: Fri Aug 25 10:52:00 EST 2023, Task: task1.txt, Thread No: 1, Input: 8, Result: 64
Time: Fri Aug 25 10:52:00 EST 2023, Task: task3.txt, Thread No: 2, Input: 6, Result: 36
Time: Fri Aug 25 10:52:00 EST 2023, Task: task2.txt, Thread No: 2, Input: 9, Result: 81
Time: Fri Aug 25 10:52:01 EST 2023, Task: task1.txt, Thread No: 2, Input: 12, Result: 144
All files processed.
```

Remark: For the same input the output may look somewhat different from run to run and the Time/Date should be different when executed again.

Note:

1. Ensure proper synchronization using semaphore to avoid conflicts, deadlock or starvation when writing to common log.
2. Ensure proper synchronization using semaphore to ensure execution of task threads in the allocated processors in a mutual exclusive manner, avoiding any starvation or deadlocks.
3. Simulate processing delay in each task thread proportional to input integer (*see above*).

Problem 3: Monitoring Cluster Management

You will need to implement a solution for Problem 2 (*Cluster Management*) using **monitor**.

Using a **monitor**, design and implement an algorithm that manages the scheduling of tasks and threads, related to a task, running into allocated processors. Use **threads** to simulate scheduling of tasks as well as running the multi-threaded tasks.

Your solution must be fair and starvation free, conflict-free and deadlock free and all the specification in Problem 2 applies the implementation should be done using monitors instead of semaphores.

The sample input/output will be the same as shown in Problem 2.

Problem 4: Additional Task For COMP6240 Students Only

[NOT for COMP2240 students]

In 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 tasked 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 or 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.

2. Other Submission Requirements:

Your submission must also conform to the follow considerations.

2.1. Programming Language:

The programming language is Java 17 (*Java 17 is the current LTS version*), as per the University Lab Environment (*you may use ES209 as a test environment).* You may only use standard Java libraries as part of your submission.

2.2. User Interface:

The output should be printed to the console, and strictly following the output samples given in the assignment package. There will be a deduction when the result format varies from the sample provided.

2.3. Input and Output:

Your program will accept data from an input file of name specified as a command line argument. The sample files `P1-1.txt` and `P2-1.txt` (*containing inputs for Problem 1 and 2/3 respectively*) are provided to demonstrate the required input file format. **Hint:** the **Java Scanner Library** is something you will likely want to use here!

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* `P1-1.txt` *and* `P2-1_tasklist.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.

2.4. Mark Distribution:

Mark distribution can be found in the assignment feedback document (`Assign2Feedback2240.pdf` and `Assign2Feedback6240.pdf`).

2.5. Deliverable:

1. Your submission will contain your program source code for all three problems, with documentation and the report (*simply called* `Report.pdf`) in the root of the submission (*you may reuse classes between problems*). These files will be zipped and submitted in an archive named **`c9876543.zip`** (*where* **`c9876543`** *is your student number*) – do not submit a `.rar`, or a `.7z`, or 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 P3.java`** respectively. Your program will be executed by running **`java P1 input.txt`**, **`java P2 input.txt`** and **`java P3 input.txt`** respectively.

   …where **`input.txt`** can be *any* filename – **do not hardcode filenames or extensions**!

Note: If your program can not be compiled and executed in the expected manner (*above*) then please add a `readme.txt` (containing any special instructions required to compile and run the source code) in the root of the submission.

Please note that such non-standard submissions will be **marked with heavy penalty**.

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`.

NOTES:

a) Assignments submitted after the deadline (**11:59 pm 15th September 2023)** will have the maximum marks available reduced by 10% per 24 hours.

b) If your assignment is not prepared and submitted following above instructions then you will lose most of your marks despite your algorithms being correct.

c) Don't use an AI!