CO3401 Advanced Software Engineering Techniques
Assignment 2019/20 (Part 1) - A Christmassy Concurrency Conundrum

Suggested Deadline: Wednesday 25th December 2019, 2.00pm

Coursework contributes 40% to the overall module mark for CO3401, with the examination contributing 60%.
Part 1 of the assignment contributes 18% (i.e. 45% of the 40%) and assesses the following learning outcomes:

3. *Analyse concurrency problems in a range of applications.*
4. *Use appropriate tools and methods for real-time software development.*
5. *Design and implement a small concurrent system in an appropriate language.*

Note: Because Learning Outcome 5 can't be tested in the examination, you must demonstrate that you can produce a program that uses concurrency appropriately.
A submission for this part that does not include a working program that passes information between threads via a buffer, means that your overall coursework mark will be capped at 35%.

Background Scenario – A Fairytale of Preston City

UCan't plc operates a Christmas theme park *"100% Lapland Style!"* in a muddy field that used to be a roundabout on the outskirts of Preston. Here, University students are employed at minimum wage during the holidays to dress up as Elves and Santas, disappointing crowds of cold and bored children by eventually handing out tacky gifts from various make-shift Grottos. All the while, miserable looking reindeer nibble on the artificial snow in their paddocks beside the lengthy queues for the disgusting portable toilets. The park is organised so that after paying the hefty entrance fee, children join a queue for a specific Grotto according to their age – the theory being that they will receive an age-appropriate gift from one of several Santas.

A disappointed family describe their visit to a similar theme park to the BBC here.

Unfortunately UCan't is facing financial difficulty since the City Council (under pressure from local consumer groups) have discontinued a substantial tax break for the company this year. A new manager, Dr. Kris Krampus, has been brought in to effect cost savings. Dr. Krampus has convinced the directors that by automating the sorting of gifts into sacks in the central "workshop" (a crudely disguised former abattoir building) using his own patented machine, the services of several student Elves could be dispensed with!

Up until now, Elves have carried armfuls of carefully chosen gifts from the Workshop to the various Grottos for distribution by one of the Santas. In the new system, at the start of their shift, Elves will fill giant hoppers with randomly selected gifts. The hopper will feed the gifts at a pre-determined rate onto a conveyor belt leading into the Sorting machine. The Sorting machine directs each gift along a sequence of conveyor belts and turntables, to be deposited in the appropriate sack. When the machine has finished or the sacks are full, elves can deliver them in a trolley to the appropriate grotto.

The 'elf and safety union insist that a maximum number of gifts can be safely loaded into a sack for an Elf to pick up at any one time, dependent on the Elf's age and general fitness. The capacity of each sack must therefore be clearly labelled in such a way that the machine knows how many presents it can put into it.

Once started by the elf in charge, the machine will run for a set length of time. If the machine is no longer able to direct presents into sacks because they are full, it will pause until either the timer has run out, or an elf changes the full sack for an empty one.

Your task is to develop a simple **Java simulation** of the workings of the Patent Present-Sorting Machine, using Concurrency.

Specification of Dr Krampus' Patent Present-Sorting Machine

Gifts are scanned automatically as they move from the hopper onto the input belt.

Each turntable has exactly four entry/exit ports oriented at 90° to one another. Each port can be configured to be one of either "entry" (through which gifts are moved onto the turntable), "exit" (through which gifts are moved off the turntable) or "blocked" (through which gifts cannot pass). Only one gift can be accommodated on each turntable at any one time. The turntable is capable of rotating in either direction, in multiples of 90° at a time.

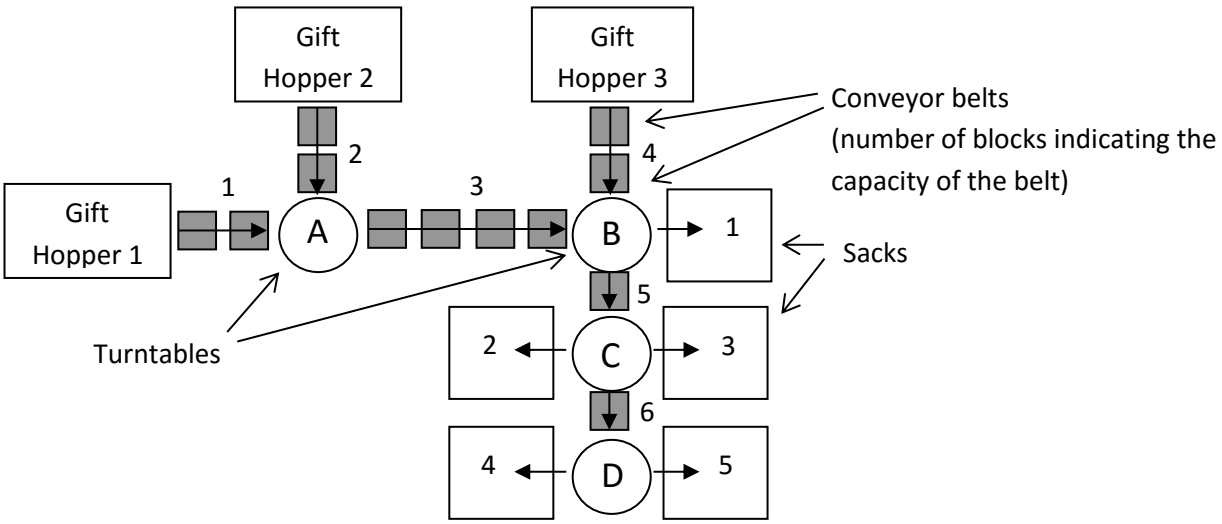
The conveyor belts which link the turntables move in only one direction, and automatically advance any gift as far as is possible (i.e. so that a turntable is always able to accept the leading gift from a conveyor containing gifts).

Each conveyor has a fixed capacity of the number of gifts which can be queued along its length, and this capacity is determined by the configuration of the system.

For the purposes of this assignment, you need only consider a simplified type of conveyor belt (more like a set of rollers) where any spaces between gifts placed on the conveyor automatically close up. So, for example, if two gifts are placed on a conveyor, they will both be queued next to each other at the far end of the belt until they can move onto a turntable. You may also assume that all gifts are in boxes which are the same size. Think of this like a queue data structure.

At the final stage of their journey, gifts are ejected from a turntable onto a chute leading directly to a numbered sack. The capacity of each sack is finite (remember 'elf and safety rules!), and gifts will only be ejected from the final turntable when a sack with spare capacity is in place at the bottom of the chute. Only one turntable may deposit gifts into any given sack (although a turntable may itself service up to three sacks).

An example configuration of such a system is illustrated below:

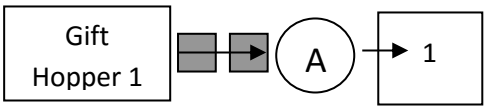


Java Implementation **29 Marks** /45

You must develop a console-based Java application to simulate Patent Present-Sorter over a single period of operation at the theme park. Your application should produce a report detailing the status of the machine at the end of operations.

Ideally, the present-sorter can be configured from a file to fill sacks for several different age groups, however I suggest you start simple, and add complexity once you are confident the concurrency aspects are working correctly.

To pass the assignment with a maximum mark of 40% you need to implement a situation with a single hopper placing presents on to a single conveyor belt, passing over a single turntable and falling into a single sack as shown below:



For the purposes of this exercise, implement any collection data types for yourself using **arrays** and **your own classes**. Do **not** use any of the Java collection types (Vector, etc.). Do **not** use dynamic data structures (e.g. linked lists). This requirement means you have to ensure safe concurrent access to the collection for yourself. A fixed-size collection is *more* likely to generate deadlock ☺. Ensure your implementation is thread-safe. Ideally the system will maximise performance by allowing as much as possible to go on concurrently. However, thread safety is more important than speed… we wouldn't want gifts going missing, and students/elves being accused of stealing them!

ESSENTIAL CLASSES

There follows an outline description of the classes you should create in order to build your application. You may need to add some add additional methods and attributes to certain of these classes, but you **MUST** follow the basic structure of my description.

Present class:

- Present objects should **NOT** run as threads.
- Present objects are created at the start of the simulation and loaded into the hopper to simulate the toys to be distributed.
- The class needs to have attributes which store the type of toy, the age-group of child it is suitable for (i.e. its destination chute).
- As well as constructor and destructor methods, a Present must have a method to allow the Sorting machine to read its destination.

Sack Class:

- Sack objects should NOT run as threads.
- Sacks should be implemented as a fixed size array, and act as a buffer for depositing Presents. They need to provide methods for the turntables to determine whether there is space in the sack for another Present.

Hopper class:

- Hopper objects should run as threads.
- Hoppers have a collection of presents.
- Hoppers are associated with a conveyor belt, and have a speed of working.
- According to its pre-set speed of working, at appropriate intervals until it is empty, a hopper will attempt to place presents onto the conveyor belt – as long as there is space on the belt.

Turntable class:

- Turntable objects should run as threads.
- When a turntable detects that a gift is waiting at one of its input ports (e.g. by polling all connected input conveyor belts in turn), the table will turn to receive the gift. Having determined the gift's destination sack (by interrogating the gift), the table will then turn to line up with the appropriate output port and eject the gift if it is able to do so.
- The turntable has 2 alignments – North-South, and East-West. Presents can be moved in either direction by the turntable, so it should only ever need to move through 90 degrees from one alignment to the other. Eg. If a present was moving from West to East, the turntable would not need to rotate with the Present on it.
    - The thread should sleep for a certain amount of time to simulate the time taken in turning. It should also sleep to represent moving the present on or off the turntable.
        - It should take 0.5 seconds to rotate 90 degrees.
        - It should take 0.75 seconds to move a present either on or off a turntable
    - To keep things simple, as part of the configuration, each turntable will have a set of destination sacks associated with each of its output ports (as well as knowing the identities of any attached conveyor belts / sacks) so that, for example if turntable "B" in the configuration illustrated on page 2 receives a gift destined for sack "4", it will know to eject the gift to the South.

Conveyor Belt class:

- The conveyor belt objects should NOT run as threads.
    - In the scenario, the belts are acting as passive buffers (shared memory space) between the various hoppers, turntables and sacks. In other words, you have something like several Producer/Consumer scenarios chained together.
- A belt will act a little like a queue data structure. The belt class should be implemented as a fixed size array to store the gift objects as they pass onto and off it (capacity according to the configuration of the machine). DO NOT use a Java collection instead of an array.
    - One simple way to implement the belt is as a circular buffer, enforcing a strict ordering to the addition and removal of gifts.
    - If you are *really* stuck with implementing the conveyor belt, you could build it with capacity for only one gift, so that you didn't need to use an array at all – just a single variable. You would lose *lots* of marks for doing this, but it might allow you to get the program working and gain other marks elsewhere. It is better to have a simpler but working program than a confused mess!
- You will need to provide a method for a Hopper to check if there is space on the belt before attempting to add a Present (to the "input" belts), and another for the turntables to see whether a gift is available for them or not.
- It is important that your program is thread-safe, and you need to take care to ensure that your conveyor belts can't be corrupted (e.g. when two turntables try to access one at the same time).

TIMING

The simulation should run for one "session" – the length of the session representing the time set on the machine by the elf. At the end of the session, the hoppers should immediately cease adding Presents to the input hoppers. Ideally, any Presents currently in the machine should continue to be sorted into the appropriate sack if this is possible, before the machine finally shuts down.

REPORTING

The system needs to report various things to the **console** as the simulation progresses:

- The following messages should be output (with timestamps starting from 0h:0m:0s when the machine starts) at the appropriate times:
    - Machine started
    - Input stopped (i.e. at the end of the timed period)
    - Machine shutdown (i.e. when all the turntable conveyor threads have stopped)
- The following messages should be output (with timestamps starting from 0h:0m:0s when the machine starts) every 10 seconds while the machine is running:
    - Total number of Presents remaining in hoppers
    - Total number of presents delivered into sacks.

At the end of the simulation, a summary report needs to be made to the **console** on the activities of the day. This should summarize:

- The configuration file(s) used.
- The length of time the machine ran for (including any extra time to process presents)
- For each Hopper, the number of presents put on the machine, and the total time spent waiting.
- The total number of presents left on the machine (on each conveyor and on each turntable).
- A check that the number presents put into the machine, minus those put into sacks, minus those left on the machine is zero – indicating that presents have not been stolen!

CONFIGURATION FILES

The configuration files I will give you for testing/demonstration follow a specific format. As an example, the configuration on page 2 would look as follows, with a section (object) heading followed by the number of objects of that type to be created, then a line representing the essential attributes of that object:

```
BELTS
6
1 length 2 destinations 1 2 3 4 5
2 length 2 destinations 1 2 3 4 5
3 length 4 destinations 1 2 3 4 5
4 length 2 destinations 1 2 3 4 5
5 length 1 destinations 2 3 4 5
6 length 1 destinations 4 5

HOPPERS
3
1 belt 1 capacity 20 speed 2
2 belt 2 capacity 10 speed 2
3 belt 4 capacity 50 speed 2

SACKS
5
1 capacity 20 age 0-3
2 capacity 20 age 4-6
3 capacity 30 age 7-10
4 capacity 15 age 4-6
5 capacity 20 age 11-16

TURNTABLES
4
A N ib 2 E ob 3 S null W ib 1
B N ib 4 E os 1 S ob 5 W ib 3
C N ib 5 E os 3 S ob 6 W os 2
D N ib 6 E os 5 S null W os 4
```

For the turntables, `N E S W` refer to compass directions. The connections are represented as follows: `ib` stands for input belt, `ob` output belt, `os` output sack, and `null` no connection. So in the above, turntable A has its North port connected to belt 2, from which is receives input, etc.

The initial collection of gifts in each sack may be represented as a section in the configuration file where the section heading is PRESENTS then the sack number, followed by a line indicating the number of Presents, and subsequent lines representing the age range for each present as follows:

```
PRESENTS 1
2
0-3
4-6
```

Presents should be output from the hopper in the order given.

The final line of the configuration file determines the length of time (in seconds) that the sorting machine should run for:

```
TIMER 300
```

MAIN PROGRAM

The main program should look something like this:

- Read from a configuration file, and create the configuration of Hoppers, Belts, Turntables and Sacks.
- Fill the hoppers with Presents according to the configuration file.
- Call the run methods on the threaded objects
- At the appropriate time, instigate the shutdown of the machine
- Make the final report

When the program runs, capture the console output to include in your documentation.

To get a mark, you will need to **demonstrate** your code running (either in a lab, or at some other time to be arranged). For the demonstration, I will provide some example configuration files, and your system's behaviour will be assessed against what I expect for those configurations.

Marks breakdown for your implementation:

Demonstration & Output trace showing correct system behaviour:	***12 Marks.***
Correct use of Java concurrency primitives:	***12 Marks.***
Appropriate use of data structures (e.g. buffers for the conveyors and sacks):	***5 Marks.***

Written Questions **16 Marks** /45

1: Explain how your code works (or, if it doesn't, how you *could* make it work) to stop all the threads and finish gracefully at the end of the simulation. (400 words max.) ***8 Marks.***

2: If you believe there is a chance of deadlock or livelock in the system, explain how it might happen, and describe any extra rules or special actions you have taken to avoid any occurrence of such. Conversely, if you believe deadlock cannot occur, explain why. (400 words max.) ***8 Marks.***

Note: You may include code snippets in your answers (not included in the word limits)

Any Java code you include in your documentation should be formatted in a non-proportionally spaced font (e.g. `consolas`*), and each line of code should fit on a single line on the page without wrapping round.*

Submission

Please refer to documentation published on the Student Notice-board on Blackboard, regarding plagiarism, and rules for submission of course-work.

Please submit the following electronically via Blackboard:

- Your full Java code - include just the .java files (not the .class files) for the classes you have implemented, and also the main program.
- A **single word processed document** containing a copy of the **console output** after running your simulation a number of times, and **answers to questions**.