CO661 Assessment 1 - Concurrent File Server

The goal of this assessment is to gain experience with shared-memory concurrency primitives in Java as well as CCS modelling. The aim is to build a simple file server which can be accessed concurrently by multiple clients.

The accompanying source files on the Moodle provide interfaces and key classes which should not be modified:

- `FileServer.java` – the main interface;
- `File.java` – a class representing *files* which clients can see;
- `FileFrame.java` – a class representing file information internally on the server;
- `Mode.java` – public enumeration of *file modes* (describes the status of a file and is also used to request access to a file with a certain mode e.g., readable or read-writeable).
- `TestSuite.java` – a standalone application for testing your program.

Your submission should be a zip file comprising at least one file for each task:

(Task 1) `MyFileServer.java`
(Task 2) `Client.java`
(Task 3) `model.ccs`

You can create extra files (e.g., for extra classes) but you should have at least the above files.

Do not modify and do not submit the provided classes.

Task 1 - Implement the shared file server (60%)

Define a class which implements the interface provided by `FileServer.java`:

```
public interface FileServer {
  // Create a new file on the server
  public void create(String filename, String content);

  // Attempt to open a file -- may block if the file is not available at that mode
  // Returns an Optional.empty() if no such file exists
  public Optional<File> open(String filename, Mode mode);

  // Close a file
  public void close(File file);

  // Check on the status of a file (the mode it is currently in)
  public Mode fileStatus(String filename);

  // What files are available on the server?
  public Set<String> availableFiles();
}
```

Clients can access files via `open(filename, mode)` with the following behaviour:

- A client can open a file for reading (`Mode.READABLE`) if it is currently closed or currently already open for reading (by other clients perhaps).
- A client can open a file for writing (`Mode.READWRITEABLE`) if it is currently closed only (`Mode.CLOSED`). If the file is in a different mode, the call to `open` should block until the file is closed (and therefore available).

For example, if a client opens a file `stuff.txt` for reading, then other clients can also open `stuff.txt` for reading. If a client opens `stuff.txt` for writing, then `stuff.txt` must already be closed (no other clients can have it open for reading or writing), otherwise `open` should block until it available. Furthermore, whilst a client has a file open for writing, any other use of `open` for that particular file (whether reading or writing) should block until the writing client has closed it.

You do not need to actually open and save actual files (you shouldn't be importing a file handling library); instead, just represent files inside your file server implementation (i.e., via a list or hash map), for which you can also use the `FileFrame` class which captures the mode and the content (string).

You should aim for some kind of fairness but there are different choices about who gets priority (e.g. readers, writers, etc.).

You can run your implementation of the file server against the test suite by compiling `TestSuite.java` and running:

```
$ java TestSuite MyFileServer
```

where `MyFileServer` is the name of your file server class.

Tip: you may want want to develop Task 2 at the same time as a further way of checking the working of your file server.

Submit: at least one Java file for your file server, e.g., `MyFileServer.java`. Include a short description (as a comment) at the start of the file (clearly marked) that explains your approach to concurrency in solving this problem including how you avoid race conditions and starvation, and how you provide fairness and mutual exclusion with respect to writing.

You may use things from Java's standard library like collections and Java's `ReentrantLock` and `Semaphore` which we covered in lectures.

On ReadWriteLocks: You may use Java's `ReentrantReadWriteLock` library to solve this, but if you do the maximum possible mark for Task 1 will be capped at 45/60. This still means you can get a first-class mark overall, but the implication is that if you can solve it using your own techniques (involving some locks/semaphores) then you can score more highly for task 1. The idea is that it would be great to see you think for yourself about how to solve this using a smaller set of basic components, though if you are having real trouble solving the problem you can still default to the `ReadWriteLock` implementation and get a good mark.

Task 2 - Client (20%)

Define a client that randomly picks files and randomly chooses to read or write them. It can write whatever text you like (e.g., the original text plus some other junk).

Tip: `ThreadLocalRandom.current().nextBoolean()` can be used for generating random booleans (for choice) and `ThreadLocalRandom.current().nextInt(start, end)` for random integers (for selecting files).

Make the client print a message explaining what it is doing each time it interacts with the file server (e.g., trying to open a file, reading/writing/closing a file).

Submit: at least one Java file `Client.java` which includes a static main method that initialises your server and spawns several clients interacting with it.

Task 3 - Model (20%)

1. Give a simple CCS model of your file server and clients in the Concurrency Workbench.

Tips Aim for simplicity: the model should capture the essential concurrent behaviour and the top-level methods (opening, closing) but need not model every detail of the implementation nor the `create`, `fileStatus`, and `availableFiles` methods. You may model just one "file" in the server for simplicity, and if you are modelling semaphores, you need only to model two or three permissions. I recommend separating out clearly the server process and the client process (which can then be used multiple times to model multiple clients; two will suffice).

2. Check that your model shows the correct mutual exclusion property: that a file cannot be open for both reading and writing at the same time. For example, if you have an observable (i.e., not scope restricted) action representing a request to open a particular file in reading mode, e.g., openRfile, and an observable action representing a request to open the file in writing mode, e.g. openWfile, then the HML property:

   ```
   <<openRfile>>(<openWfile>tt);
   ```

   describes a trace exhibiting a file opened for both reading or writing at the same time.

Submit: a text file `model.ccs` with comments explaining your model and how it relates to your implementation.