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:

- `FileServer.java` – the main interface;
- `File.java` – a class representing *files*;
- `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 one file for each task:

(Task 1) `MyFileServer.java`

(Task 2) `Client.java`

(Task 3) `model.ccs`

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 (`Model.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 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: a single Java file for your file server, e.g., `MyFileServer.java`. Include a description (as a comment) at the start of the document 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.

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: a single 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%)

Give a CCS model of your file server in the Concurrency Workbench. It should capture the 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.

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 action representing a request to open a particular file in reading mode, e.g., openRfile, and an 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.