Graded lab assignment: Downloader

1 Overview

In this exercise, you will analyze a download tool. The download tool is capable of downloading a file in chunks, with multiple downloads running in parallel. First, the main thread launches an HTTP request against the server, which may allow HTTP range requests. In that case, the part to be downloaded is split into chunks of equal size, and all but the first chunk are downloaded by worker threads in addition to the main thread. The downloaded data is saved into a file, written to by using `java.io.RandomAccessFile`, so the different chunks can be written in parallel.

If a file is large and each thread has the same throughput, all threads will finish at roughly the same time, with the main request being about one chunk into the full length (see Figure 1). In that case, the file will be closed, and the main thread will stop downloading. If the main thread finishes first, the worker threads will stop downloading. The check whether to continue downloading is made before each packet (data item which fits into a fixed-length buffer) is read.[1]

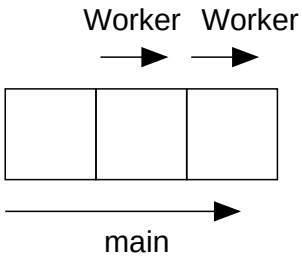

Figure 1: Overview of the download client.

The typical sequence of downloads is as follows:

1. The main thread initiates the first HTTP request; the server responds.
2. The main thread parses the header to obtain the length information of the file.
3. Based on the length (and settings), the main thread launches a number of worker threads to download smaller chunks in parallel.
4. Whenever the full file is downloaded, the output file is closed, and the remaining threads will no longer read data from the server.

To be able to run the exercise in JPF, we use *jpf-nas* to handle the client/server pair. The program has been modified such that the server is single-threaded, and the HTTP protocol has been simplified such that only the range request is sent and processed; the file data itself is hardcoded to represent the first n characters of the alphabet in lower case.

In the assignments below, "correct behavior" means no uncaught exceptions, assertion violations, data races, or deadlocks.

Coverage of certain system configurations can be shown as partial JPF executions (with logged screen output) or by temporarily adding `assert(false)` at the location of interest, and showing an error trace leading to that assertion; see Figure 2. If you print or log something inside the exception handler, you can inspect the execution output to confirm that this statement has been reached at least once. To get an execution trace to the location in question, `assert(false)` will entice JPF to produce a counterexample leading to that statement.

[1] The size of this "packet" may not correspond to the size of payloads of packets in the underlying transport layers.

```
try {
    ...
} catch(IOException e) {
   assert(false); // add to show that this handler is used
   die ("...", e);
}
```

Figure 2: How to show that an exception handler is tested.

2 Assignments: Mandatory tasks

2.1 Installation of jpf-nas

The extension *jpf-nas* is able to run multiple processes in JPF, without modification, and without exploring certain redundant interleavings between threads of different processes. It is therefore much more efficient than centralization in general. The extension is installed by first creating a `site.properties` file in `~/.jpf`. You can do this by hand or by using the script in Figure 3.

```
#!/bin/sh
[ -d ~/.jpf ] || mkdir ~/.jpf
if [ -f ~/.jpf/site.properties ]
then
echo "*** ~/.jpf/site.properties file exists; remove it to create a new one. ***"
exit
fi
cat > ~/.jpf/site.properties <<\EOF
jpf-home=${user.home}/jpf
jpf-core=${jpf-home}/jpf-core
jpf-nas=${jpf-home}/jpf-nas
extensions=${jpf-core}
extensions+=,${jpf-nas}
EOF
```

Figure 3: Creating the `site.properties` configuration file for using *jpf-nas*.

The extension itself is installed by cloning the repository using git:

1. Create the necessary entries in `~/.jpf/site.properties`.
2. `cd ~/jpf`
3. `git clone https://github.com/javapathfinder/jpf-nas`
4. `cd jpf-nas`
5. `ant`

2.2 Diagnosis

The exercise is configured so you can run the program immediately in JPF, with data race detection enabled. For scalability reasons, only one worker thread is configured, so only two downloads run concurrently. After a few seconds, JPF will terminate and report a data race.

You compile and run the program using
`./build.sh && ./run.sh`

1. Diagnose the error trace shown by JPF. Use a table like Table 1. If you have many transitions with "ins[truction]s w[ith]o[ut] sources", you can summarize these as "library code" together with more interesting actions of other threads.

Thread	0	1	
Trans.	Server main	Downloader	Worker
0	read configuration		
	call accept		
1		create output file	
		connect	
2	accept		
	read request		
	... (to be completed by you)		
12–17			connect
	... (to be completed by you)		

Table 1: Example showing how thread transitions can be summarized.

2. Write a descriptive summary of the actions leading to the data race. Show how the server and clients interact, and how the failure happens. This can be done as an enumeration of about 5–12 key actions.

3. Fix the defect; run JPF again.

4. JPF will report another race; fix it (without documenting the error trace again) and provide a **patch to fix all data races**.

5. JPF should now take about 5–10 seconds to analyze the full example and report no errors.
To be sure that no exceptions are thrown, add `assert(false)` to the code in "die", or inspect the output of the program to check that no handled and logged exceptions persist.

3 Optional tasks

3.1 Validation, analysis

1. Explain why your solution is likely correct. Are workers added and executed thread-safe way? Is the access to the shared output file always safe (no attempt to write to a closed file; possible exceptions handled)?

2. What can you say about the resulting system after it has been verified with JPF? Is it proven to be correct? Why (not)?

3.2 Double close

1. Insert an extra assertion in `java.io.RandomAccessFile`, which ensures that a file is not closed twice. (This is not prohibited per se, but a sign of a flawed implementation).

2. Run the program again, summarize (in a table) and explain (in text) the error trace. You can summarize that part that is the same as before as text.

3.3 Output oracle

At the moment, the test execution in JPF only confirms the absence of data races and the lack of file accesses to a file that is already closed, because otherwise, an exception would be thrown (and visible in the log).

The exercise comes with a model of `java.io.RandomAccessFile`, as the built-in implementation in jpf-core does not successfully provide a persistent state across closing the file (in the downloading thread) and re-opening it.

1. Add a new `import` statement in `Downloader.java`, to import `env.java.io.RandomAccessFile`.

2. Add a function `oracle`, with assertions checking the contents of the file. That function opens the same output file again (in a new instance of `RandomAccessFile`, using read-only mode `"r"`), and checks that the contents are complete (3 bytes in this example), and contain the right data (bytes encoding letters 'a', 'b', and 'c' in UTF-8).
3. Call the oracle function in all states where the file contents are supposed to be complete.

3.4 Stale value?

In concurrent program, a *stale value* is a local copy of shared data that may get outdated if other threads update the shared data. In this sense, *nDownloaders* in line 156 in Downloader.java is such a local copy of a shared value. What happens if you change the code as below and use the shared value *downloaders* instead?

Explain the relevant part of the error trace (from where it differs from the trace for the base assignment).

```
diff -urw base/src/Downloader.java stale-value/src/Downloader.java
--- base/src/Downloader.java 2020-04-12 14:38:12.000000000 +0200
+++ stale-value/src/Downloader.java 2020-04-12 14:38:46.000000000 +0200
@@ -153,8 +153,7 @@
  int end = length;
  int chunkSize = length / (downloaders + 1);
  int start = 0;
- int nDownloaders = downloaders;
- for (int i = 0; i < nDownloaders; i++) {
+ for (int i = 0; i < downloaders; i++) {
  start = end - chunkSize;
  new Downloader(start, end).start();
  end = end - chunkSize;
```

3.5 Stubbing

Unfortunately, without further optimization, it is not possible to check the Downloader with two worker threads. However, this can be achieve by *stubbing out* the implementation of the web server, and replacing the whole process with a few model classes (stubs).

3.5.1 Stub usage

1. In `Downloader.java`, change the import statements for `java.io` and `java.net` to `env.java.io` and `env.java.net`.
2. Change the setting `DL_THREADS` to 2.
3. Implement the two variants of `java.io.InputStream.read` as specified by the comments.
4. Run JPF using `run_standalone.sh`, to run only the Downloader. The run should be successful (but it may take about one minute).

3.5.2 Fault injection (requires oracle implementation)

1. Use `gov.nasa.jpf.vm.Verify.getBoolean` to inject a fault in the `InputStream.read` method that returns the content (i. e., not the length) of the request.
2. Run the modified program, which now fails again. Summarize (in a table) and explain (in text) the error trace. You can summarize that part that is the same as before as text.

3.6 Extra (optional)

You find an unexpected defect (in this assignment or jpf-core or jpf-nas) that is not directly related to a previously known defect, and you are able to document it and or/fix it convincingly.

Note: There is no guarantee for the presence or absence of additional defects in this exercise.

4 Grading criteria

Description	Code	Trace	Doc.	Points
There is a coherent and convincing argument on the correctness of the resulting system design.			✓	+0.5
There is a correct assessment on the soundness of the verification methodology.			✓	+0.25
The assertion to check against double close is correct, and the resulting error trace is explained correctly.	✓	✓	✓	+0.75
The test oracle is function is implemented correctly.	✓			+0.5
The oracle function is used in all states where the file is supposed to be complete.	✓			+0.25
The effect of the change that eliminates the stale value is explained correctly.		✓	✓	+0.5
Stubs for `InputStream.read` are implemented correctly.	✓			+0.5
Fault injection in InputStream.read is implemented correctly.	✓			+0.25
The resulting error trace (with a failing oracle function) is explained correctly.		✓	✓	+0.5
Extra (at the discretion of the examiner).	?	?	?	+0.5

Table 2: Extra points for advanced features/properties. "Doc." represents documentation beyond short comments in code, which are subsumed by "Code".

Pass (E):

- The submitted work is your group's own original work. You may look at examples from the web for inspiration, but you are not allowed to copy code from the web.
- All mandatory parts are implemented correctly.

Extra points are awarded according to Table 2.

Reduced points for late submission! The maximal number of points is reduced by 1 for each missed deadline. Two deadlines:

- Base JPF exercises.
- JPF project.