École Polytechnique de Louvain

LSINF2251

Software Quality Assurance

Assignment 3 : Daisy : a Simple File System

Vahid Beyraghi - Guillaume Bellon

32381600 - 30151600

INFO

vahid.beyraghi@student.uclouvain.be

guillaume.bellon@student.uclouvain.be

May 16, 2021

Section 1

Question 1 : Code analysis

With the basic code, there are two operations (READ_OPERATION and DELETE_OPERATION) that can be executed with one iteration and one file. There are two DaisyUserThreads that will be doing those operations, which means that we have one binary choice over two threads. The number of concurrent test cases, according to its definition in the instructions, is computed below in equation below.
$\#\_concurrent\_test\_cases = \prod_{N\_thread}(N\_OPERATION\_CHOICES \times FILECOUNT \times ITERATIONS)$
$\#\_concurrent\_test\_cases_1 = (1 \times 1 \times 1)^2 = 1$

Section 2

Question 2 : First JPF Run

Here you can see the output statistics of the JPF analysis. We will consider the number of visited states for our analysis throughout this report.

```
==================================================== statistics
elapsed time:         00:00:22
states:               new=134152,visited=128860,backtracked=260811,end=2
search:               maxDepth=2740,constraints=0
choice generators:    thread=134151 (signal=95,lock=16112,sharedRef=83121,threadApi
   =1992,reschedule=32831), data=0
heap:                 new=43454,released=59000,maxLive=1008,gcCycles=233361
instructions:         20748838
max memory:           494MB
loaded code:          classes=90,methods=1942
```

Listing 1: Statistics

We can see an elapsed time of 42 seconds, a size of memory used of 494MB and a number of visited states of 128860. All these numbers are huge because we do not impose a depth limit and thus the trace becomes big.

Section 3

Question 3 : JPF Run with depth limit

To obtain the shorter trace that still report the errors we had to run the code many times with a different depth limit. The main idea was to reduce the depth limit to the minimum depth that still detects an error to produce a trace with a minimal amount of transitions, and thus of minimal length. This was done through trial and error, and the different outputs are reported in table 3.1.
The trace length is approximated to the number of lines of the output of JPF.
For this question we do not provide a shell script to reproduce the reported results as we directly modified the source code to produce the results. However we exported all the output of JPF in files that can be founded at the "traces_question_3" directory.

3.1 Resume

Depth limit	Time (s)	Trace length	Number of states	Error detected ?
No depth limit	22	15162	128860	yes
1000	11	12385	73484	yes
500	4	10950	25552	yes
50	0	22	3758	no
100	0	9750	2735	yes
80	0	9690	1567	yes
70	0	22	7696	no
78	0	9684	1454	yes
77	1	22	9227	no

Table 1: Statistics results by searching the minimal depth limit

The minimal trace length is around 9684 lines with a depth limit of 78. Below that number, the error is not detected anymore.

Section 4

Question 4 : Fixing the first issue

When the delete operation is done completely before the read (and the lookup), the lookup does not find the file and returns an `ERR_NOENT`. However, the read simply reads the content in the `inodenum` given by the `filehandle` (`fh`) that has not been initialized yet.

Indeed, when the lookup finds the file, it is supposed to set the `inodenum` of `fh` to the `inode` number corresponding to the file that we want to read, and the read operation is done on the `inode` with number "`fh.inodenum`". This is why, in the case where lookup doesnt set anything the `fh.inodenum`, the read is done on `inode` 0 (default long value) and why the read doesnt fail (as there is data in `inode` 0 -root folder-, the deleted file was located in `inode` 1).

The solution to this problem that we chose was to encapsulate the `inodenum` field within the `FileHandle` class, as it is a shared object between threads, and to add another field initialized that is set to false initially, but that is turned to true as soon as the `inodenum` field is set. This way, it is possible to see if the `FileHandle inodenum` field was initialized or not, in order to check if the read operation needs to be done or not. As a not initialized `inodenum` field corresponds to a lookup that didn't find the file that needs to be read, the read operation does not need to be done (or at least not on the default value of `inodenum`).

Section 5

Question 5 : New JPF Run, analysis and correction

Here is the JPF statistics output after with no depth limit after the first correction.

```
======================================================= statistics
elapsed time:        00:00:44
states:              new=135770,visited=130446,backtracked=263987,end=2
search:              maxDepth=2772,constraints=0
choice generators:   thread=135769 (signal=95,lock=17710,sharedRef=83121,threadAp
                                       i=2012,reschedule=32831), data=0
heap:                new=43906,released=59589,maxLive=1008,gcCycles=234949
instructions:        21024045
max memory:          494MB
```

```
loaded code:          classes=90,methods=1944
```

Listing 2: Statistics

We can see a elapsed time of 44 seconds, a number of visited states of 130446 and a memory size of 494MB.
To find the error we analysed the trace and identified the assert that raised it. By observing the code, we realized that the assertions on the `READ_OPERATION` and `WRITE_OPERATION` corresponding to the case where the lookup did not find the file the respectively read or write need to be disabled as the read and write are done on the default `inodenum` value (0) that has not been initialized yet. This allowed us to remove all errors, for remaining code without a depth limit.

Section 6

Question 6 : No issue JPF Run analysis

Here you can see the output and statistics of the "no issue" jpf run.

```
======================================================= statistics
elapsed time:         00:03:59
states:               new=679685,visited=988382,backtracked=1668067,end=3
search:               maxDepth=2854,constraints=0
choice generators:    thread=679685 (signal=6391,lock=94072,sharedRef=416588,threadApi
    =2094,reschedule=160540), data=0
heap:                 new=1917762,released=167914,maxLive=1009,gcCycles=1460287
instructions:         53404239
max memory:           499MB
loaded code:          classes=86,methods=1903
```

Listing 3: Statistics

This run takes much more time than the earlier ones because there are no error anymore and the search tree is fully explored.

Section 7

Question 7 : Specific executions

For this question, multiple runs with specific parameters are needed. You can see all theses output following and a resume table is available at point 7.1. Like the third question we don't provide a script as we directly modified the source code to produce these results. Nevertheless we export the JPF output to obtain all the traces for this question. You can find all the traces files at the "traces_question_7" directory.

7.1 Results

Test	Time (s)	Memory (MB)	Number of states
a	1	492	15958
b	7	493	88524
c	7	493	90132
d	5	494	54988
e	10	494	117192
f	6	493	96886

Table 2: Resume of the statistics of the different JPF run

7.2 Results discussion

Following the equation in section 1, we can find the theoretical number of concurrent test cases and compare it to the computed data.
We decided to compare the concurrent test case number to the number of states and see if the evolution is similar.
To compare the following executions we had first to run an original JPF execution to obtain the number of states.
15700 states are visited for this initial run.

7.2.1 a

The first thread still doing a read operation and the second thread performs now a write operation. The results should be similar because each thread has only one choice of operation. The equation still $(1 \times 1 \times 1)^2 = 1$ and the visited states should approximately not change. This is the case because we obtain a number of states of 15958 very close to the 15700 from the original run.

7.2.2 b

The first thread still doing a read operation and the second thread performs now a random operation.
There are 6 operations that this thread can performs :

- READ_OPERATION
- WRITE_OPERATION
- CREATE_OPERATION
- DELETE_OPERATION
- SET_ATTR_OPERATION
- GET_ATTR_OPERATION

The `#_OPERATION_CHOICES` grows now to 6 because of that and thus gives us a number of concurrent test cases of $(1 \times 1 \times 1) \times (6 \times 1 \times 1) = 6$. The number of states should thus be approximately 6 times bigger than the original run. The number of states for this run is 88524 which is 5.63 times bigger than the initial 15700 states which is pretty similar.

7.2.3 c

The first thread still doing a single read operation but now the second thread performs two random operations. With the same 6 operations available the number of concurrent test cases reach now $(1 \times 1 \times 1) \times (6 \times 1 \times 2) = 12$. By looking at the number of states reached with this run, 90132, we see that we only reached a multiplication of 5.74 which is pretty far from the computed 12 factor.

7.2.4 d

The first thread still doing a single read operation and the second thread performs a single delete operation, but this time, on two different files. The number of concurrent test cases is now computed by $(1 \times 2 \times 1) \times (1 \times 2 \times 1) = 4$. The real number of state obtained, 54988, is 3.5 bigger than the initial 15700 number of states, which approximately correspond the 4 factor estimated.

7.2.5 e

Now the two threads have to performs their operations on three files which increased the number of test cases to $(1 \times 3 \times 1) \times (1 \times 3 \times 1) = 9$. 117192 states are visited which correspond to 7.46 times the initial number of states. Theses factors are approximately similar.

7.2.6 f

For this part, a third thread is used. The actions performed by the three treads are respectively a read, a delete and a write operation. By computing the number of test cases we obtain $(1 \times 1 \times 1) \times (1 \times 1 \times 1) = 1$. The number of states should approximately stay the same.
Unfortunately a deadlock appeared at this time and so the collected data are not relevant.