CHARACTER DEVICE

Op de Raspberry PI type 1A

SUZANNE PEERDEMAN & TIM VISSER

1 INTRODUCTION

For this assignment we decided to create a loadable kernel module. A loadable kernel module means that the module in question can be compiled without recompiling the entire kernel, so rather than changing the existing kernel the module will extend the kernel. This will save both time when compiling, and effort when coding. This paper will highlight two kernel modules that we have created: the first will deliver a message at boot and shutdown, and the second will control one of the GPIO (general purpose input/output) pins on the raspberry pi in order to blink a LED. The code will be written in the programming language C.

2 EXPERIMENTAL SETUP

CODE UITLEGGEN EN WEGHALEN ALS CODE This research uses Java code and R code that is interpreted in the JVM through the Renjin library. The JVM has been given the arguments `-Xms2048M` and `-Xmx2048m` to increase the stack size to the maximum for a 64-bit JVM.

To retrieve outputs from `java.util.Random` the following code will be used:

Listing 1: Retrieve random values through a recursive function and store them in an array

```
...
private static Random random;
private static long[] timestamps;

private static float[] initializeRandomArray (int size) {
        return addRandom(new float[size], new long[size], 0,
            size);
}

/**
* Recursive function to construct a float[] with length of count
    filled with java.util.Random output values.
* @param array Enriched with a new random.nextFloat() each
    iteration.
* @param timestamps Saves System.currentTimeMillis() each
    recursive call.
```

```
* @param count The amount of values the result will eventually
    have.
* @return float[] filled with count number of java.util.Random
    output values.
*/
private static float[] addRandom(float[] array, long[]
    timestamps, int index, int count) {
        if (index >= count) {
                // Set static variable 'timestamps'
                Experiment.timestamps = timestamps;
                return array;
        } else {
                timestamps[index] = System.currentTimeMillis();
                array[index] = random.nextFloat();
        }
        return addRandom(array, timestamps, ++index, count);
}
...
```

In the code above 1 the researcher chose to use a primitive array of type float to store outputs into. He chose that because he uses Renjin for this research, and one can easily pass primitives to the Renjin R interpreter.

WE/THE RESEARCHER AND DEFINE 'DAMAGE' In the code of listing 1 we can also see that outputs are collected through a recursive function. The researcher believes that timestamps can be a contributing factor in the mechanism of `java.util.Random.nextFloat()`. Therefore, in an effort to minimize the damage different timestamps could have on ouputs, we first store all function calls on the stack. When we exit the recursive function all calls are then fired all at once. In addition the timestamps (in milliseconds) are also collected and stored in `long[] timestamps`.

CORRELATION OR PATTERN?? => PATTERN The experiment is conducted by retrieving `java.util.Random.nextFloat()` output values in as little time possible. This is done four times with array lengths: `10`, `100`, `1000` and `7000`. We then calculate the arithmetic mean, median, mode and standard deviation. These values can then be used to compare mean with median and to calculate probability with R's `pnorm` function. The results of these comparisons and probabilities can then be analyzed. If we can find one or more correlations in the output values this means that `java.util.Random.nextFloat()` is not random at all but instead follows those correlations.

3 RESULTS

MAAK ER EEN TABEL VAN The following (listing 2) is the output of the program.

Listing 2: Ouput of the program

```
Experiment [n=10]:
Generating java.util.Random values took 1.0 millisecond(s)
```

```
Mean                 [1] 0.39593282938004
Median               [1] 0.32979026436806
Mode                 [1] 0.14058691263199
Standard deviation              [,1]
[1,] 0.26815442694344
-------------------------------------------
Mean +- median       [1] 0.06614256501198

Probability n > 0                      [,1]
[1,] 0.93009655189399
Probability n < 1                      [,1]
[1,] 0.98786045781739
Probability n > median                 [,1]
[1,] 0.59741373140378
Probability n < median                 [,1]
[1,] 0.40258626859622
Probability n > mode                   [,1]
[1,] 0.82951098505196
Probability n < mode                   [,1]
[1,] 0.17048901494804



Experiment [n=100]:
Generating java.util.Random values took 1.0 millisecond(s)

Mean                 [1] 0.49393919229507
Median               [1] 0.46484676003456
Mode                 [1] 0.10284048318863
Standard deviation             [,1]
[1,] 0.2714489869447
-------------------------------------------
Mean +- median       [1] 0.02909243226051

Probability n > 0                      [,1]
[1,] 0.96559300898027
Probability n < 1                      [,1]
[1,] 0.96885980880285
Probability n > median                 [,1]
[1,] 0.54267476085866
Probability n < median                 [,1]
[1,] 0.45732523914134
Probability n > mode                   [,1]
[1,] 0.92517680512418
Probability n < mode                   [,1]
[1,] 0.07482319487582



Experiment [n=1000]:
Generating java.util.Random values took 2.0 millisecond(s)
```

```
Mean                 [1] 0.49332589697838
Median               [1] 0.4915097951889
Mode                 [1] 0.93314707279205
Standard deviation                [,1]
[1,] 0.29082325410522
-------------------------------------------
Mean +- median       [1] 0.00181610178947

Probability n > 0                      [,1]
[1,] 0.95508624436552
Probability n < 1                      [,1]
[1,] 0.95926382325647
Probability n > median                 [,1]
[1,] 0.50249125566892
Probability n < median                 [,1]
[1,] 0.49750874433108
Probability n > mode                   [,1]
[1,] 0.06522477234239
Probability n < mode                   [,1]
[1,] 0.93477522765761



Experiment [n=7000]:
new double array length = 7000
new double array length = 7000
Generating java.util.Random values took 1.0 millisecond(s)
new double array length = 7000
building DoubleVector = 7000
IntArrayVector alloc = 7000

Mean                 [1] 0.50724935386862
Median               [1] 0.50939035415649
Mode                 [1] 0.31117027997971
Standard deviation               [,1]
[1,] 0.2861182375764
-------------------------------------------
Mean +- median       [1] -0.00214100028787

Probability n > 0                      [,1]
[1,] 0.96187456005885
Probability n < 1                      [,1]
[1,] 0.95748266186443
Probability n > median                 [,1]
[1,] 0.49701477412488
Probability n < median                 [,1]
[1,] 0.50298522587512
Probability n > mode                   [,1]
[1,] 0.75342515998576
Probability n < mode                   [,1]
[1,] 0.24657484001424
```

We want to find one or more correlation in the output above, thus proving that `java.util.Random.nextFloat()` follows a pattern. If we can prove that

Table 1: Results of Computations

a pattern is being followed, our hypotheses is wrong.

In all executions of the program (`n=10, n=100, n=1000` and `n=7000`) the mean lies very close to the median. This indicates that the differences between each `n` and `n+1` are more or less the same for the whole range (0-1). In other words, it indicates that we have little to no outliers in our results.

If we compare the probabilities of each experiment (`n=10, n=100, n=1000` and `n=7000`), we also see that results are generally very close to each other. The probability of an entry `n > 0` means: what is the chance that this entry is bigger than `0`? And so we can see that all our data sets are evenly distributed, because $n <> median \approx 0.5$ and our modes are never close to `0.5`.

4 DISCUSSION

The researcher states some facts about JVM runtime environment, but this research never clears if those facts indeed change the measurement results.

Stack memory is limited and is therefore a bottleneck in how many recursive function calls can be stored on it. In the case of this study only 7000 output values with their corresponding timestamps could be retrieved without triggering a `stack overflow exception`. We do not know how many output values are actually needed for a trustworthy result. More is better is the norm here.

The researcher believes timestamps can have an impact on the outputs of `java.util.Random.nextFloat()`, while never proving it. This could be inspected by looking at the actual source code of `java.util.Random` and determining if the source code is in any way associated with time and/or timestamps.

5 CONCLUSION

GIVEN THE THINGS MENTIONED IN DISCUSSION, WE CAN ASSUME THAT... Q.E.D. Because we see a clear correlation between our data sets in that they are all evenly distributed we can conclude that this is a pattern that outputs of `java.util.Random.nextFloat()` follow, thus proving that `java.util.Random.nextFloat()` **does** follow a clear pattern.