INTERNAL REPORT

Analysis of Random Number Generation

Vivek Yadav

An enterprise software company sells a program that prints out 100,000 random integers. We have placed 2 copies of this program in your home directory. With version 1 of the software (RandomGenerator1), the time it takes to print those 100,000 integers is pretty fast. Unfortunately inadequate performance testing has caused the company to ship RandomGenerator2 that runs much more slowly.

First, run the 2 programs and verify they indeed print 100,000 integers. How can you be sure?

Next, compare the running time of the two versions of the software. Is the second one actually slower? If so, how much slower? Is it slow all the time or just some of the time?

Finally, try to find the root cause as to why the second program runs more slowly. Is it the software itself? The system? Something else? All of the above?

ANALYSIS

PART 1

The first part of the program was running the class files of the two programs. Running the programs was achieved using the following java commands –

```
Java RandomGenerator1

Java RandomGenerator2
```

Running the programs printed 10000 numbers to the console and verifying them from console by counting was not feasible. So to count the total number printed, I stored the output in a file, and did a wc (word count) command to count the number of lines in the file.

```
interview@ip-10-0-0-127:~$ java RandomGenerator1 > RandomGenerator1_output.txt

interview@ip-10-0-0-127:~$ cat RandomGenerator1_output.txt | wc -l

100000

interview@ip-10-0-0-127:~$ java RandomGenerator2 > RandomGenerator2_output.txt

interview@ip-10-0-0-127:~$ cat RandomGenerator2_output.txt | wc -l

100000
```

PART 2

The second part of the program was comparing the running time of the two versions of the software.

For this a method was required to find the running time of each of these programs.

Running commands manually in separate windows to accomplish this task, would introduce human centric latency in calculating the correct time.

So I created a java program for finding this.

Note: Initially, running the program manually I found that printing 1000000 numbers on screen was taking more time than writing 1000000 numbers to a file.

I used the below algorithm

```
// store time in a variable just before starting the RandomGenerator program

final long startTime = System.nanoTime();

// run the RandomGenerator program

Runtime rt = Runtime.getRuntime();

Process proc = rt.exec("java RandomGenerator1");

// get the time as soon a program finishes and then subtract the starttime from it

final long duration = System.nanoTime() - startTime;
```

I ran this program but the program hung at the step where RandomGenerator program ran. After doing some analysis on google I understood that this process was running on a different thread, and for some reason it was not able to return back to the main thread, so the program hung and did not show any output. (I eventually realised this understanding was wrong. More on that analysis later)

At this step i had to find a way to have this new thread (which the Process 'proc' creates) to return back to the main thread. After some google searching, i went ahead with the approach of making a new thread in my program to run this piece of code using the Java Runnable class. I assumed that the new thread would automatically return to the main thread after doing its work. But this approach also did not work and the program hung.

Then after some more research on google i found the 'Thread.join' command which joins any new thread to the main thread. So I thought that this would definitely work after using the join command. But the program still hung.

At this point i commented the running of the RandomGenerator program in my custom program and found that the thread was returning back (checked using some print statements). So i found that the problem was not with the thread not returning, but with the RandomGenerator program itself.

After some more analysis on google, i found that the output of the process ‘proc’ (which runs the ‘java RandomGenerator’ command in my custom program) needs to be captured and printed. Basically for the program to return to the main thread and finish execution, the buffer needs to be emptied.

Then after some analysis i modified this program to run the RandomGenerator and store its output in a bufferedReader and print the 100000 numbers on the screen

```
// store time in a variable just before starting the RandomGenerator program

final long startTime = System.nanoTime();

// run the RandomGenerator program

Runtime rt = Runtime.getRuntime();

Process proc = rt.exec(“java RandomGenerator1”);

// get the time as soon a program finishes and then subtract the starttime from it

final long duration = System.nanoTime() - startTime;
```

This time the program ran and printed the 1000000 numbers on the screen. The average time i saw was about 0.7 to 0.9 seconds.

But i already found that printing on screen was taking more time than writing to a file.

So i changed this algorithm to write to a file.

```
// store time in a variable just before starting the RandomGenerator program

final long startTime = System.nanoTime();

// run the RandomGenerator program

Runtime rt = Runtime.getRuntime();

Process proc = rt.exec(“java RandomGenerator1”);

BufferedReader stdInput = new BufferedReader(new InputStreamReader(proc.getInputStream()));

FileWriter writer = new FileWriter("output.txt");

BufferedWriter bw = new BufferedWriter(writer);

while ((s = stdInput.readLine()) != null) {

      bw.write(s);

}

// get the time as soon a program finishes and then subtract the starttime from it

final long duration = System.nanoTime() - startTime;
```

On running this I found the execution time changed from 0.7 - 0.9 seconds to about 0.4 – 0.6 seconds. I feel this is the fastest measurement I can do.

I also tried a final approach to make the above faster by getting the data in the buffered reader, but not printing it either to a file or to the console (not printing them would save time). But this doesn't work. The process buffer (which stores the 1000000 numbers) needs to be read from and emptied for the program to move forward.

Once my program was ready, I ran both the programs and found the difference between RandomGenerator 1 and RandomGenerator1 as the below –

Random Generator 1 – 0.4 to 0.6 seconds to generate 1000000 numbers

RandomGenerator 2 – 100 to 400 seconds

PART 3

The next part was to find why the second program runs so slowly.

Since there were only .class files present (byte code) and no .java files, I had to decode and see if I could find what was written in the actual program.

I started with the javap utility and found the below important contents

```
interview@ip-10-0-0-127:~$ javap -c -p -v RandomGenerator1.class
Constant pool:
      #2 = Class              #22             //  java/util/Random
```

```
interview@ip-10-0-0-127:~$ javap -c -p -v RandomGenerator2.class
Constant pool:
      #19 = Class              #35             //  java/security/SecureRandom
```

As seen from above outputs, the random number is being generated by different classes. After researching on google on both these classes, some important points to note –

- A Random class has only 48 bits whereas SecureRandom can have up to 128 bits.

- By default, SecureRandom (from java.security package) is more than ten times slower than Random (from java.util package)

- SecureRandom make use of any one of the random devices on Linux: /dev/random and /dev/urandom. The best randomness comes from /dev/random, since it's a blocking device, and will wait until sufficient entropy is available to continue providing output. Assuming your entropy is sufficient, you should see the same quality of randomness from /dev/urandom; however, since it's a non-blocking device, it will continue producing “random” data, even when the entropy pool runs out.

From the above points i was almost sure that the RandomGenerator2 was using SecureRandom with /dev/random. To confirm this finding i ran the below commands –

`java -Djava.security.egd=file:/dev/./urandom RandomGenerator2` (my observation was that this ran very fast because this uses /dev/urandom which is non blocking)

`java -Djava.security.egd=file:/dev/./random RandomGenerator2` (my observation was that this ran very slow because it uses /dev/random which is blocking)

As a last step to confirm this I ran the strace command (the last RD_ONLY is to /dev/random)

```
interview@ip-10-0-0-127:~$ strace -o a.strace -f -e file java RandomGenerator2
interview@ip-10-0-0-127:~$ cat a.strace | grep random
24970 stat("/dev/random", {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 8), ...}) = 0
24970 stat("/dev/urandom", {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 9), ...}) = 0
24970 open("/dev/random", O_RDONLY)     = 12
24970 open("/dev/urandom", O_RDONLY)    = 13
24970 open("/dev/random", O_RDONLY)     = 14
```

At his point I assume that my observations are correct and i went ahead to see what affects /dev/random slow execution times.

After researching on google I found that, on most Linux systems, /dev/random is powered from actual entropy gathered by the environment. If the system isn't delivering a large amount of data from /dev/random, it likely means that there is not enough environmental randomness (entropy) generated to power it.

When the entropy pool is empty, reads from /dev/random will block until additional environmental noise is gathered.

To see the entropy generated in the system, I ran the below command and watched the output –

```
interview@ip-10-0-0-127:~$ watch cat /proc/sys/kernel/random/entropy_avail
49
```

I found that the entropy was always within the range 0 to 64, and as soon as it reached 64, it reset back to a number between 0 -10. This reason it got reset around 64 is because of another random generation related parameter ‘read_wakeup_threshold’

```
interview@ip-10-0-0-127:/proc/sys/kernel/random$ cat read_wakeup_threshold
64
```

Basically when the entropy_avail reaches or crosses the read_wakeup_threshold, all the processes that are currently blocked from reading from /dev/random due to low entropy, will wake up again and start to read from /dev/random.

So when these devices read from the random pool, they use the entropy and the entropy gets reset. Once the entropy resets these process get blocked again until the entropy_avail reaches 64.

So at this point i understood that our RandomGenerator2 program is reading from /dev/random once the entropy reaches 64, but it is not able to gather all 1000000 numbers in one go. It requires multiple tries to read from /dev/random to gather all 1000000 numbers. And since /dev/random is blocking, when entropy is low, our program needs to keep waiting until all 1000000 numbers get gathered, hence the slow execution time of RandomGenerator2.

At this point, i also tried to find out how many times RandomGenerator2 needs to read from the random pool buffer to get all the desired 1000000 numbers it needs. I was assuming that this should turn out to be a constant number because in one go if it reads x amount of numbers, then it should required 1000000/x number of times. I was assuming it would be reading x everytime (example 5000) and this should remain constant.

When I ran my test for the above, I found that this number was not a constant for RandomGenerator2. The findings were as follows –

	Duration of RandomGenerator2 execution	number of tries to read from pool
Try1	3 min 48 sec	12
Try2	5 min 12 sec	15
Try3	3 min 15 sec	10
Try4	6 min 9 sec	18

The difference in time made sense because that depended on the speed of entropy generation. And entropy generation speed is not constant, as it depends on external factors such as noise in the environment. But the varying number of tries was not expected.

On more analysis in the system (EC2 machine) I found that, there was another process running which was accessing /dev/random

```
Every 2.0s: lsof /dev/random                                            Sun Jan 23
11:56:23 2022
COMMAND   PID      USER    FD    TYPE DEVICE SIZE/OFF NODE NAME
java     1050 interview   12r    CHR    1,8      0t0 1051 /dev/random
java     1050 interview   14r    CHR    1,8      0t0 1051 /dev/random
python   1209 interview    3r    CHR    1,8      0t0 1051 /dev/random
```

This python program is a constantly running program since Jan 14 and seems it is always trying to read from /dev/random

```
interview@ip-10-0-0-127:~$ ps -ef | grep 1209
intervi+  1209  1208  0 Jan14 ?        00:00:00 python /usr/local/bin/rdoe
intervi+ 10874 10854  0 16:24 pts/0    00:00:00 grep --color=auto 1209
```

So this explains why our RandomGenerator2 is slow and also why each execution requires different number of attempts to gather the 1000000 numbers. Because this python program reads from /dev/random when entropy_avail reaches 64. So this program and RandomGenerator2 are sharing the random data from /dev/random pool.

To see if the super slowness of RandomGenerator2 can be any faster, I tried to pause the python process for a while to see if it affected the RandomGenerator2 speed. And my finding was that it did affect, and RandomGenerator2 was considerably faster when the other python program as paused (and hence temporarily not reading from /dev/random). I saw the execution times decreased to between 47 to 90 seconds. Results below –

```
interview@ip-10-0-0-127:~$ lsof | grep /dev/random
python     1209       interview    3r      CHR    1,8      0t0   1051 /dev/random
java       5173       interview   12r      CHR    1,8      0t0   1051 /dev/random

interview@ip-10-0-0-127:~$ kill -STOP 1209

interview@ip-10-0-0-127:~$ java Multi3 RandomGenerator2
try 0
total lines read in bufferedreader : 100000
available entropy of the system: 12
69193021624 nanoseconds
69 seconds
************************
try 1
total lines read in bufferedreader : 100000
available entropy of the system: 52
22073511965 nanoseconds
22 seconds
************************
try 2
total lines read in bufferedreader : 100000
available entropy of the system: 66
47903609633 nanoseconds
47 seconds
************************
try 3
total lines read in bufferedreader : 100000
available entropy of the system: 54
73525282734 nanoseconds
```

```
73 seconds
************************
try 4
total lines read in bufferedreader : 100000
available entropy of the system: 47
51822368859 nanoseconds
51 seconds
************************
try 5
total lines read in bufferedreader : 100000
available entropy of the system: 44
54067620346 nanoseconds
54 seconds
************************
try 6
total lines read in bufferedreader : 100000
available entropy of the system: 73
90604146627 nanoseconds
90 seconds

interview@ip-10-0-0-127:~$ watch cat /proc/sys/kernel/random/entropy_avail
interview@ip-10-0-0-127:~$ kill -CONT 1209
```