Advanced Programming: JVM & Tools

Dario Bekic

Question 1

Write a simple Java program JustCreate that in a very long loop creates at each iteration one object, discarding immediately any reference to it.
Every 1000 iterations the program must print the number of objects created since the beginning and must pause for 50 milliseconds (use e.g. Thread.sleep()).
Inspect the program execution with VisualVM, and try to tune the parameters so that the heap occupancy profile has the shape of a shark's teeth

To resemble a Shark's teeth we need an increasing phase, which corresponds to the allocation phase, and then a decreasing one.
To actually see the shark's teeth we need to decrease the difference between the heap allocated size for the jvm and the heap used by our **java.lang.Object**s.
To simplify we chose the serial garbage collector, which has an easier-to-predict behaviour w.r.t the newer and parallel counterparts. With repeated tests, the application consumes $\approx$ 5Mb at the start of the application and at each sweep made by the GC, thus a 2Mb for Eden's space seemed appropriate.

```
~$ java -XX:+UseSerialGC -Xmx8m JustCreate
```

Listing 1: How to launch JustCreate

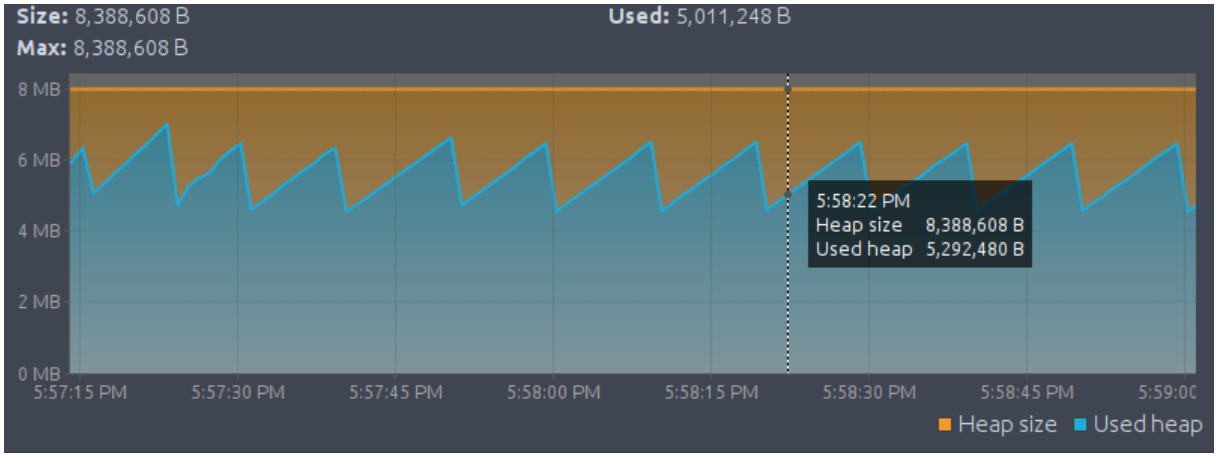

Figure 1: Sharks teeth

Question 2

As we have seen recently, the JVM has two instructions that can be used to compile a switch statement: tableswitch and lookupswitch. Taking inspiration from the examples seen at lesson, try to determine when your Java compiler uses tableswitch and when it uses lookupswitch. Does this depend only on the distance between the smallest and the largest constants in the case clauses? Are other criteria considered as well? Use javap -c -v ClassName.class or godbolt.org to disassemble the compiled bytecode.

I found here the source code which chooses how to pick which instruction to emit.

```
// Determine whether to issue a tableswitch or a lookupswitch
// instruction.
        long table_space_cost = 4 + ((long) hi - lo + 1); // words
        long table_time_cost = 3; // comparisons
        long lookup_space_cost = 3 + 2 * (long) nlabels;
        long lookup_time_cost = nlabels;
        int opcode =
            nlabels > 0 &&
            table_space_cost + 3 * table_time_cost <=
            lookup_space_cost + 3 * lookup_time_cost
            ?
            tableswitch : lookupswitch;
```

It seems the difference between highest (hi) and lowest (lo) is only the space factor. The costants 3 and 4 are probably due to the space needed for emitting the two `switch` instructions. The space cost for the continuos jump table is correctly calculated as $hi - lo + 1$ considering the default clause and all other labels.
Lookup time cost is linear even though the JVM sorts the labels which could enable a binary search, however, it seems that the benefit for such implementation is not considered.

Question 3

Run the program WrongQueue and inspect its behaviour using VisualVM. Can you explain the continuous growth of the heap? Find the code causing the bug and fix it. Clearly, there are several ways to fix the bug: for this exercise, try to find one with minimal "edit distance" from the source, i.e. the one obtained with the minimum amount of character insertions or deletions.

`current` and `head` represent semantically the same thing, but in the code of procedure `dequeue` only `current` is updated. Since `head` keeps reference to the object pointed before by `current` it cannot be garbage collected. I believe the minimum edit distance to make the program correct, in the sense that WrongQueue behaves like a FIFO Queue, is adding `= head` in the second instruction of the `dequeue` procedure.

```
public String dequeue(){
        String result = null;

        if(current != null){
            result = current.data;
            current = head = current.next;
        }

        return result;

    }
```

Question 4

In the last lesson, the lecturer claimed that when compiling a switch based on a String value, the Java compiler uses a hashing function for strings. Verify that this is true. Also, try to understand the bytecode produced by the compiler: if the hash value coincides with the hash of a string of a case clause, apparently an additional comparison between the two strings is performed. Why is this necessary? Using javap or godbolt.org, disassemble the compiled code to see if the lecturer is right. Also, search in the Java or JVM Specification the precise place where this compilation scheme is prescribed. Goal: Reading disassembled bytecode; checking the Java/JVM specification.

Indeed this happens, as shown in the `visitStringSwitch` method here and here.
The additional check is necessary because of potential problems regarding hashing, e.g. collisions, so an additional check is made using the `String.equals` method.

Question 5

Exploring threads. Explore the threads of a running application using VisualVM. Write a simple multi-threaded program which spawns a new thread every few second. Monitor the threads, their state and the code they are executing using the thread monitor of VisualVM and the Thread Dump facility. Look for information about the threads dedicated to JIT compilation and to GC

To JIT compilation are dedicated two threads, taken from a random thread dump:

```
"C1 CompilerThread0" #17 [27735] daemon prio=9 os_prio=0 cpu=1276,54ms
    elapsed=79,64s tid=0x00007dbbdc15d550 nid=27735 waiting on condition  [0
    x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task

   Locked ownable synchronizers:
        - None
```

and

```
"C2 CompilerThread0" #14 [27734] daemon prio=9 os_prio=0 cpu=3451,28ms
    elapsed=79,64s tid=0x00007dbbdc15be40 nid=27734 waiting on condition  [0
    x0000000000000000]
   java.lang.Thread.State: RUNNABLE
   No compile task

   Locked ownable synchronizers:
        - None
```

For Garbage Collection we have the `Finalizer` thread which performs method `Finalize` on unreachable objects.
It is possible to get the logs of the JIT threads adding the following parameters to the `java` command:

Listing 2: Parameters for logging JIT Threads activities

```
XX:+UnlockDiagnosticVMOptions -XX+LogCompilation
```

Question 5 continued on next page...

Each thread in my application calls the following method:

```
public int stupid() {
    int i=0;
    while (i<10000) {
        i = i + 1;
    }
    System.out.println("done");
}
```

The first reference we find about the method `stupid` is the following:

```
<task_queued compile_id='6' compile_kind='osr' method='lol stupid ()V'
   bytes='25' count='7' backedge_count='60416' iicount='7' osr_bci='2'
   level='3' stamp='2,646' comment='tiered' hot_count='60416'/>
```

We observe that it is queued an OSR(i.e. On Stack Replecement) which is a request to the immediate replacement of interpretation with a compiled version of the method, this is done because the `while` is detected having lot of iterations(`backedge_count='60416'`).