Michal Skalský
A08N0025P

Java Virtual Machine

1 Overview

First of all JVM is currently far from being complete. I try to describe what is working along with the information about stuff that needs to be implemented. It is important to note that I expect good knowledge of java virtual machine mechanics as well as class file structure. This information is freely available and I won't go into much detail here.

2 Current version

Current version of JVM can load java class file and execute its methods within following limitations.

- Class can be derived from Object class or other user created class. No classes from java library are supported. That means for example no strings, no exceptions and of course much more.
- Because Exception class is not supported, whole exception handling isn't working.
- Threads are not supported.
- Multi-dimensional arrays aren't supported.
- Only couple of instructions is actually implemented (about 50 out of 200). I implemented only these instructions that I needed for testing. But all the difficult, important ones are there (memory allocation, method invocation, return, object manipulation…)
- Utf-8 is handled as simple char (1-byte) array.

And now what is working.

- Object creation and manipulation (new, field read/write…)
- Method invocation and termination.
- Static members.
- Arrays.
- All base data types including double and long.
- Certain arithmetic operations.
- Garbage collection.

3 Implementation

The project is split into several major parts. Virtual machine itself, stack, heap, garbage collector and java class representation. All of them will be covered in more detail in upcoming chapters.

3.1 Class representation

In the code, java class is represented by c++ class cJavaClass. This class is responsible for loading class from file and it has methods to further process it. I'll present only brief description of java class file structure. Detailed info can be found freely on the Web. I won't cover the self-explanatory parts of class file like magic constant or version representation. Instead I will focus on the important stuff. First we need to load constant pool. There is special structure for every kind of constant. They are all named like "CONSTANT_...". For example CONSTANT_MethodRef. Design of these structures follows

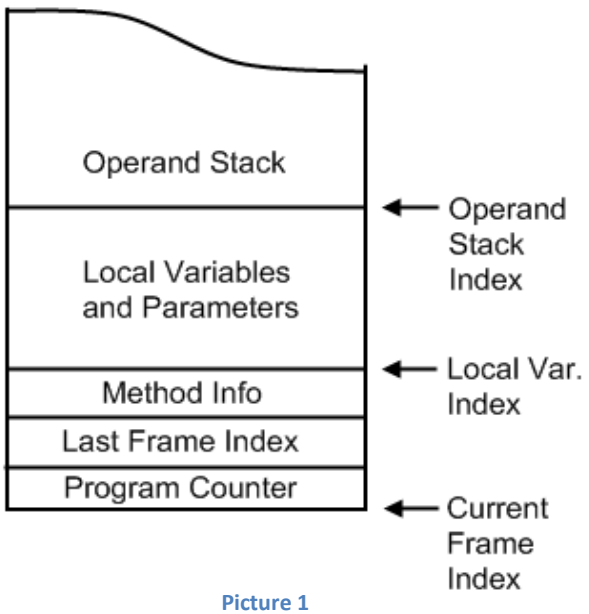
closely class file format. Only sometimes I added pointer[1] or two to speed up things a bit during bytecode execution. It is important to note that first constant in constant pool is never used[2] and that double and long constants take up two constant pool entries. Next part of class file worth noting is field declaration. Class's fields are loaded one by one. They are internally represented by FIELD_INFO structure. Again, FIELD_INFO closely follows class file structure. This time I added information about field's size, whether field is reference or value type or whether field is a constant. Next methods are loaded. Internally structure METHOD_INFO. And again, METHOD_INFO is heavily based on class file structure. On top of that it stores information about bytecode[3] and exceptions. Now we have most of the information we need. After all the data is loaded from file we still need to resolve references from/to constant pool and we need to compute class's size in bytes. We have to go through all constants in constant pool and replace indices by direct pointers when necessary. During this process we may find another class. If that's the case we ask VM for pointer to that class. VM is then responsible for loading that class or for simple return of a pointer in case that class was already loaded. If class cannot be found VM returns an error. When all the symbols are resolved we have to compute class's size in bytes. We do this separately for static and normal fields. At the very end of class loading we allocate memory[4] for static fields and invoke class initialization method (clinit). For further details consult class file description or source code.

3.2 Stack

Stack is represented by two separate arrays. First array contains stack vales – structure STACK_ITEM. Second array contains byte flags. These flags tell if certain stack value is reference to the heap or not. This information is vital for correct working of garbage collector. Stack value is structure big enough[5] to hold any data type except double and long. 8-byte values take up two stack values. First go high bytes and then low bytes. Stack is implemented in cStack class.

3.2.1 Method Frame

Stack is divided in to method frames. Top frame is frame of currently executed method. Whenever during the execution a method is called new frame is pushed on top of stack. When method returns its frame is popped from stack. Method frame is illustrated on picture 1. On the picture, "Program Counter" means pc of previous method (return address). It is used when current method returns as a program counter. "Last Frame Index" is index to stack where previous frame begins. "Method Info" is pointer to current method's info. You can get class pointer through it too. This information is needed when reconstructing previous frame after current frame was popped. "Local



Picture 1

[1] All references are handled trough indices to constant pool. Pointers speed things up a lot.
[2] It has value of null.
[3] Bytecode is normally one of the method's attributes.
[4] Memory for static fields is allocated directly from OS. It is NOT subject to memory management.
[5] Should be 4-bytes

Variables and Parameters" is set of stack values. They will be used to store parameters and local variables during execution. "Operand Stack" is set of stack values used to store intermediate data during expression calculation. Size of Operand stack and Local variables is determined during compilation and passed through class file.

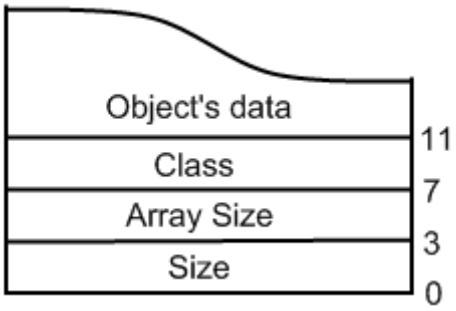
When method is invocated its frame is pushed exactly behind last currently used operand stack value, so no stack value is wasted. After new frame is pushed its values and flags are set to zero.

3.2.2 Stack reference flags

As I said before every stack value has associated a flag. This flag tells garbage collector if this value is reference to object or not. It is EXTREMELY important that flags are correctly set. Every instruction that writes data to stack MUST make sure to set appropriate flag.

3.3 Heap

Heap serves as memory pool for allocating new objects. It is split into two equal parts by design[6]. Only one half is used at any given time. Allocated objects are on the bottom and there is free space on the top of currently used half. Allocation proceeds always on top of a heap. No search for free space is necessary, so the allocation is very quick. When there is no more free space, garbage collector is called. It seeks and transfers all live objects to the bottom of the other half. It means that heap halves are basically switched upon every garbage collector invocation. More details about garbage collection are given in separate chapter. Every heap allocated object shares the same header. Object representation is outlined in picture 2. "Size" means overall object's size in bytes including header. "Array Size" means array length. It is one for non-arrays. "Class" means pointer to class this object is instance of. It has NULL value for array of value data type. Behind header follows object's data itself. Heap is implemented in cHeap class.



Picture 2

3.4 Virtual Machine

Virtual machine is responsible for class management and for bytecode execution. First is achieved by STL[7] map[8] where key is class's name and value is class itself. Every newly created class is added to this map so it can be easily accessed during VM's lifetime. Second responsibility is bytecode execution. In order to execute bytecode every instruction must be implemented somewhere in VM. Common approach to this problem is to create one big switch where every case corresponds to certain instruction. We decide which one to process based on opcode then. I decided to go in little bit different direction instead. Every instruction has its own function (i.e. it is implemented in one specific function). VM has array of pointers to that specific function. Opcode is simple index to that array. I believe this solution must be faster because we don't have to go through several (possibly a LOT) conditions to find the code we need. Of course there's same time associated with function call but I would say it is faster in the end. In any case that's the way I chose. The whole process of instruction execution is then simple loop. On top of that VM needs heap, stack and information

[6] Current design. It can be easily changed.
[7] Standard Template Library
[8] I believe it is red black tree internally.

about current class[9]. Every thread has to have its own stack but then again threads aren't supported so there is only one global stack. On the other hand heap can be easily shared among all threads. VM is implemented in class cJVM. It has function to load main method from class file and execute it.

As stated before current version doesn't support strings nor does it support java library. That means no standard output as well. That presents a little bit difficulty because you can't really get any sort of output out of the VM. And you definitely need that if you want to ensure proper working. For my debugging purposes I manually write value of the first variable in main method to the stdout. I know it is far from being elegant or even universal but it worked for me.

3.5 Run-time errors

As stated before, exception handling isn't supported. There is temporary solution to make sure code interpretation is safe. Normally every time an error occurs, exception is thrown. Current solution breaks the execution and prints error message to standard output instead. It is used to check array indices, null references or out-of-memory exceptions.

3.6 Garbage Collector

Garbage collector is responsible for freeing of memory that is not used anymore. Its goal is to locate all live objects and to transfer them to the bottom of the heap so that there's continuous free block of memory on the top. Live object is object that can be reached from the code. Dead objects aren't referenced by any variable or field and thus are unreachable. As stated before live objects are relocated to the bottom of the other half of the heap. No search for free space is necessary that way and whole process is rather fast. I try to explain how live objects are located now. In order for object to be live, it has to be referenced by some variable or field. Garbage collector must go through stack values where variables, arguments and operands lie. Every stack value has flag associated with it and that flag tells garbage collector if current stack value represents reference to the heap. If this is the case we have found live object. For every live object, garbage collector has to check for internal references. For example if object is an array of other objects, GC has to go through all valid references in this array and relocate them too. Same applies to class's fields. This recursive approach leads to location of all live objects. Another problem that GC needs to solve is multiple references to the same object. When object is encountered for a first time, its data is relocated and reference is updated. Unfortunately other references to the same object will still point to its old destination. Valid solution would be to search for all references to this object and update them all. But that would take ages. GC will alter header of the object in its former place instead. Two parts of object's header need to be changed. ArraySize is set zero to signal that object was already relocated and Size is changed to hold new address of the object. It means that during the execution GC checks ArraySize first to find out if object was already relocated. If so, it only updates reference with new address stored in Size field. It is fast and elegant.

4 The Code

The source code is "heavily" commented. At least on my standards. It shouldn't be too difficult to follow it. In any case I can be reached at mskalsky@students.zcu.cz. The development took place in MS Visual Studio 2008. It means it works perfectly with MS C++ compiler. I did my best not to use any Windows library, so it should be portable. Special care must be taken when running on different

[9] Class to which current method belongs.

computers thou. Java class file uses big-endian format and this project uses little-endian format. It wouldn't run on Motorola CPU or some older Mac for example.

5 The Code map

I present here what individual files contain. It is only a basic "code map" but hopefully enough to get you started. Brief function descriptions are directly in the code, so I won't list them here. List is alphabetically sorted.

- Class.h/cpp – It holds definition of structures for constant pool constant types, FIELD_INFO, METHOD_INFO and cJavaClass.
    - FIELD_INFO – Information about one field.
    - METHOD_INFO – Information about one method.
    - cJavaClass – represents one class file.
- Common.h/cpp – Contains common macro definitions and common functions. Nothing really interesting here.
- OpCodes.h – Should contain macro definition for every opcode. The list is incomplete.
- Heap.h/cpp – It defines structure OBJECT and class cHeap
    - OBJECT – it is a header of every heap allocated object. See 3.3 for more.
    - cHeap – implementation of heap.
- JVM.h/cpp – Only one class in here. cJVM.
    - cJVM - represents java virtual machine.
    - Main function (entry point) is in cJVM.cpp for debug purposes.
- Stack.h/cpp – It defines STACK_ITEM structure and cStack class.
    - STACK_ITEM – represents one stack value. A union of all possible values.
    - cStack – it is a implementation of a stack.