What's the calling convention for the Java code in Linux platform?

Asked 2 years, 11 months ago Active 2 years, 11 months ago Viewed 1k times

4

2

We know the calling convention that "first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX (R10 in the Linux kernel interface[17]:124), R8, and R9" for the c/c++ code in the Linux platform based on the following article.https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions

However what's the calling convention for the Java code in Linux platform (suppose the JVM is hotspot)? The following is example, what's registers store the four parameters?

```
protected void caller( ) {
callee(1,"123", 123,1)
}

protected void callee(int a,String b, Integer c,Object d) {

}
```

java jvm

edited Jan 17 '17 at 10:13
Florian Albrecht
1,865 15 22

asked Jan 17 '17 at 9:39
YuFeng Shen
961 7 23

2 I do not think that they are stored in any registers. Java is a virtual machine, having its own registers and stacks. – Florian Albrecht Jan 17 '17 at 9:44

2 I suppose you mean conventions, not conversations? Java is a VM language. It has its own machine definition spec, but there is no necessary connection between the X86 registers and the JVM registers. – RealSkepticJan 17 '17 at 9:44

However ,finally the Java code would be jitted, so they finally would be stored in the X86 registers. However I don't know how Jit compiler would assign the Java parameters to X86 registers. I have no idea if there are any rules exists. – YuFeng Shen Jan 17 '17 at 10:13

@FlorianAlbrecht : JVM is not using registers, it is completely stack based VM – rkosegi Jan 17 '17 at 18:54

2 Answers

It is not specified how JVM calls Java methods internally. Various JVM implementations may follow

10

different calling conventions. Here is how it works in **HotSpot JVM** on **Linux x64**.

- A Java method may run in interpreter or it can be JIT-compiled.
- Interpreted and compiled code use different calling conventions.

1. Interpreter method entry

Each Java method has an entry point into the interpreter. This entry is used to jump from an interpreted method to another interpreted method.

- All arguments are passed on stack, from bottom to top.
- `rbx` contains a pointer to `Method*` structure - internal metadata of a method being called.
- `r13` holds `sender_sp` - stack pointer of a caller method. It may differ from `rsp + 8` if `c2i` adapter is used (see below).

More details about interpreter entries in HotSpot source code: templateInterpreter_x86_64.cpp.

2. Compiled entry

A compiled method has its own entry point. Compiled code calls compiled methods via this entry.

- Up to 6 first integer arguments are passed in registers: `rsi`, `rdx`, `rcx`, `r8`, `r9`, `rdi`. Non-static methods receive `this` reference as the first argument in `rsi`.
- Up to 8 floating point arguments are passed in `xmm0` ... `xmm7` registers.
- All other arguments are passed on stack from top to bottom.

This convention is nicely illustrated in assembler_x86.hpp:

```
|------------------------------------------------------|
| c_rarg0   c_rarg1  c_rarg2 c_rarg3 c_rarg4 c_rarg5   |
|------------------------------------------------------|
| rcx       rdx      r8      r9      rdi*    rsi*      | windows (* not a c_rarg)
| rdi       rsi      rdx     rcx     r8      r9        | solaris/linux
|------------------------------------------------------|
| j_rarg5   j_rarg0  j_rarg1 j_rarg2 j_rarg3 j_rarg4   |
|------------------------------------------------------|
```

You may notice that Java calling convention looks similar to C calling convention but shifted by one argument right. This is done intentionally to avoid extra register shuffling when calling JNI methods (you know, JNI methods have extra `JNIEnv*` argument prepended to method parameters).

3. Adapters

Java methods may have two more entry points: `c2i` and `i2c` adapters. These adapters are pieces of dynamically generated code that convert compiled calling convention to interpreter layout and vice versa. `c2i` and `i2c` entry points are used to call interpreted method from compiled code and compiled method from interpreted code respectively.

P.S. It does not usually matter how JVM calls methods internally, because these are just implementation details opaque to end-user. Moreover, these details may change even in a minor

JDK update. However, I know at least one case when the knowledge of Java calling convention may appear useful - when analysing JVM crash dumps.

answered Jan 17 '17 at 20:57

apangin
61.6k 9 130 164

Thank you so much apangin, you mentioned Java calling convention is useful when analysing the dump, and as I know the gdb just call print the c/c++ call stack by the command backtrace , then frame N(N is the thread number). and x/20i $pc-64 to check the assembly code. However the Java method can not be printed in the call stack from gdb, so how to check the assembly code for the Java method from the dump? – YuFeng Shen Jan 18 '17 at 2:51

@Hermas HotSpot Serviceability Agent can do this for you. I've mentioned it in another answer. – apangin Jan 18 '17 at 22:29

Please kindly help to check this related question stackoverflow.com/questions/42313695/… – YuFeng Shen Feb 18 '17 at 10:00