About Hotspot JVM Frame

Aug 25, 2019

The following summarizes the frames used when interpreting in the Hotspot JVM.

Article content

- OpenJDK version: Jdk11u of changeset 51892: e86c5c20e188
- OS: Something from the Linux distribution
- CPU architecture: x86_64

It is written on the assumption of the environment.

About frame

The Hotspot JVM's Template Interpreter defines a class called frame. This is a class that represents the stack frame used when processing Java methods, including, for example, local variables, method parameters, and return addresses. Prepared at the start of each Java method's interpret and destroyed at the end.

The actual state of frame is CPU-dependent, and on x86 it has the following structure (`src/hotspot/cpu/x86/frame_x86.hpp`quoted from the comment, lower address in the figure).

```
// ------------------------------ Asm interpreter ----------------------------------------
// Layout of asm interpreter frame:
//    [expression stack      ] * <- sp
//    [monitors              ]   \
//     ...                        | monitor block size
//    [monitors              ]   /
//    [monitor block size    ]
//    [byte code pointer     ]                   = bcp()               bcp_offset
//    [pointer to locals     ]                   = locals()            locals_offset
//    [constant pool cache   ]                   = cache()             cache_offset
//    [methodData            ]                   = mdp()               mdx_offset
//    [Method*               ]                   = method()            method_offset
//    [last sp               ]                   = last_sp()           last_sp_offset
//    [old stack pointer     ]                     (sender_sp)         sender_sp_offset
//    [old frame pointer     ]   <- fp           = link()
//    [return pc             ]
//    [oop temp              ]                     (only for native calls)
//    [locals and parameters ]
//                               <- sender sp
// ------------------------------ Asm interpreter ----------------------------------------
```

Each element of frame:

- expression stack
  - rsp? at completion of frame creation (= own memory address)
- monitors
  - do not know. Maybe monitor lock related?
- monitor block size
  - Maybe the number of monitors above
- byte code pointer
  - Points to the Java bytecode address to be interpreted next
  - Sometimes abbreviated as bcp
- pointer to locals
  - Parameters start address on stack
- constant pool cache
  - do not know. Maybe there is a mechanism to cache the information of constant pool and it is related

        - cache should reduce the number of returns from interpreter to VM runtime
- methodData
    - do not know. Collect statistical information of methods and use them for optimization such as JIT compilation?
- Method *
    - interpret A pointer to the target method
- last sp
    - 0x00 when creating a frame
- old stack pointer
    - rsp before frame creation
    - Also called sender sp
- old frame pointer
    - rbp before frame creation
- return pc
    - Interpret return on completion
    - (Address on native code, not Java byte code)
- oop temp
    - do not know
- locals and parameters
    - Area where local variables and parameters of the method are placed

frame of available applications is the stack frame at the time as Java methods interpret written on it, `frame_x86.hpp`in comments

> A frame represents a physical stack frame (an activation) .Frames can be C or Java frames, and the Java frames can be interpreted or compiled.

It is explained that there are other uses. It is difficult if you do not understand the difference between C frame and Java frame.

Also, in the interpreter of Hotspot JVM, there is Cpp Interpreter other than Template Interpreter mentioned here (previous post), but it seems that this frame is not used there. According to the description of RuntimeOverview # interpreter , Cpp Interpreter manages the stack frame separately from the native stack (so there is overhead).

> it (CppInterpreter) uses a separate software stack to pass Java arguments, while the native C stack is used by the VM itself.A number of JVM internal variables, such as the program counter or the stack pointer for a Java thread, are stored in C variables, which are not guaranteed to be always kept in the hardware registers.Management of these software interpreter structures consumes a considerable share of total execution time.

Source reading of main start processing

As an example of using frame, `public static void main`let's look at how the Hotspot JVM starts interpret, the entry point for Java programs .

`JavaCalls::call`That 's where the main method interpret starts . We recognize that this method is used when migrating from VM runtime to interpret Java bytecode.

I feel that the relationship between VM runtime and interpreter is similar to the relationship between OS and user program. When you start interpret (for example to execute the main method), the VM runtime prepares it and then lets the interpreter start, similar to the way the OS loads user programs into memory. However, when the interpreter needs difficult processing (see eg constant pool), it asks the VM runtime for processing once and returns again, which is similar to a system call.

`JavaCalls::call` In order to understand the transition steps from VM runtime to interpreter in Linux, here we will look at the following two.

- (1) Save the state before interpret (generated by `generate_call_stub`)
- (2) Creating a frame (generated by `generate_normal_entry`)

Both of these processes are performed using native code generated during VM initialization. Hereafter, for convenience, this native code is called call_stub and normal_entry code, respectively. `JavaCalls::call`Reading, following `StubRoutines::call_stub`executes the process of in call_stub, in its internal `entry_point`running the normal_entry code that was passed as, begin to interpret the Java method from there, and so on.

```
// in src/hotspot/share/runtime/javaCalls.cpp

void JavaCalls::call_helper(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS) {
      ...
      StubRoutines::call_stub()(
        (address)&link,
        // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
        result_val_address,          // see NOTE above (compiler problem)
        result_type,
        method(),
        entry_point,
        args->parameters(),
        args->size_of_parameters(),
        CHECK
      );
      ...
}
```

The order changes, but first `entry_point`, follow the true identity, confirm that a frame is created with normal_entry, and then check the code generation part and its contents for call_stub as well.

The identity of entry_point

`JavaCalls::call_helper` Now brings the entry point for the interpreter as follows:

```
  // in JavaCalls::call_helper

  // Since the call stub sets up like the interpreter we call the from_interpreted_entry
  // so we can go compiled via a i2c. Otherwise initial entry method will always
  // run interpreted.
  address entry_point = method->from_interpreted_entry();
  if (JvmtiExport::can_post_interpreter_events() && thread->is_interp_only_mode()) {
    entry_point = method->interpreter_entry();
  }
```

Method has multiple types of entry point addresses. I think this is probably for interpreter or JIT compiled code.

Here, we want to see the process of the interpreter `method->interpreter_entry`. This simply `address _i2i_entry`brings in the Method's possession .

```
  // in method.hpp

  // Entry point for calling both from and to the interpreter.
  address _i2i_entry;           // All-args-on-stack calling convention
```

It looks like how this is generated `Method::link_method`. I think that link refers to the operation mentioned in 5.4 Linking of JVM spec 11.

```
  // in method.hpp and method.cpp

  // setup entry points
  //
  // Called when the method_holder is getting linked. Setup entrypoints so the method
  // is ready to be called from interpreter, compiler, and vtables.
  void link_method(const methodHandle& method, TRAPS) {
      ...
      if (!is_shared()) {
        assert(adapter() == NULL, "init'd to NULL");
        address entry = Interpreter::entry_for_method(h_method);
        assert(entry != NULL, "interpreter entry must be non-null");
        // Sets both _i2i_entry and _from_interpreted_entry
```

```
        set_interpreter_entry(entry);
      }
      ...
  }
```

`set_interpreter_entry`The `i2i_entry`only important thing `Interpreter::entry_for_method(h_method)`is the entry point address returned from , because is only set to etc.

```
  // in abstractInterpreter.hpp

  static address entry_for_method(const methodHandle& m) {
      return entry_for_kind(method_kind(m));
  }

  static address entry_for_kind(MethodKind k) {
      assert(0 <= k && k < number_of_method_entries, "illegal kind");
      return _entry_table[k];
  }
```

`Interpreter::_entry_table`You can see that we are looking for an address to return from. `MethodKind`This is because the following values are defined in the enum type. Speaking of the main method you are looking at, it `zerolocals`should be fine.

```
  enum MethodKind {
    zerolocals,                                 // method needs locals initialization
    zerolocals_synchronized,                    // method needs locals initialization & is synchronized
    native,                                     // native method
    native_synchronized,                        // native method & is synchronized
    empty,                                      // empty method (code: _return)
    accessor,                                   // accessor method (code: _aload_0, _getfield, _(a|i)return)
    abstract,                                   // abstract method (throws an AbstractMethodException)
    method_handle_invoke_FIRST,                 // java.lang.invoke.MethodHandles::invokeExact, etc.
    method_handle_invoke_LAST                   = (method_handle_invoke_FIRST
                                                     + (vmIntrinsics::LAST_MH_SIG_POLY
                                                         - vmIntrinsics::FIRST_MH_SIG_POLY)),
    java_lang_math_sin,                         // implementation of java.lang.Math.sin   (x)
    java_lang_math_cos,                         // implementation of java.lang.Math.cos   (x)
    java_lang_math_tan,                         // implementation of java.lang.Math.tan   (x)
    java_lang_math_abs,                         // implementation of java.lang.Math.abs   (x)
    java_lang_math_sqrt,                        // implementation of java.lang.Math.sqrt  (x)
    java_lang_math_log,                         // implementation of java.lang.Math.log   (x)
    java_lang_math_log10,                       // implementation of java.lang.Math.log10 (x)
    java_lang_math_pow,                         // implementation of java.lang.Math.pow   (x,y)
    java_lang_math_exp,                         // implementation of java.lang.Math.exp   (x)
    java_lang_math_fmaF,                        // implementation of java.lang.Math.fma   (x, y, z)
    java_lang_math_fmaD,                        // implementation of java.lang.Math.fma   (x, y, z)
    java_lang_ref_reference_get,                // implementation of java.lang.ref.Reference.get()
    java_util_zip_CRC32_update,                 // implementation of java.util.zip.CRC32.update()
    java_util_zip_CRC32_updateBytes,            // implementation of java.util.zip.CRC32.updateBytes()
    java_util_zip_CRC32_updateByteBuffer,       // implementation of java.util.zip.CRC32.updateByteBuffer()
    java_util_zip_CRC32C_updateBytes,           // implementation of java.util.zip.CRC32C.updateBytes(crc, b
    java_util_zip_CRC32C_updateDirectByteBuffer,// implementation of java.util.zip.CRC32C.updateDirectByteBu
    java_lang_Float_intBitsToFloat,             // implementation of java.lang.Float.intBitsToFloat()
    java_lang_Float_floatToRawIntBits,          // implementation of java.lang.Float.floatToRawIntBits()
    java_lang_Double_longBitsToDouble,          // implementation of java.lang.Double.longBitsToDouble()
    java_lang_Double_doubleToRawLongBits,       // implementation of java.lang.Double.doubleToRawLongBits()
    number_of_method_entries,
    invalid = -1
  };
```

`Interpreter::_entry_table` Initialization is `TemplateInterpreterGenerator::generate_all`done as part of VM initialization .

```
// in src/hotspot/cpu/x86/templateInterpreterGenerator.cpp

void TemplateInterpreterGenerator::generate_all() {
  ...
#define method_entry(kind)                                              \
```

```
  { CodeletMark cm(_masm, "method entry point (kind = " #kind ")"); \
    Interpreter::_entry_table[Interpreter::kind] = generate_method_entry(Interpreter::kind); \
    Interpreter::update_cds_entry_table(Interpreter::kind); \
  }

  // all non-native method kinds
  method_entry(zerolocals)
  ...
}
```

`generate_method_entry`You can see the `generate_normal_entry`code generation for zerolocals .

```
// in src/hotspot/cpu/x86/templateInterpreterGenerator_x86.cpp

address TemplateInterpreterGenerator::generate_normal_entry(bool synchronized) {
    // determine code generation flags
    bool inc_counter  = UseCompiler || CountCompiledCalls || LogTouchedMethods;

    // ebx: Method*
    // rbcp: sender sp
    address entry_point = __ pc();

    const Address constMethod(rbx, Method::const_offset());
    const Address access_flags(rbx, Method::access_flags_offset());
    const Address size_of_parameters(rdx,
                                     ConstMethod::size_of_parameters_offset());
    const Address size_of_locals(rdx, ConstMethod::size_of_locals_offset());

    // 以下 assembler を使用したコード生成がずらずら
    ...
}
```

Therefore, generate_normal_entry is the native code generation part that creates the target frame. The native code generated by the generate_normal_entry is `-XX:+PrintInterpreter`in can be confirmed.

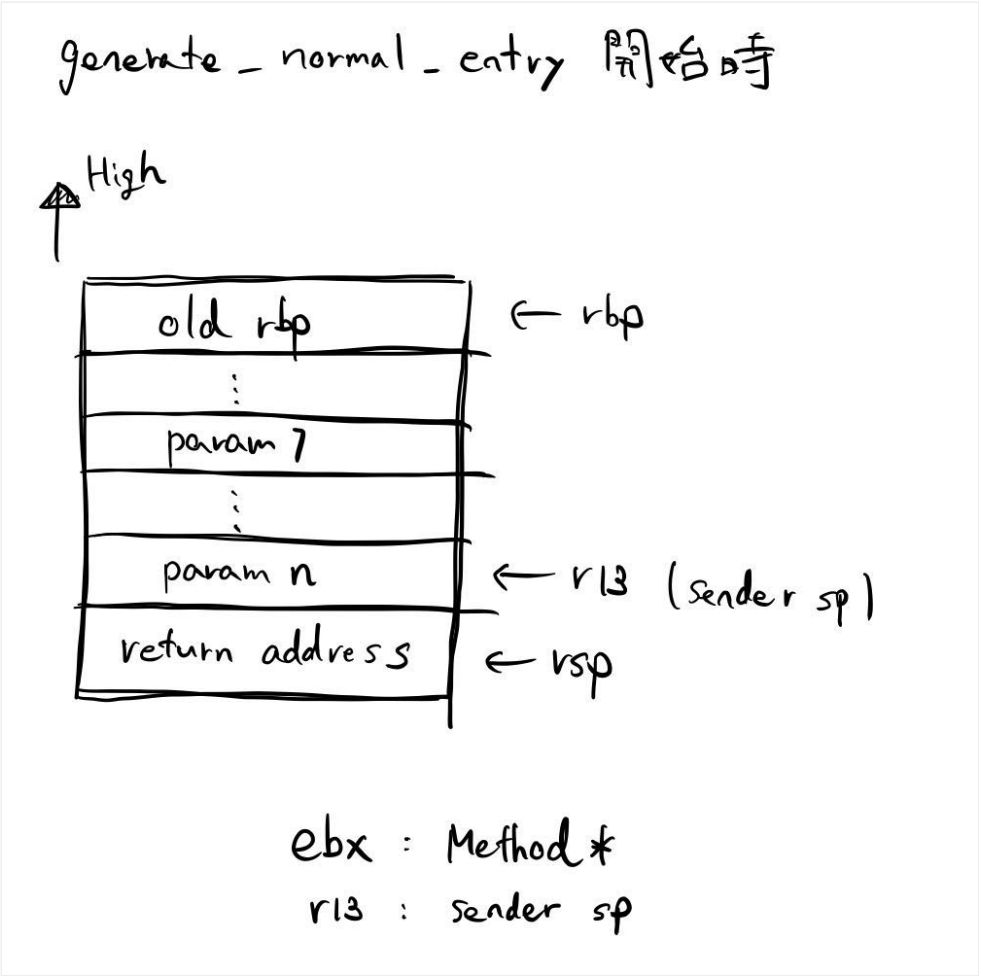
```
method entry point (kind = zerolocals)  [0x00007f2bd5016140, 0x00007f2bd5016f20]  3552 bytes

  0x00007f2bd5016140: mov    0x10(%rbx),%rdx
  0x00007f2bd5016144: movzwl 0x34(%rdx),%ecx
  0x00007f2bd5016148: movzwl 0x32(%rdx),%edx
  ...
```

normal_entry Check code contents

Here, we will roughly understand the contents of the code (normal_entry) generated by generate_normal_entry that was followed in the previous section.

The state of the register and stack at the start of normal_entry is as follows. The register is determined from the comment at the beginning of the generate_normal_entry method, and the stack is determined from the call_stub code described later.



Starting stack and registers

- old rbp points to the rbp register
  - rbp when calling call_stub
- param1-n is already set in call_stub
- (At this point) the destination of the r13 register is called the sender sp
  - Feeling like a stack pointer when calling normal_entry?
- The destination pointed by rsp is the return address set by the call instruction when calling normal_entry
- Reference to Java method to interpret in ebx register (`Method*`)

Check the stack and registers at the start of normal_entry with gdb. Use the following Java program as a sample.

```
class Locals {
    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        int z = 3;
        int sum = x + y + z;
        System.out.println("sum: " + sum);
    }
}
```

```
$ java Locals.java
$ javap -c -v Locals.class
  ...
  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=5, args_size=1
  ...
```

```
# breakpoint at the beginning of method entry point (kind=zerolocals)
(gdb) p $rsp
$3 = (void *) 0x7ffff59ed8f0
(gdb) p $rbp
$4 = (void *) 0x7ffff59ed960
(gdb) p ($rbp - $rsp) / 8
$7 = 14
(gdb) x /120xb $rsp
0x7ffff59ed8f0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00 # <- rsp
0x7ffff59ed8f8: 0x38    0x57    0x6f    0x19    0x07    0x00    0x00    0x00 # param1 (args) <- r13
0x7ffff59ed900: 0xa0    0x1f    0x00    0x00    0xff    0x7f    0x00    0x00
0x7ffff59ed908: 0x40    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed910: 0x00    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed918: 0x40    0xdb    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed920: 0x00    0xa8    0x01    0xf0    0xff    0x7f    0x00    0x00
0x7ffff59ed928: 0xe0    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed930: 0x40    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed938: 0xe8    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed940: 0x0a    0x00    0x00    0x00    0xff    0x7f    0x00    0x00
0x7ffff59ed948: 0x88    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00
0x7ffff59ed950: 0x40    0x61    0x01    0xe1    0xff    0x7f    0x00    0x00
0x7ffff59ed958: 0x10    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed960: 0xc0    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00 # <- rbp

# check param1
(gdb) x /40xb 0x07196f5738
0x7196f5738:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7196f5740:    0xa0    0x5c    0x02    0x00    0x02    0x00    0x00    0x00
0x7196f5748:    0xed    0xea    0x2d    0xe3    0xf6    0xea    0x2d    0xe3
0x7196f5750:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7196f5758:    0x40    0x08    0x00    0x00    0x03    0x00    0x00    0x00

(gdb) p /x $r13
$6 = 0x7ffff59ed8f8

(gdb) p ((Method *) $rbx)->_constMethod->_constants->_pool_holder->_name->as_C_string()
$1 = 0x7ffff00193d0 "Locals"
```

The output of the first rbp, rsp, and the stack area between them is consistent with the figure. 12 words from (rbp-1) to (rbp-12) are prepared in call_stub.

Looking at the destination pointed to by param1, you can see a structure like an array object.

The r13 register also points to the expected location.

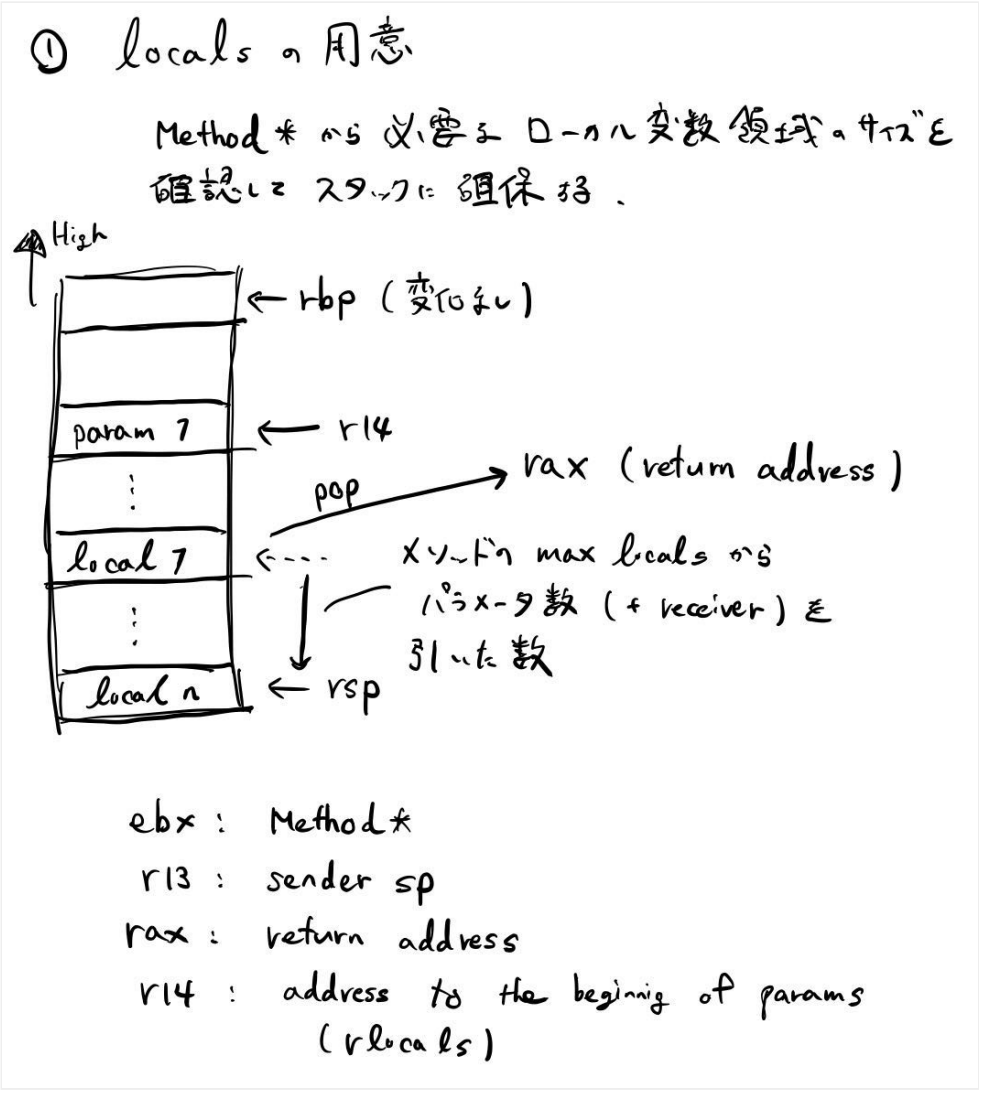
The class name that owns the method can be traced from the contents of the rbx register. `Method *`It looks good to see that the expected value is set.

1. Prepare locals

First, allocate a locals (local variable) area on the stack.

The required size is max locals-parameters-receiver (but receiver is not a static method). These can be seen as part of the method information when viewed with javap. In the above example of the Locals class, max locals is 5 and parameters are 1, so the required locals is 4.

In normal_entry, this information is `Method*`calculated from the ebx register .



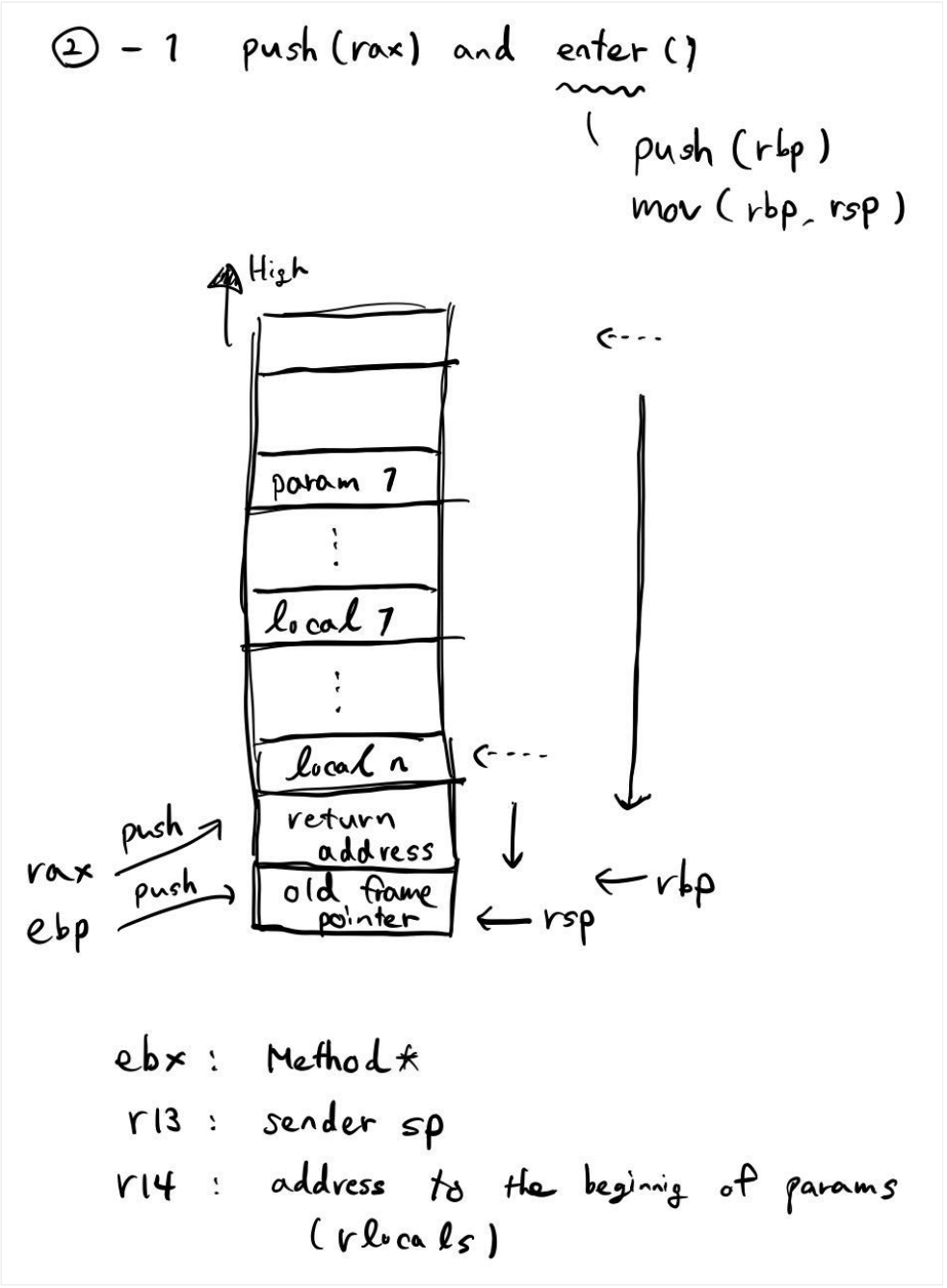
Prepare locals

- The r14 register points to the start of the parameters on the stack
- Pop return address into rax register before putting locals

2. Creating a frame

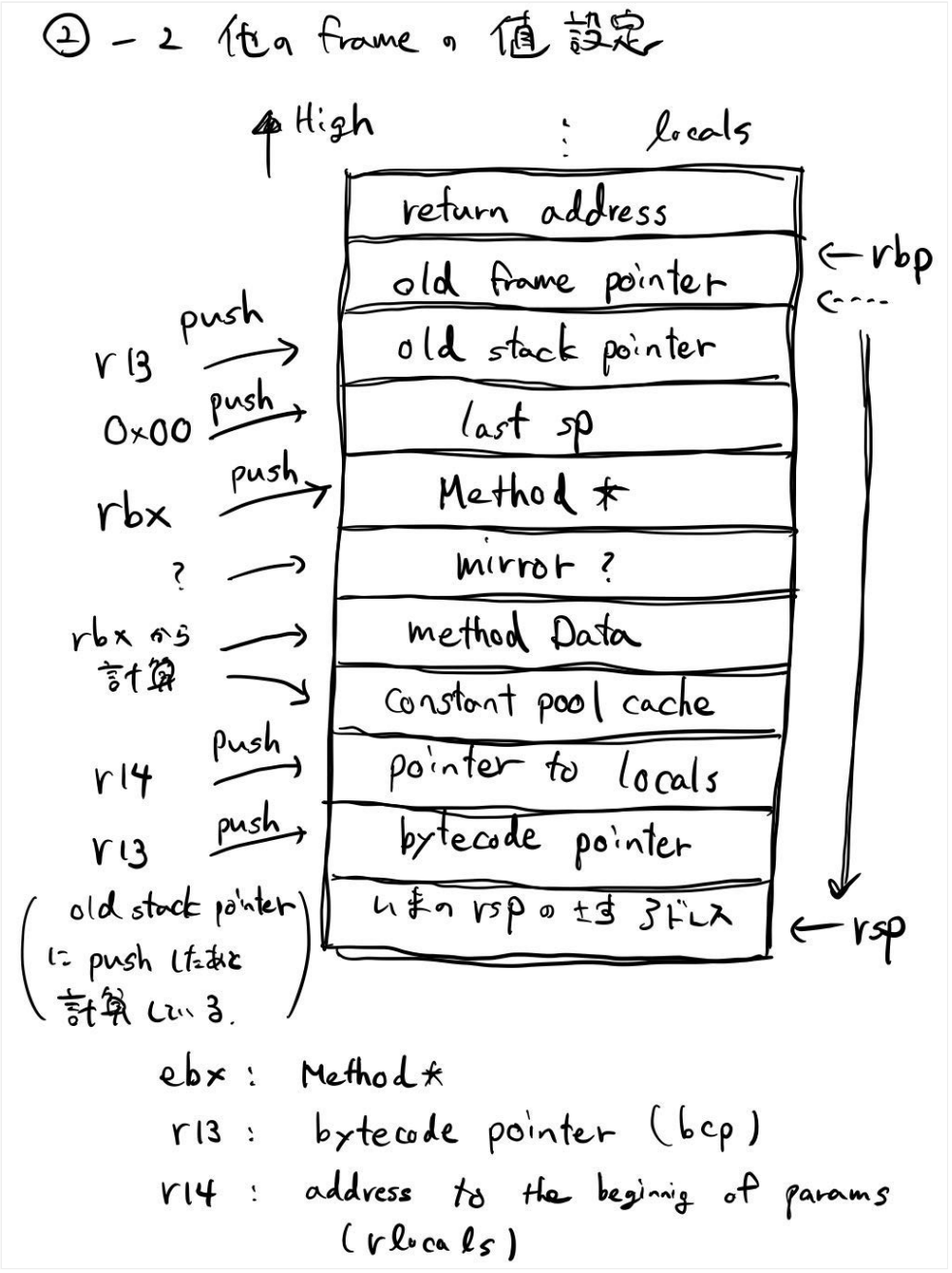
Set the value of frame after locals.

First, the return address of the saved rax register is returned to the stack, and the value of rbp is updated to the same value as rsp. As shown in the figure, rsp and rbp point to the previous rbp, which is called the old frame pointer in the frame.



push (rax) and enter ()

Set the rest of the frame. There are some elements that are not well understood, but they are summarized in the figure.



Set remaining frame values

- The r13 register is used twice here for different purposes
  - First push the stored sender sp (called old stack pointer in frame)
  - Next, calculate the address to the starting bytecode of the interpreted Method and push it (called the bytecode pointer in the frame)
- last sp is 0x00 at this time
  - I guess it was like setting the rsp at that point when calling another method internally?
- poor understanding of mirror, methodData, constant pool cache
  - And mirror `frame_x86.hpp`doesn't exist in the comment, but maybe the comment is wrong. It is present in the assembly at all, and exists in other CPU implementations

Let's look at the frame at this point with gdb. We use the Locals class as above.

```
# breakpoint after finishing generate frame
# set breakpoint judging by TemplateInterpreterGenerator::generate_fixed_frame
(gdb) p $rsp
$13 = (void *) 0x7ffff59ed880
(gdb) p $rbp
$14 = (void *) 0x7ffff59ed8c8
```

```
(gdb) p ($rbp - $rsp) / 8
$15 = 9
(gdb) x /160xb $rsp
0x7ffff59ed880: 0x80    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # <- rsp
0x7ffff59ed888: 0x58    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00 # bytecode pointer
0x7ffff59ed890: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # pointer to locals
0x7ffff59ed898: 0xf0    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00 # constant pool cache
0x7ffff59ed8a0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # methodData
0x7ffff59ed8a8: 0x98    0x52    0x6f    0x19    0x07    0x00    0x00    0x00 #  mirror
0x7ffff59ed8b0: 0x88    0x53    0xa2    0xcd    0xff    0x7f    0x00    0x00 # Method*
0x7ffff59ed8b8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # last sp
0x7ffff59ed8c0: 0xf8    0xd8    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old stack pointer
0x7ffff59ed8c8: 0x60    0xd9    0x9e    0xf5    0xff    0x7f    0x00    0x00 # old frame pointer <- rbp
0x7ffff59ed8d0: 0xf3    0x09    0x00    0xe1    0xff    0x7f    0x00    0x00 # return address
0x7ffff59ed8d8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local 4
0x7ffff59ed8e0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local 3
0x7ffff59ed8e8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local 2
0x7ffff59ed8f0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00 # local 1
0x7ffff59ed8f8: 0x38    0x57    0x6f    0x19    0x07    0x00    0x00    0x00 # param 1
0x7ffff59ed900: 0xa0    0x1f    0x00    0x00    0xff    0x7f    0x00    0x00
0x7ffff59ed908: 0x40    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed910: 0x00    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00
0x7ffff59ed918: 0x40    0xdb    0x9e    0xf5    0xff    0x7f    0x00    0x00
```

Roughly, frame settings are now complete.

3. dispatch_next

In normal_entry, processing to start interpret after frame creation continues. This is `InterpreterMacroAssembler::dispatch_next`a part of the processing by the code generated by

- Set dispatch_table address to rscratch1 (r10) register
- jmp from the rbcp (r13) and rscratch1 (r10) registers to the interpret entry point for the first Java bytecode

Interpret seems to start.

Identity of StubRoutines :: call_stub

Now that we have confirmed the frame creation process in normal_entry, let's look at call_stub.

`JavaCalls::call`In, `StubRoutines::call_stub()`call_stub was called.

```
// in src/hotspot/share/runtime/stubRoutines.hpp

class StubRoutines: AllStatic {
  ...
  static CallStub call_stub() {
    return CAST_TO_FN_PTR(CallStub, _call_stub_entry);
  }
  ...
}
```

`_call_stub_entry`Is generated by generate_call_stub defined by stubGenerator. generate_call_stub is called by generate_initial which is called from the constructor of StubGenerator.

```
// in src/hotspot/cpu/x86/stubGeneratro_x86_64.cpp

class StubGenerator: public StubCodeGenerator {
  ...
  address generate_call_stub(address& return_address) {
    ...
  }
  ...
  // Initialization
```

```
  void generate_initial() {
    // Generates all stubs and initializes the entry points
    ...
    StubRoutines::_call_stub_entry =
      generate_call_stub(StubRoutines::_call_stub_return_address);
    ...
  }
  ...
}
```

So the call_stub code is generated by generate_call_stub of stubGenerator. The actual generated code - `XX:+PrintStubCode`can be checked by.

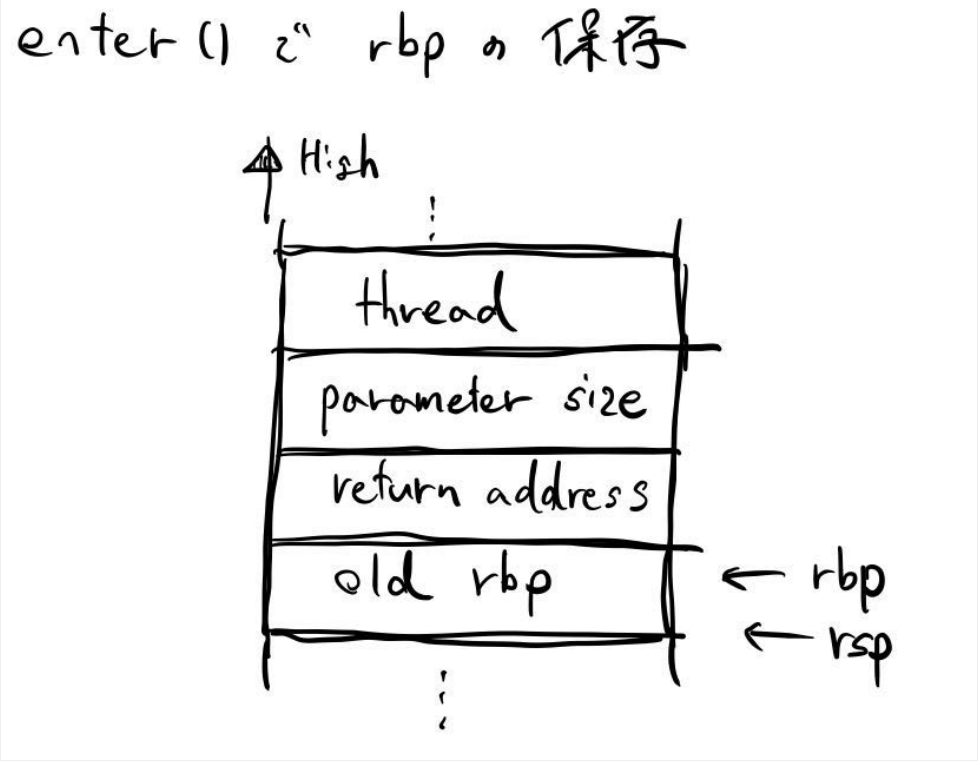
```
StubRoutines::call_stub [0x00007f1ddd0008e4, 0x00007f1ddd000c22[ (830 bytes)
  0x00007f1ddd0008e4: push   %rbp
  0x00007f1ddd0008e5: mov    %rsp,%rbp
  0x00007f1ddd0008e8: sub    $0x60,%rsp
  ...
```

call_stub Check code contents

I will roughly grasp the contents of call_stub.

1. Save rbp with enter ()

First, `push rbp`from the `mov rsp, rbp`sets the value of rsp in to save the rbp.



Save rbp with enter ()

Here, thread and parameter size appear, which are the 7th and 8th arguments passed when calling call_stub. I will post the relevant code in javaCalls again. I don't understand CHECK properly, but it seems to be passing thread here. The calling convention here is assumed to be the System V AMD64 ABI environment referred to in x86 calling conventions .

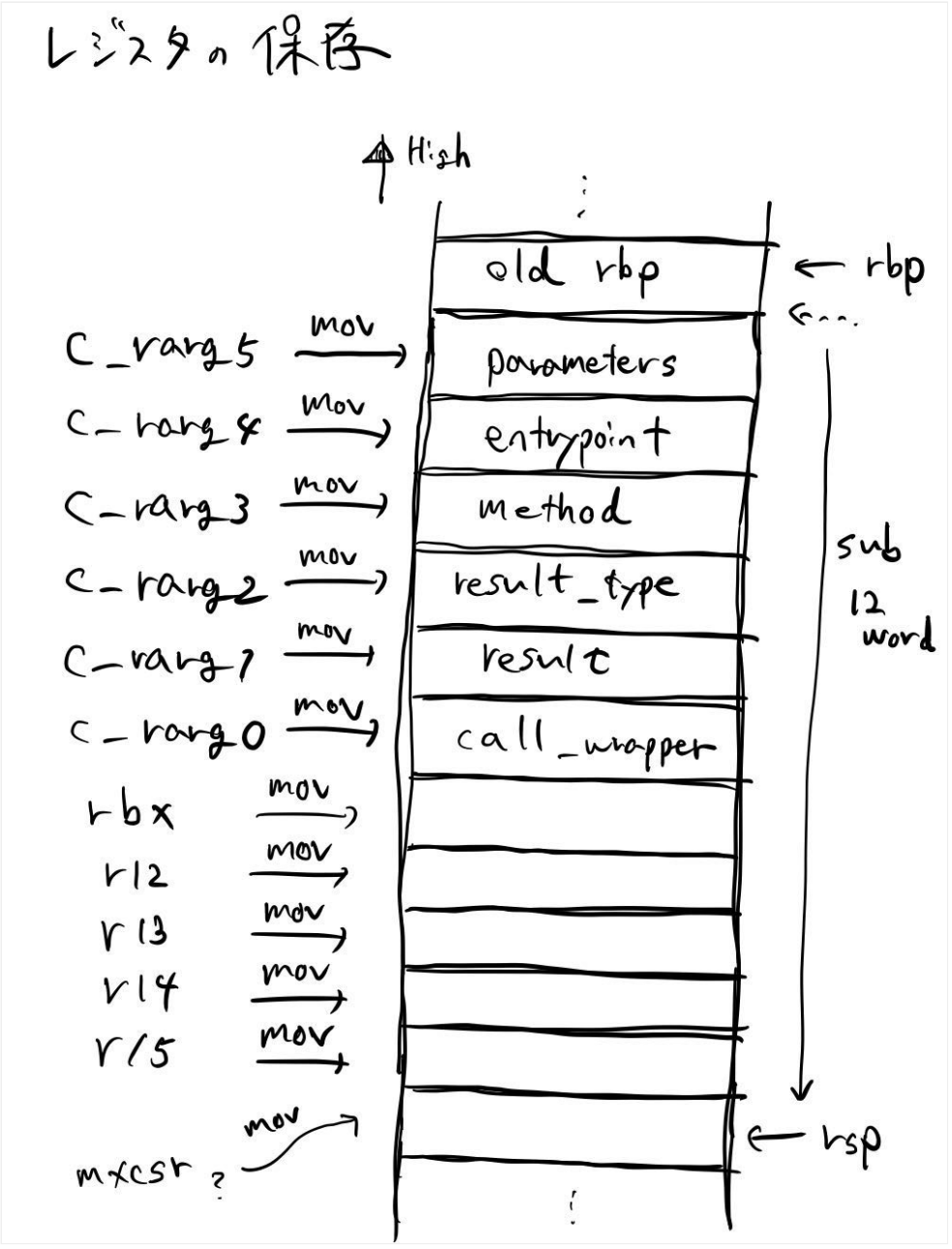
```
// in src/hotspot/share/runtime/javaCalls.cpp

void JavaCalls::call_helper(JavaValue* result, const methodHandle& method, JavaCallArguments* args, TRAPS) {
      ...
      StubRoutines::call_stub()(
```

```
        (address)&link,
        // (intptr_t*)&(result->_value), // see NOTE above (compiler problem)
        result_val_address,          // see NOTE above (compiler problem)
        result_type,
        method(),
        entry_point,
        args->parameters(),
        args->size_of_parameters(),
        CHECK
      );
      ...
}
```

2. Save registers

The registers are kept on the stack. In the figure, c_rarg0, 1, 2, 3, 4, and 5 are rdi, rsi, rdx, rcx, r8, and r9, respectively, in the current calling convention.

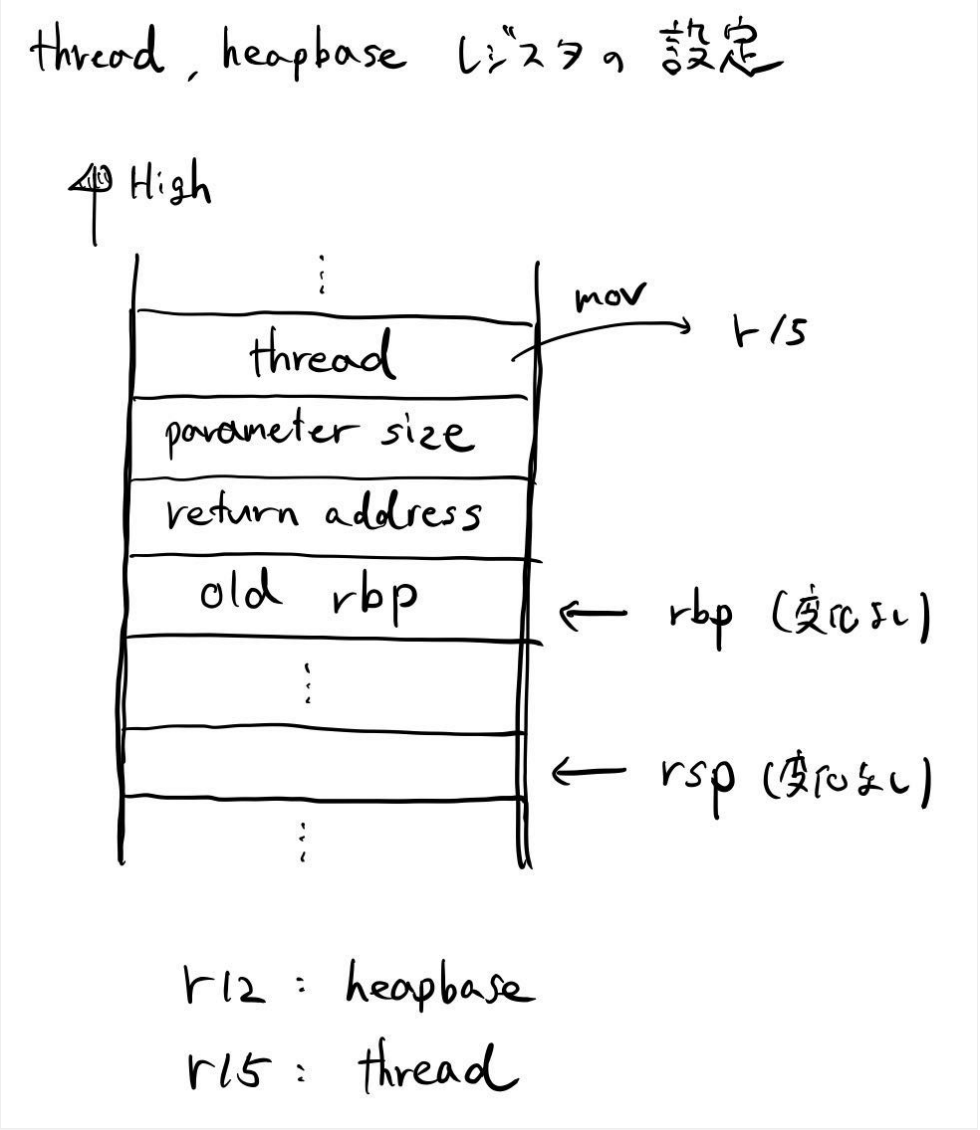


Saving registers

I also save the mxcsr? Register, but I don't know much about it.

3. Register setting for thread and heapbase

It seems that there are some registers in the interpreter, of which the r12 and r15 registers are used to point to heapbase and thread, respectively.



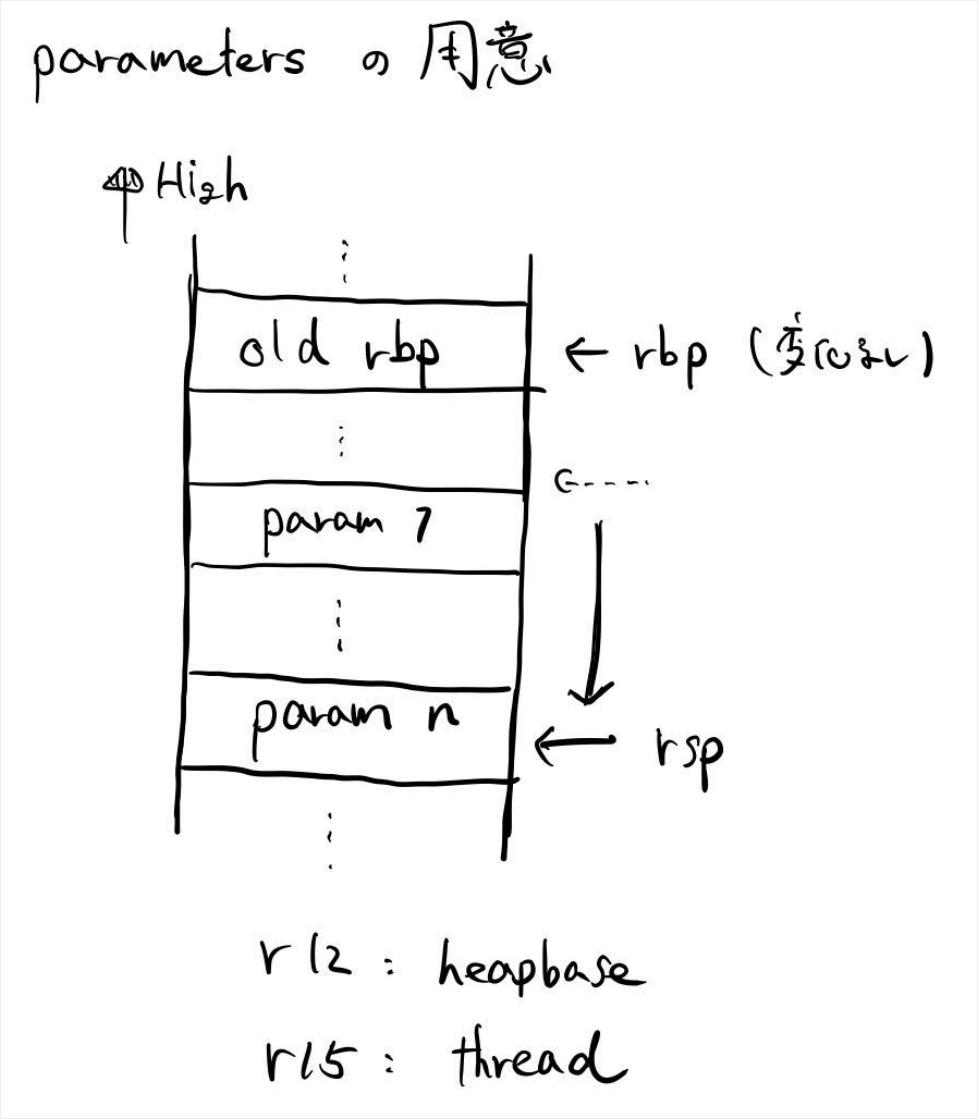
Setting of registers for thread and heapbase

heapbase is the base address used in the calculation of compressed oop. I wrote about compressed oop in a previous post . Here, the base address of oop is set, but according to my memo, it is sometimes set to the base address for Klass during interpret.

We have not investigated exactly what thread refers to.

4. Preparation of parameters

Set the method parameters you want to interpret here. The parameters and their numbers are provided in the caller of call_stub, so use them.



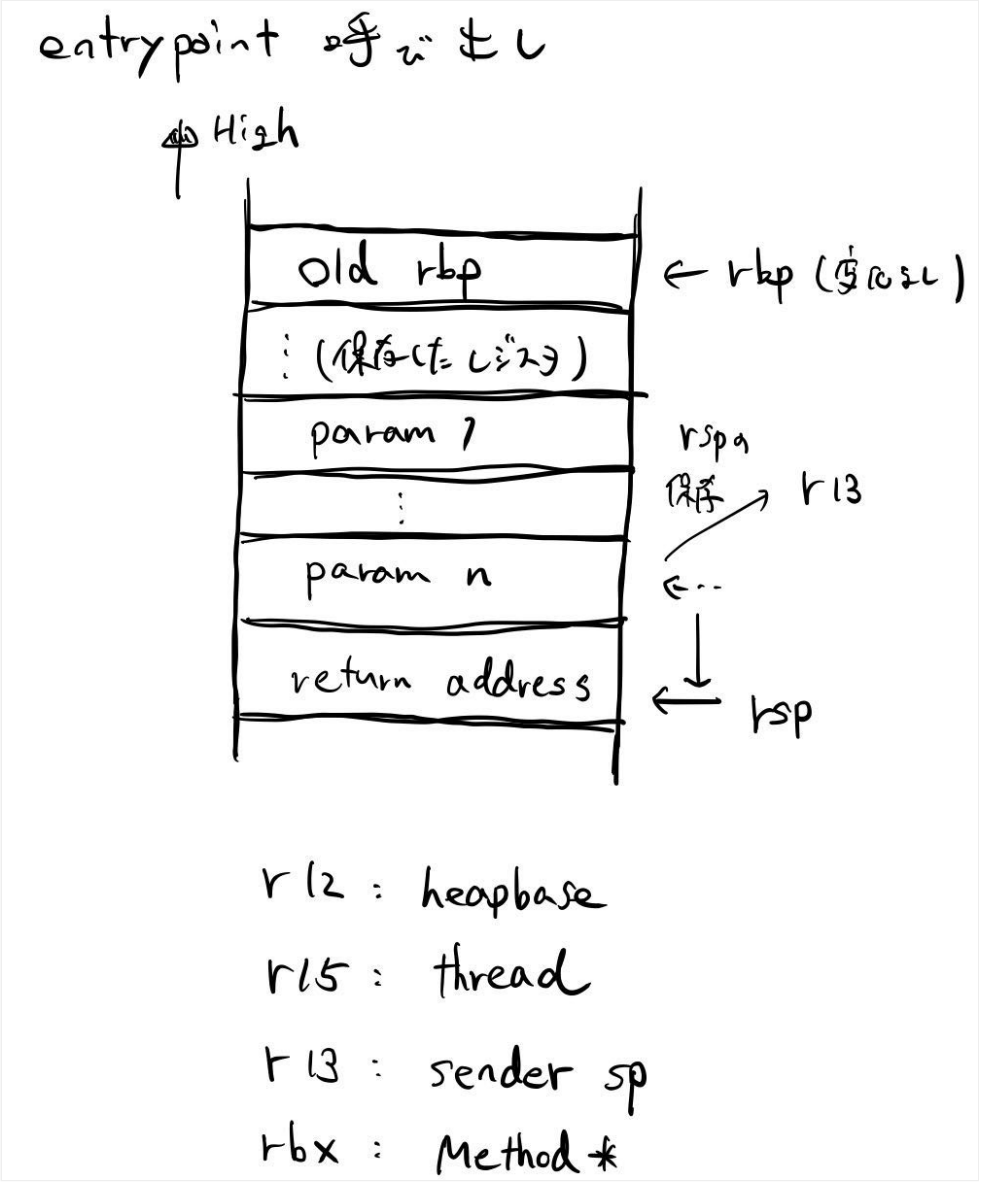
Prepare parameters

5. Call entrypoint

Finally, a jmp from call_stub to normal_entry. In call_stub, the address is passed from caller as entry_point, so use that.

Here, sender sp, `Method *`is set in r13 and rbx registers respectively .

This provides the stack and registers as expected by the normal_entry seen above.



Call entrypoint

Finally, using the Locals class, look at the state of the stack and registers just before moving to the entrypoint with gdb. Because it is before the call, unlike the figure, the return address is not yet on the stack.

```
# breakpoint before jmp to entrypoint
(gdb) p $rbp
$1 = (void *) 0x7ffff59ed960
(gdb) p $rsp
$2 = (void *) 0x7ffff59ed8f8
(gdb) p ($rbp - $rsp) / 8
$3 = 13
(gdb) x /120xb $rsp
0x7ffff59ed8f8: 0xb8    0x56    0x6f    0x19    0x07    0x00    0x00    0x00 # param1 <- rsp
0x7ffff59ed900: 0xa0    0x1f    0x00    0x00    0xff    0x7f    0x00    0x00 # mxcsr?
0x7ffff59ed908: 0x40    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00 # saved r15
0x7ffff59ed910: 0x00    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00 # saved r14
0x7ffff59ed918: 0x40    0xdb    0x9e    0xf5    0xff    0x7f    0x00    0x00 # saved r13
0x7ffff59ed920: 0x00    0xa8    0x01    0xf0    0xff    0x7f    0x00    0x00 # saved r12
0x7ffff59ed928: 0xe0    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00 # saved rbx
0x7ffff59ed930: 0x40    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00 # call_wrapper (from c_rarg0)
0x7ffff59ed938: 0xe8    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00 # result       (from c_rarg1)
0x7ffff59ed940: 0x0a    0x00    0x00    0x00    0xff    0x7f    0x00    0x00 # result_type  (from c_rarg2)
0x7ffff59ed948: 0x88    0x53    0x3a    0xd1    0xff    0x7f    0x00    0x00 # method       (from c_rarg3)
0x7ffff59ed950: 0x40    0x61    0x01    0xe1    0xff    0x7f    0x00    0x00 # entrypoint   (from c_rarg4)
0x7ffff59ed958: 0x10    0xdc    0x9e    0xf5    0xff    0x7f    0x00    0x00 # parameters   (from c_rarg5)
0x7ffff59ed960: 0xc0    0xda    0x9e    0xf5    0xff    0x7f    0x00    0x00 # <- rbp
```

```
0x7ffff59ed968: 0xaa    0xa6    0xa8    0xf6    0xff    0x7f    0x00    0x00

# check param
(gdb) x /40xb 0x07196f56b8
0x7196f56b8:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7196f56c0:    0xa0    0x5c    0x02    0x00    0x02    0x00    0x00    0x00
0x7196f56c8:    0xdd    0xea    0x2d    0xe3    0xe6    0xea    0x2d    0xe3
0x7196f56d0:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7196f56d8:    0x40    0x08    0x00    0x00    0x03    0x00    0x00    0x00

# check registers
(gdb) p /x $r12
$4 = 0x0

(gdb) p /x $r15
$5 = 0x7ffff001a800

(gdb) p /x $r13
$6 = 0x7ffff59ed8f8

(gdb) p /x $rbp
$10 = (void *) 0x7ffff59ed960
(gdb) p ((Method *) $rbx)->_constMethod->_constants->_pool_holder->_name->as_C_string()
$9 = 0x7ffff00193d0 "Locals"
```

I really want to summarize the processing of the return part, but I was exhausted.