KnownNonBugIssuesAndLimitations

1. Stack tracing/dumping is half-dead.
Post-mortem stack trace is printed on Linux@ia32 only; thread dump (as printed out on "Ctrl-\" or "Ctrl-Break") lacks thread info (e.g. name, group, monitors held, etc) and skips native threads; some (inlined) methods may be missing in reported exception stacks. In general, this area has a lot of broken or half-made code and needs big cleanup & refactoring.

2. When OOME is thrown it has no stack trace.
OOME object is pre-allocated and contains no stack trace. When OOME condition is detected, the pre-allocated object is used, and no stack trace elements are created. So, OOME always has a null stack trace.

3. JVMTI API functions should not call Java code.
If JVMTI functions call some Java code directly, like it is done currently in GetThreadInfo function, or indirectly, like using JNI FindClass, it may easily lead to infinite recursion. Calling Java code may result in some JVMTI events, which call agent callbacks, and these callbacks may use the same JVMTI API functions again. A workaround to this problem may be to disable any JVMTI events when calling Java code. A better solution would be not to call any Java code from JVMTI functions.

4. JNI function EnsureCapacity always returns success.
This function according to the specification ensures that at least some numbers of local references can be allocated. But in fact this function doesn't do anything, it always returns success. It looks like this function is actually obsolete and RI allows allocating any number of local references regardless of whether their number was ensured or not.

5. JVMTI functions don't check for invalid jmethodID arguments.
According to the specification, functions accepting such arguments should return `JVMTI_ERROR_INVALID_METHODID`. But there is no easy method to check these identifiers for validness since in DRLVM these are just heap pointers to `Method` type structs. To check these values correctly, keep a cache of all loaded methods.

6. JVMTI functions which get/set local variables don't check for variable type.
Functions like GetObjectField should return `JVMTI_ERROR_TYPE_MISMATCH` if they are called for a local variable, which is not a type object. To implement it, create type maps for all methods, either verifier or JIT can do this. Note, that this requires some interface for JVMTI.

7. JVMTI capability can_tag_objects can only be requested at OnLoad phase, and by at most one environment.

This is because tagging is implemented by storing tag pointer in object directly.

8. Get rid of duplicated simple types definitions where applicable

The content of open/types.h should be freed from types duplicated in JNI or JVMTI. VM sources should preferably be cleared from non C, non JNI, or non JVMTI types where applicable.

9. Make native symbols lookup more optimal

Currently the native symbols are resolved in reverse order for registered libraries (i.e. newly loaded lib is prepended to the list). So vmcore.dll is always requested in the last place. Besides, on Windows OS-specific mangling is tried as a last resort. Therefore, most native functions during startup are found in a least effective way.

?KNOWN ISSUES?

Please look through the issues mentioned below to determine their validity.

If the issue is not valid, please remove it.

If the issue is valid, please mark it +1.

After this check we can add the up-to-date issues to the README file to save important info.

Building in a self-hosting environment:

1. In certain cases, the following error might be produced:

```
java.lang.InternalError: Error -1 getting next zip entry
```

The error might appear during compilation of the kernel classes in a Harmony self-hosting environment due to the problems with the zip support module. A simple workaround is to restart the build or build the kernel classes separately by using the following command:

```
build.bat -DCOMPONENTS=vm.kernel_classes
```

General restrictions:

1. The number of threads must be lower than 800.

2. Java heap memory must not be greater than 1.3 GB.

3. Code must not enter the same monitor more than 256 times.

4. The real object hash code has only 6 bits.

Partially implemented features:

1. JVMTI implementation does not provide full argument validity checks. Specifically, the VM might crash when passing invalid Java objects or JNI identifiers as arguments to JVMTI API functions.

2. JVMTI API function groups for local variable access and stack inspection in the JIT-enabled mode have not been tested for adequate operation.

3. Tracing of method calls and instructions is not supported.

4. The following methods have no effect:

```
Runtime.traceInstructions(bool)
```

```
Runtime.traceMethodCalls(bool)
```

5. The `java.lang.ref.SoftReference` class implementation does not follow the intent of the specification and is currently equivalent to the `java.lang.ref.WeakReference` class.

Jitrino restrictions:

1. Parallel compilation and calls for resolution during a compilation session may lead to re-entering Jitrino from different or the same hread. In our experience, Jitrino works normally in such cases. However, certain Jitrino parts, such as timers and certain logging system features are not designed to be re-enterable. This restriction is not applicable to normal runs in the default mode with logging and timers disabled.