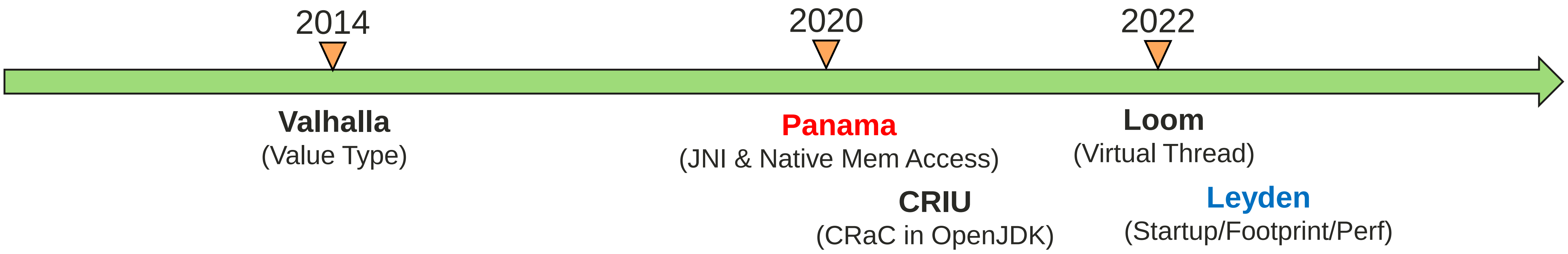
Project Panama on OpenJ9

Department name: Compiler Lab (Hong Kong Research Center)
Author’s name: Cheng Jin
Date: 2025/04/29

Security Level:

Focusing on JEPs in OpenJ9



2014

2020

2022

Valhalla
(Value Type)

Panama
(JNI & Native Mem Access)

Loom
(Virtual Thread)

CRIU
(CRaC in OpenJDK)

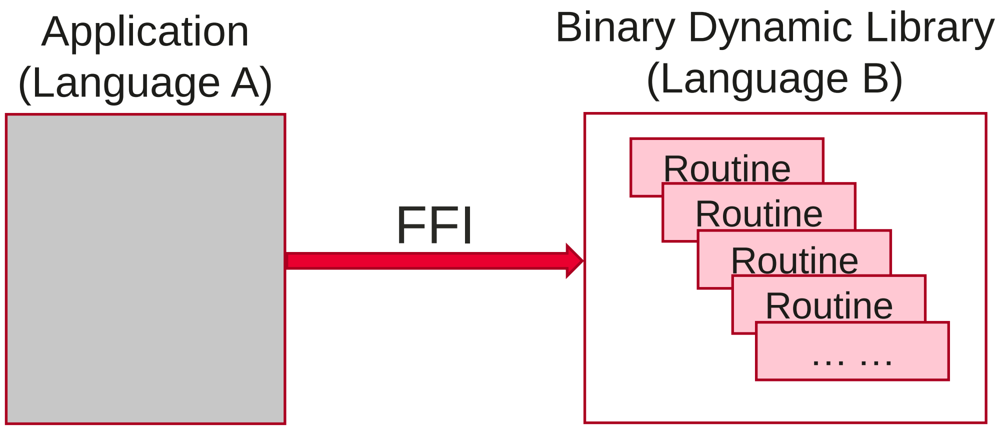
Leyden
(Startup/Footprint/Perf)

- Java roadmap
- Architectural changes in VM/Interpreter (Bytecode)
- Intention of JEPs in GC & JIT

https://openjdk.org/jeps/0

What is FFI ?

“*A mechanism by which a program written in one programming language can call routines or make use of services written or compiled in another one.*” –Wikipedia



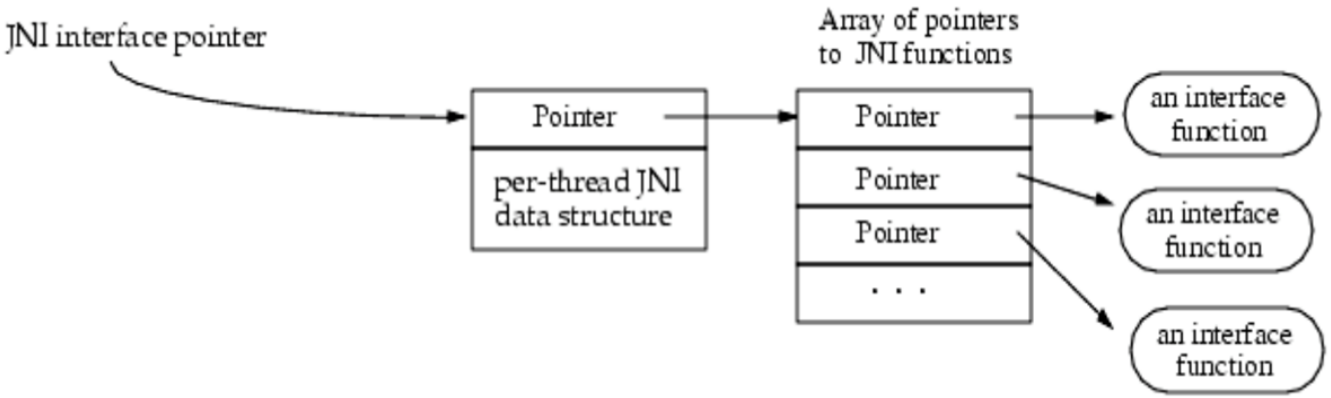
Interop with the native libraries (C/C++) in various areas:

- Scientific/Mathematical calculations
- GPU/Graphics/DB specific operations
- AI/ML/DL(Caffe, Tensorflow, etc)
- new fancy libraries

FFI in Java

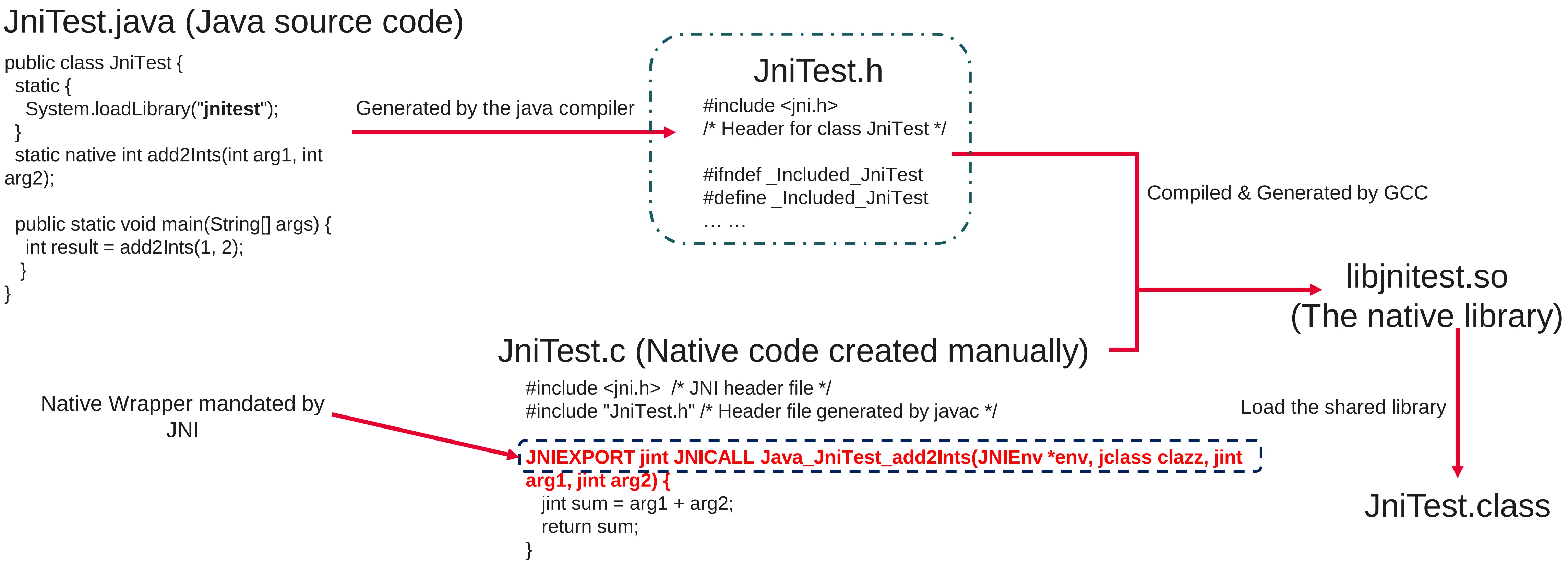
Java Native Interface (JNI)

- The 1st generation of the FFI framework in JDK
- Interact with native libraries (C/C++) from Java
- Callback into JVM from native functions
  - Invoke java method
  - Access on-heap data



An Example of JNI

Invocation from Java to Native (Linux/x86_64)



JniTest.java (Java source code)

```
public class JniTest {
  static {
    System.loadLibrary("jnitest");
  }
  static native int add2Ints(int arg1, int
arg2);

  public static void main(String[] args) {
    int result = add2Ints(1, 2);
   }
}
```

Generated by the java compiler

JniTest.h

```
#include <jni.h>
/* Header for class JniTest */

#ifndef _Included_JniTest
#define _Included_JniTest
… …
```

Compiled & Generated by GCC

libjnitest.so
(The native library)

Load the shared library

JniTest.class

JniTest.c (Native code created manually)

Native Wrapper mandated by JNI

```
#include <jni.h>  /* JNI header file */
#include "JniTest.h" /* Header file generated by javac */

JNIEXPORT jint JNICALL Java_JniTest_add2Ints(JNIEnv *env, jclass clazz, jint
arg1, jint arg2) {
   jint sum = arg1 + arg2;
   return sum;
}
```

Drawbacks of JNI

- Complicated/fragile in writing the native wrapper code
  - C/C++ knowledges: Developers need to write & compile the native code
  - Tons of Code: Native wrapper is mandatory for every native function (burdensome for maintenance)
- Tricky to exchange the aggregate data between java (object) and native structure
  - Native types that don’t match Java types: need to be split into primitives or leverage Unsafe/ByteBuffers (which leads to bugs due to the missing type information)
  - On/Off-heap marshalling: keeping object in place is required for GC to interact with the underlying structure in native
- Slow transition from Java to native via the extra indirection with wrapper
  - JVM callbacks are often required
  - Lack of optimization from the JIT perspective
- Partially improved by Java Native Access (JNA/reflection) & Java Native Runtime (JNR/generated code)
  - Implemented on top of JNI via the external facilities
  - no longer active in the community
  - Lack of sustainable support/enhancement

https://github.com/java-native-access/jna
https://github.com/jnr

How the Native Memory Access works in Java?

- **java.nio.ByteBuffer**
  - Creates direct object & off-heap byte buffers
  - 2GB limitation in size due to an int-based indexing scheme
  - Deallocation done by the garbage collector finalization (non-deterministic)
- **sun.misc.Unsafe**
  - Allows direct off-heap access to native memory
  - Extremely efficient memory access supported by JVM intrinsics & optimization via JIT
  - Error-prone programming model (JVM crashes due to illegal memory access)
  - Non-standard/Restricted Java API

What is Project Panama ?

A project that enhances connections between the JVM and interfaces used by C/C++ programmers.

Goals

- **Productivity** — Replacement of JNI with pure-Java APIs (concise/readability)
- **Performance** — Overhead of access to foreign functions & memory comparable to JNI/Unsafe
- **Broad platform support** — Enablement of native libraries' invocation on any JDK-installed platform
- **Uniformity** — Manipulation of primitive/structured data in unlimited size (native/on-heap memory)
- **Soundness** — Enhanced memory management mechanism across multiple threads (no use-after-free bugs)
- **Integrity** — Unsafe operations with native code/data with warnings by default

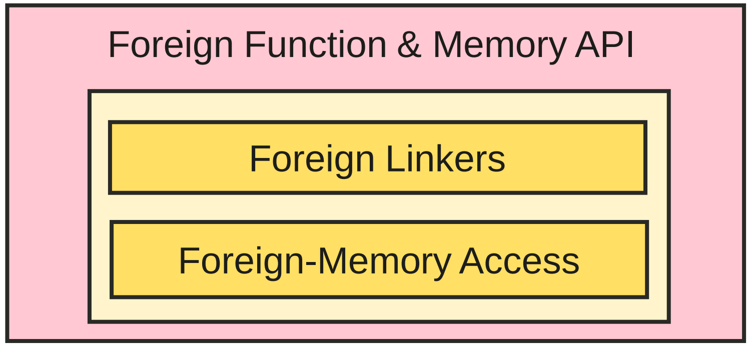
Pillars of the FFM Framework

Foreign Function & Memory API (FFM)

- Foreign-Memory Access API (FMA)
  - *Safe & efficient on/off-heap memory access & management*
- Foreign Linker API (FLA)
  - *Native invocation (downcall) & Callback (upcall)*
- Foreign-Jextract (tool)
  - *Automatic generation of the FLA binding code*

- **Arena/Scope** – native memory management (supported by the try-with-resources block)
- **Segment** – contiguous memory regions with spatial & temporal bounds (controlled by Arena)
- **Layout** – the signature representation for primitive/structures
- **MH-based Memory Access** – MH combinator & var handle

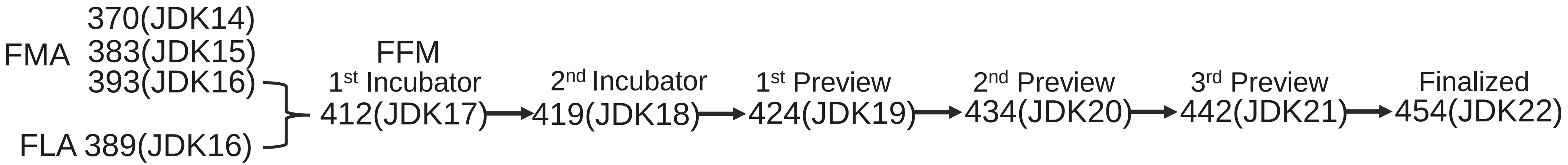
- **Linker** – Downcalls/Upcall & LinkerOptions
- **SymbolLookup** – loader/library/defaultLookup for zero-sized symbol segment (controlled by Arena)
- **FunctionDescriptor** – arrangement of layouts for return type & arguments



Foreign Function & Memory API (FFM)

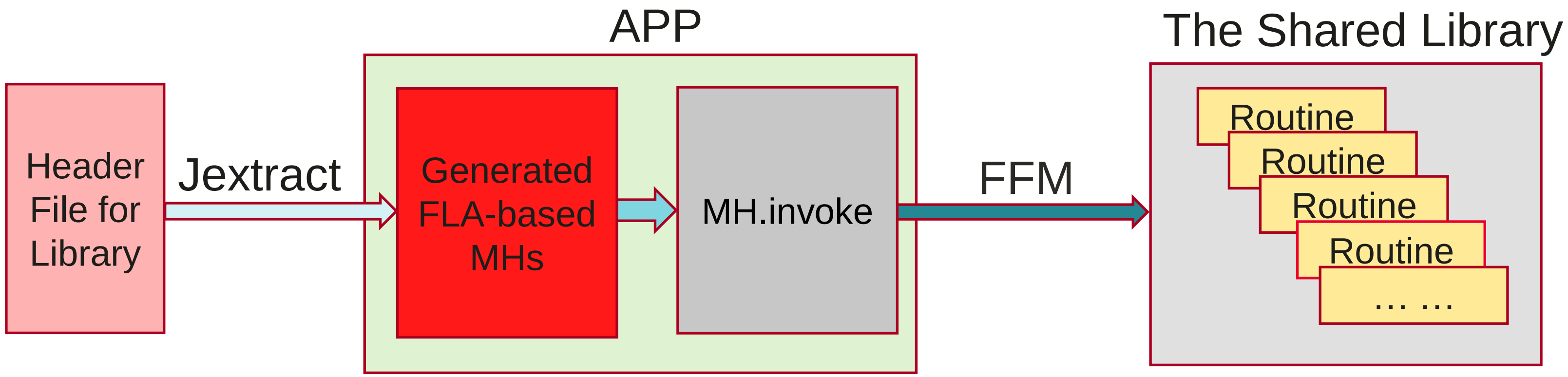
- The 2nd generation of the FFI framework in JDK
- Pure Java APIs for access to the native functions/data (easiness/flexibility/performance)
- Initially introduced in JEP389/JDK16 to support Foreign Linkers
- Replacement of JNI/Unsafe

Evolution of JEPs

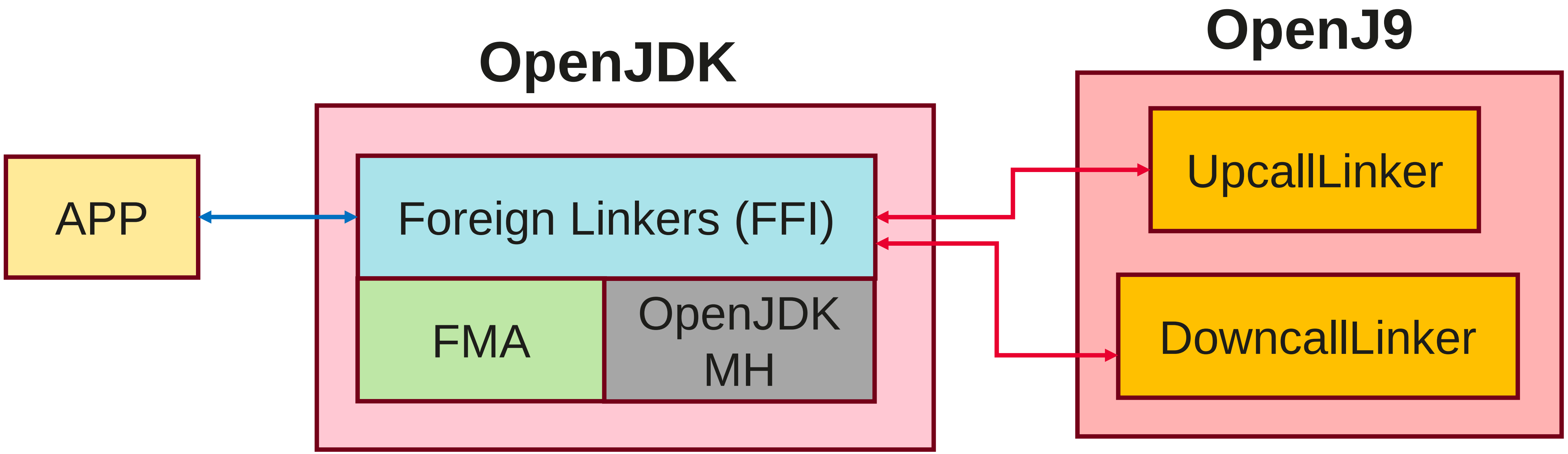


Benefits of FFM

- Easy to write/maintain (with the support of jextract)
- No rules/protocols required for the native code

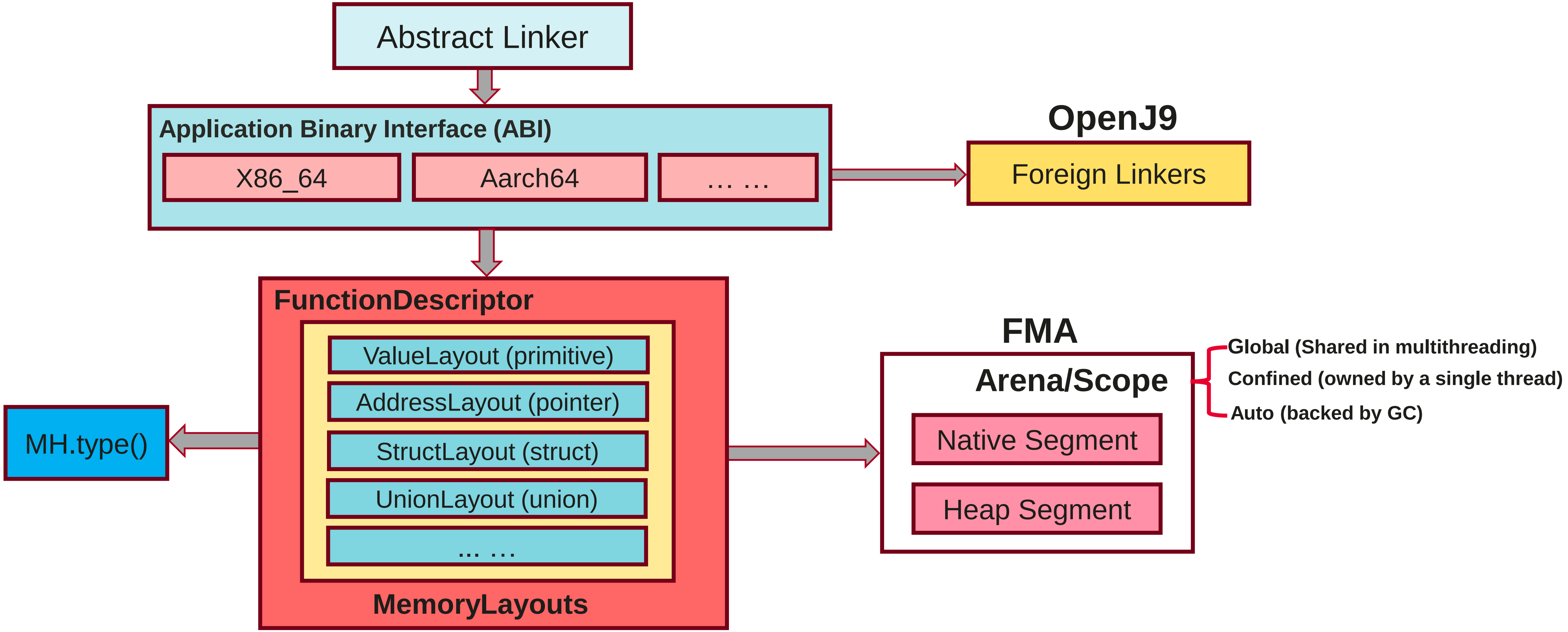


The FFM Framework



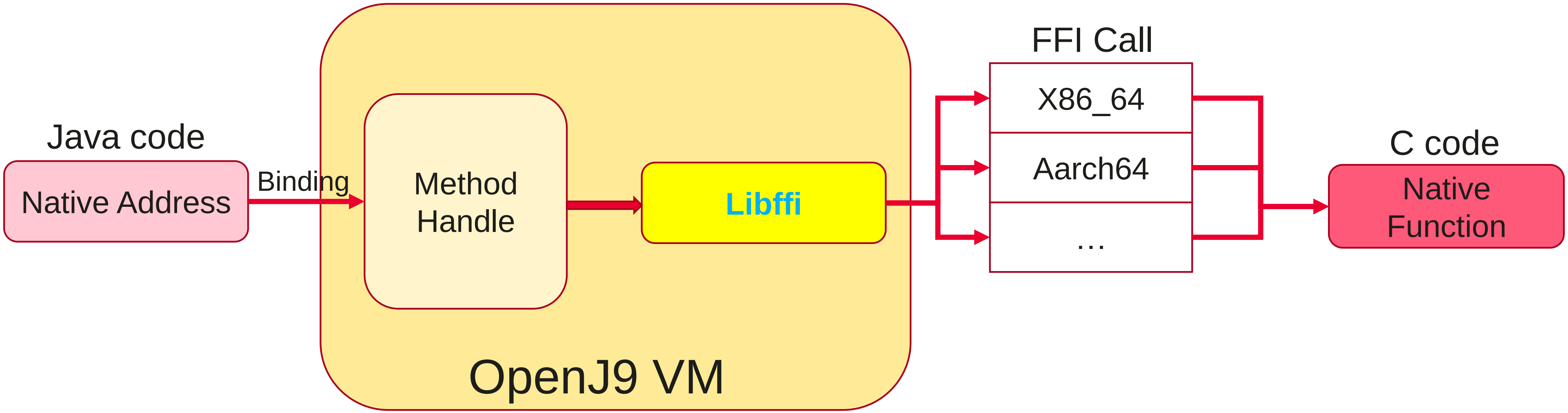
- ❑ Downcall & Upcall are performed via the Foreign Linker APIs in applications
- ❑ Everything with FFI is built on top of OpenJDK MethodHandle
- ❑ Access to the native/heap memory is mostly achieved by FMA
- ❑ The underlying linkers for Downcall & Upcall are implemented within OpenJ9

The FFM Framework (Cont.)



- A new architecture is enabled by adding its own **ABI** specifics intended for Downcall & Upcall
- **FunctionDescriptor** holds the layouts of arguments & return value (MemorySegment is mandatory for pointer/struct/union)
- MethodType of Downcall & Upcall MH is deduced from FunctionDescriptor in OpenJDK
- **Arena** takes responsible for the memory management in terms of the spatial & temporal bounds (Global/Confined/Auto)

Downcall Linker in OpenJ9



- ○ Libffi (the open-sourced FFI native library) supports multiple platforms (initially adopted by OpenJ9 JNI)
- ○ Downcall is achieved by encapsulating the libffi preparation/invocation with OpenJDK MH
  - ▪ The layout data are extracted from FunctionDescriptor to prepare libffi data for the primitive/complex structures
  - ▪ The MethodHandle bound with the native address literally invokes the native function via the libffi interface

https://github.com/libffi/libffi

Downcall Linker in OpenJ9 (Cont.)

1. Ffi_data preparation for native

Native address

```
MemorySegment symbol =
SymbolLookup.libraryLookup(
lib_path, arena)
.find(native_name).get();
```

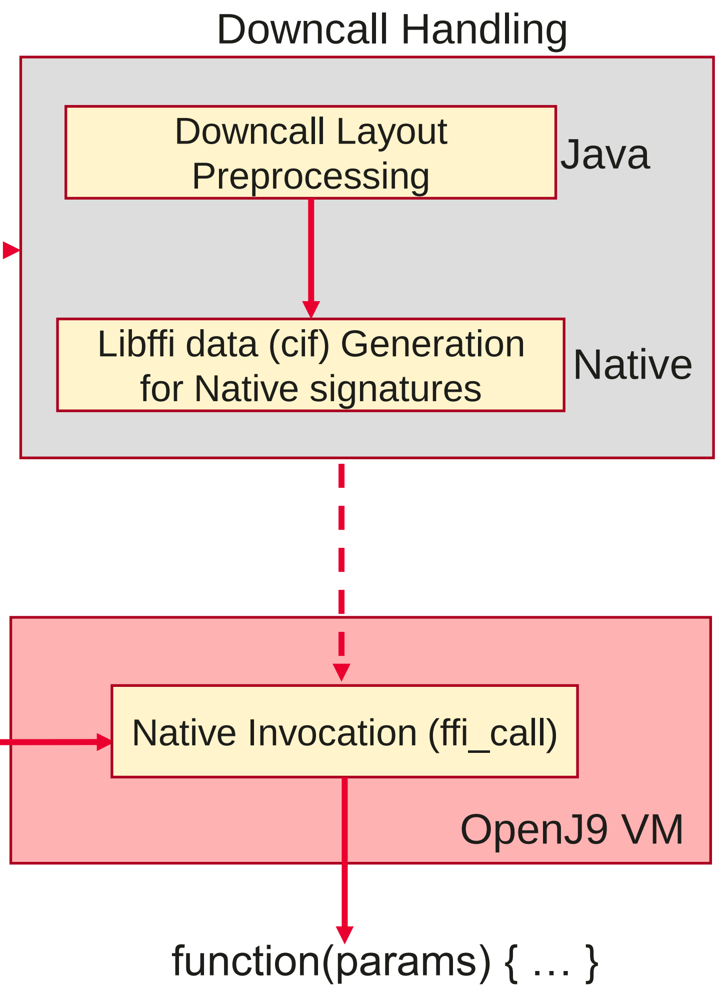
```
MethodHandle mh =
Linker.nativeLinker().downcallHandle
(
symbol,
nativeDesc);
```

```
FunctionDescriptor nativeDesc =
FunctionDescriptor.of(return_layo
ut, argument_layouts);
```

2. Native invocation via libffi

```
mh.invokeExact(arguments
);
```

OpenJDK MH
(bound with native)


Downcall Handling
Downcall Layout Preprocessing
Java
Libffi data (cif) Generation for Native signatures
Native
Native Invocation (ffi_call)
OpenJ9 VM
function(params) { … }

- Libffi data (cif) for signatures is prepared & stored in java
- Native invocation passes the libffi data to ffi_call() in native for downcall

Workflow in Downcall

1. Ffi_data preparation for native

Native address

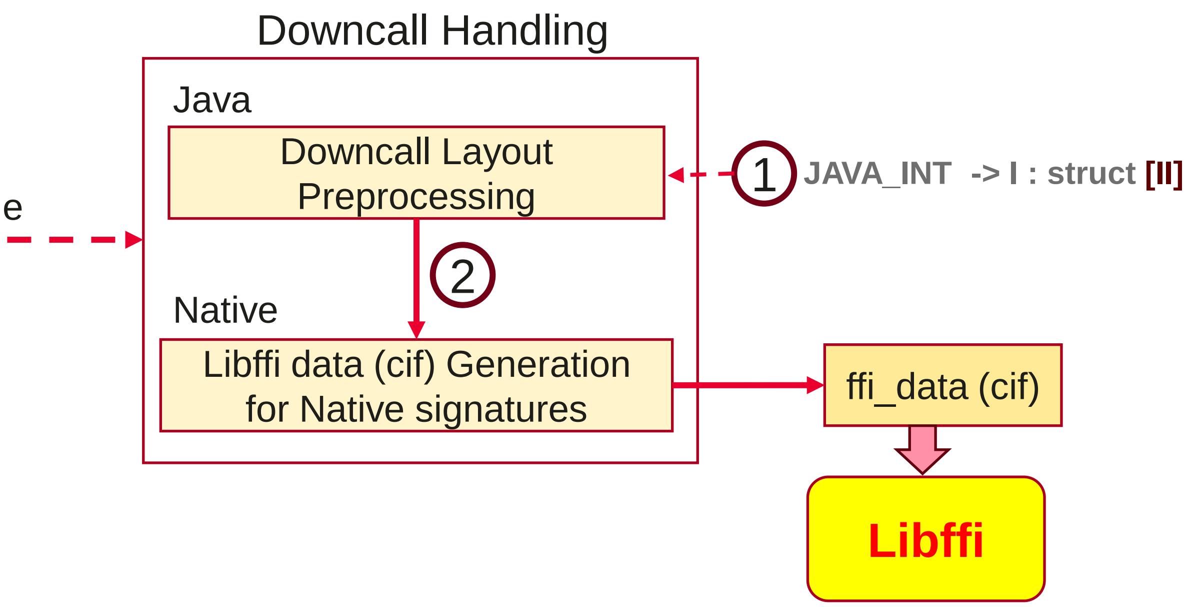
```
MemorySegment symbol
=
SymbolLookup.libraryLook
up(
lib_path, arena)
.find(add2Structs).get();
```

```
MethodHandle mh =
Linker.nativeLinker().downcallHandle
(
symbol,
nativeDesc);
```

```
FunctionDescriptor nativeDesc =
FunctionDescriptor.of(structLayout,
structLayout, structLayout);
```

```
 StructLayout structLayout =
 MemoryLayout.structLayout(
JAVA_INT.withName("elem1"),
JAVA_INT.withName("elem2")
);
```

```
typedef struct stru_2Ints
{
  int elem1;
  int elem2;
} stru_2Ints;
```

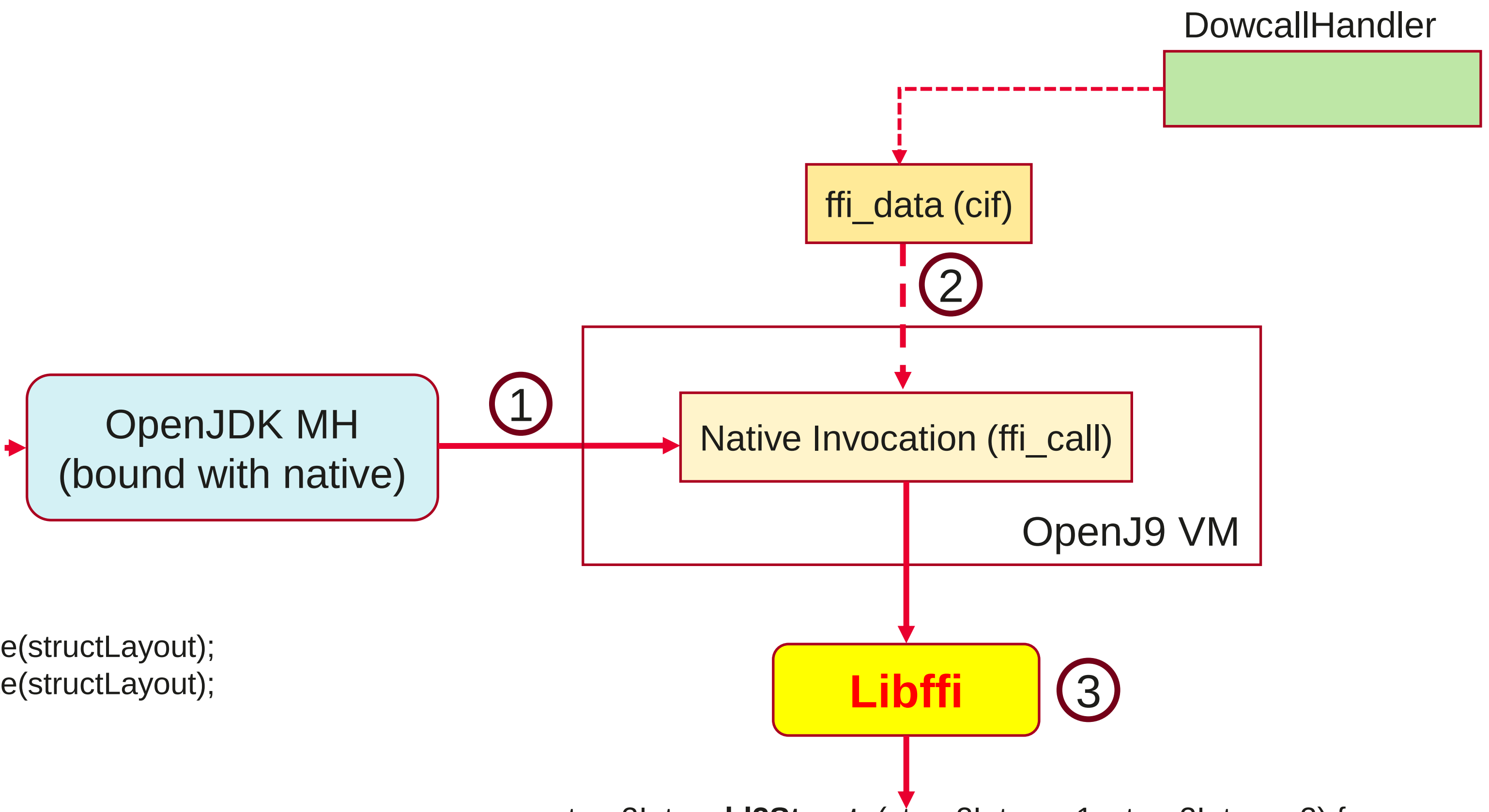

Downcall Handling
Java
Downcall Layout Preprocessing
1
JAVA_INT -> I : struct [II]
2
Native
Libffi data (cif) Generation for Native signatures
ffi_data (cif)
Libffi

1 Convert the layout to a simplified signature

2 Set ffi_data (cif) against the native signatures

Workflow in Downcall (Cont.)

2. Native invocation via libffi



```
mh.invoke(arena,
structSegmt1,
structSegmt2)
```

```
MemorySegment structSegmt1 = arena.allocate(structLayout);
MemorySegment structSegmt2 = arena.allocate(structLayout);
```

```
stru_2Ints add2Structs(stru_2Ints arg1, stru_2Ints arg2) {
    stru_2Ints sum;
    sum.elem1 = arg1.elem1 + arg2.elem1;
    sum.elem2 = arg1.elem2 + arg2.elem2;
     return sum ;
}
```

① Invoke the native method (downcall MH) in JVM

② Set the arguments from ffi_data (java) for libffi

③ Call the native function via ffi_call in libffi

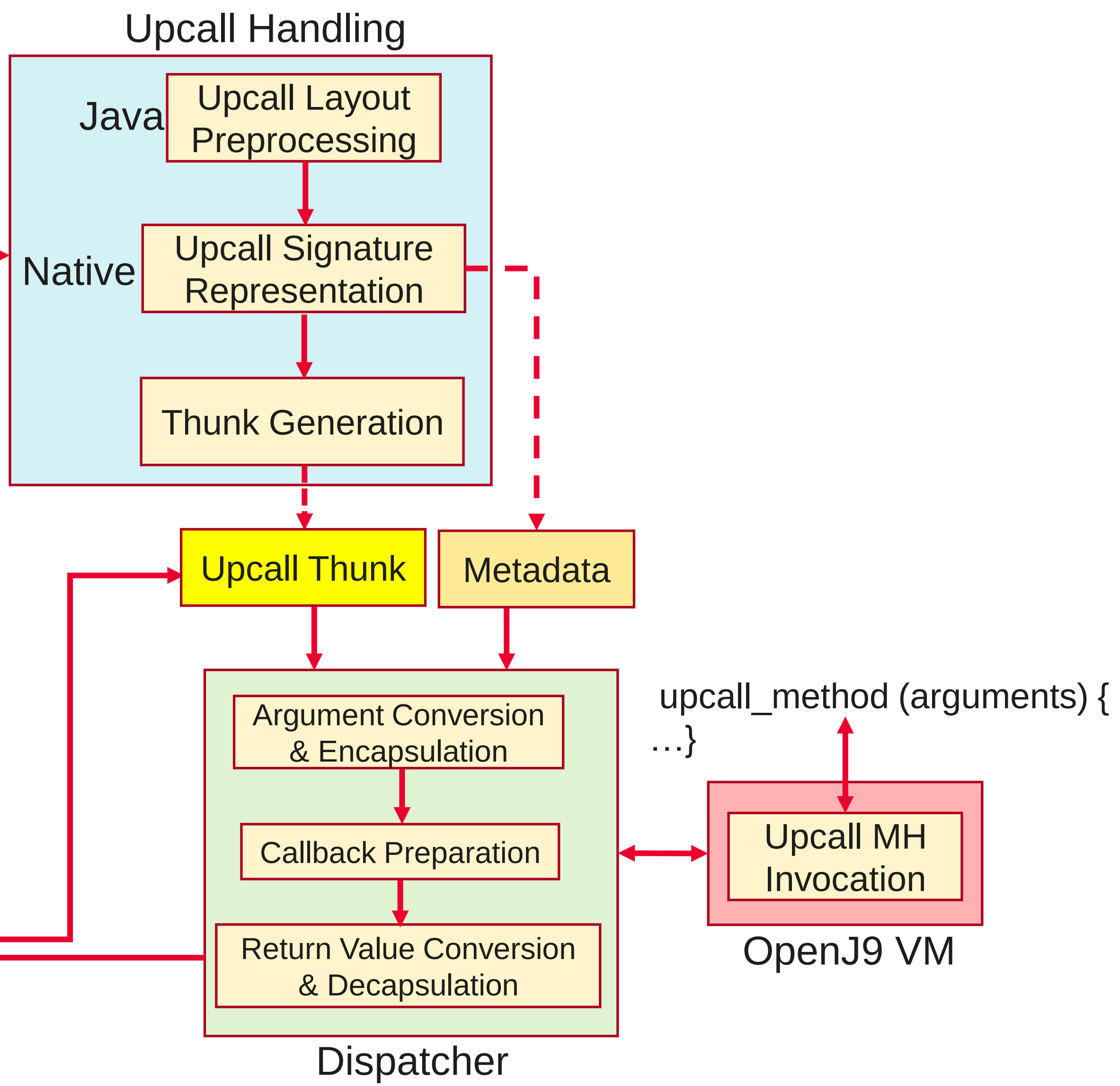
Upcall Linker in OpenJ9

1. Metadata & Thunk Generation

MethodHandle **upcall_mh** = MethodHandles.lookup().findStatic(…,upcall_methodName, upcall_methodType);

FunctionDescriptor **upcall_fd** = FunctionDescriptor.of(…);

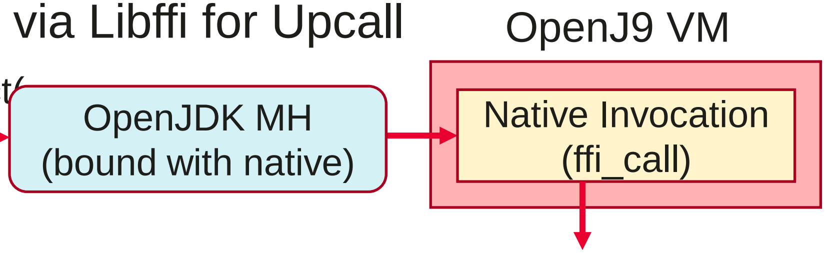
MemorySegment **upcallStubPtr** = linker.upcallStub(**upcall_mh**, **upcall_fd**, arena);



Upcall Handling
Java
Upcall Layout Preprocessing
Native
Upcall Signature Representation
Thunk Generation
Upcall Thunk
Metadata
Argument Conversion & Encapsulation
Callback Preparation
Return Value Conversion & Decapsulation
Dispatcher
upcall_method (arguments) { …}
Upcall MH Invocation
OpenJ9 VM

2. Native Invocation via Libffi for Upcall

downcall_mh.invokeExact(arguments, **upcallStubPtr**);



```
function(params, (* upcallStubPtr)(…)) {
  return (*upcallStubPtr)(…);
}
```

- triggered from the same thread as in downcall
- triggered by a new thread created in native

Functionalities of Dispatcher

- Encapsulate raw value for object arguments (pointer/struct/union)
- Extract raw value from the returned object (pointer/struct/union)
- Convert arguments/return value required on given platforms

Workflow in Upcall

1. Metadata & Thunk Generation

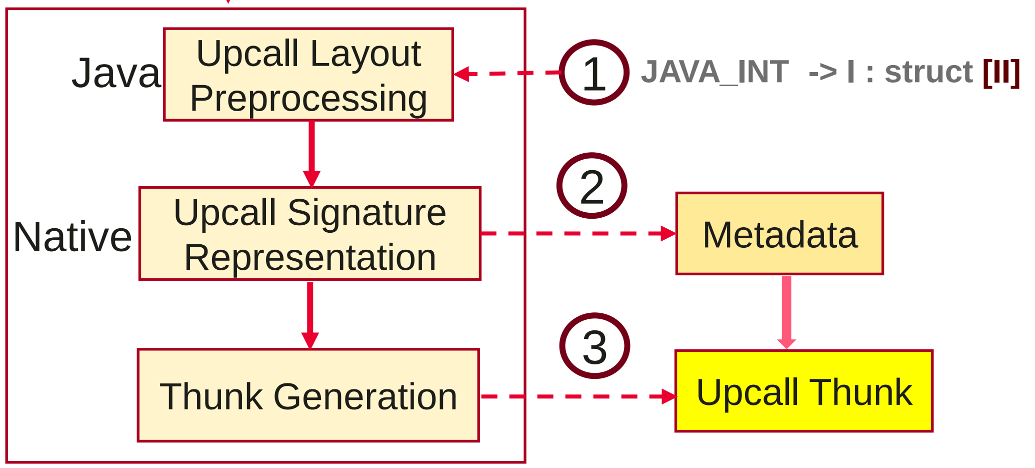
```
typedef struct stru_2Ints
{
  int elem1;
  int elem2;
} stru_2Ints;
```

```
MemorySegment
add2StructsByUpcall(MemorySegment arg1,
MemorySegment arg2) {…}
MethodHandle upcall_mh =
MethodHandles.lookup().findStatic(...,
"add2StructsByUpcall",
methodType(MemorySegment.class,
MemorySegment.class, MemorySegment.class));

StructLayout structLayout =
MemoryLayout.structLayout(JAVA_INT.withName(
"elem1"), JAVA_INT.withName("elem2"));
 FunctionDescriptor upcall_fd =
FunctionDescriptor.of(structLayout,
structLayout, structLayout);
```

```
MemorySegment upcallStubPtr = linker.upcallStub(
upcall_mh,
upcall_fd,
arena);
```

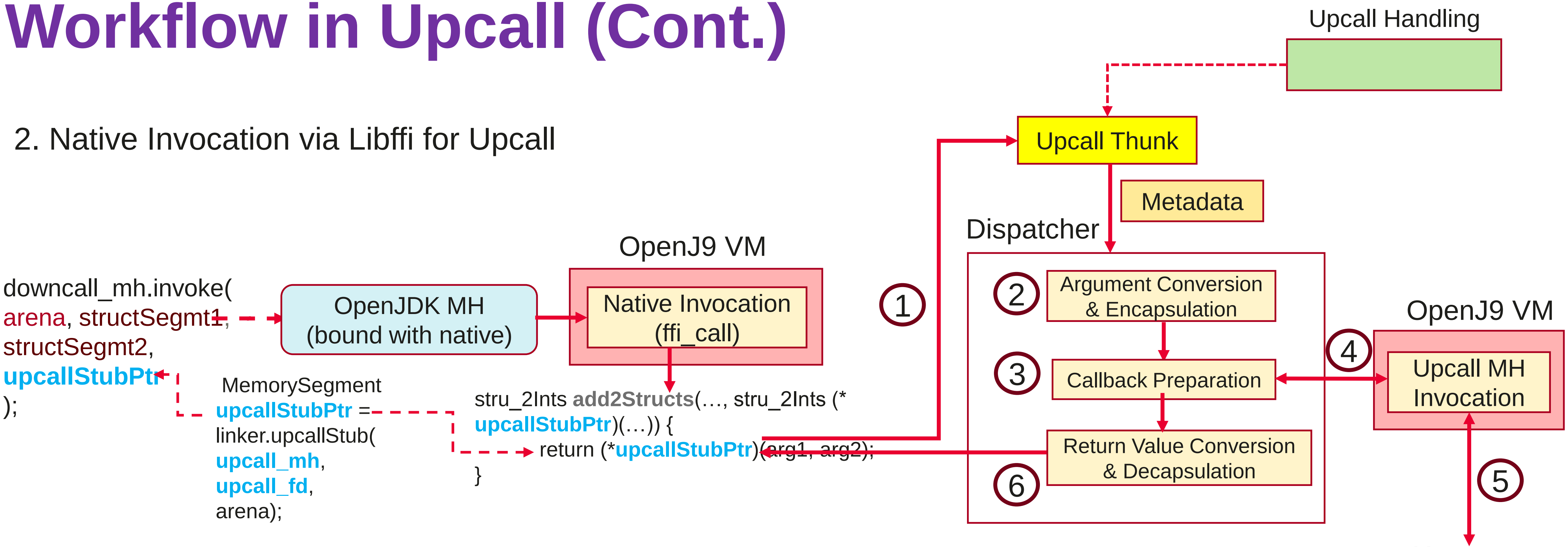
Upcall Handling


Java
Upcall Layout Preprocessing
1
JAVA_INT -> I : struct [II]
Native
Upcall Signature Representation
2
Metadata
Thunk Generation
3
Upcall Thunk

1. Convert the layout to a simplified signature
2. Encode the native signature to be stored in metadata
3. Allocate memory to generate a thunk being associated with the metadata

Workflow in Upcall (Cont.)

2. Native Invocation via Libffi for Upcall



```
downcall_mh.invoke(
arena, structSegmt1,
structSegmt2,
upcallStubPtr
);
```

```
MemorySegment
upcallStubPtr =
linker.upcallStub(
upcall_mh,
upcall_fd,
arena);
```

```
stru_2Ints add2Structs(…, stru_2Ints (*
upcallStubPtr)(…)) {
  return (*upcallStubPtr)(arg1, arg2);
}
```

```
MemorySegment add2StructsByUpcall(
MemorySegment arg1, MemorySegment
arg2)
{ …}
```

1. Thunk triggered in downcall marshals the passed-in arguments
2. Convert & encapsulate arguments against the metadata in dispatcher
3. Set upcall MH and arguments on the java stack before the JVM callback
4. Extract method information from MH in JVM
5. Invoke the upcall method and return back to the dispatcher
6. Return to the downcall after converting & decapsulating the return value

Downcall by Jextract

DownCall to Standard C Lib (Linux/x86_64)

LibcTest.java (Java source code)

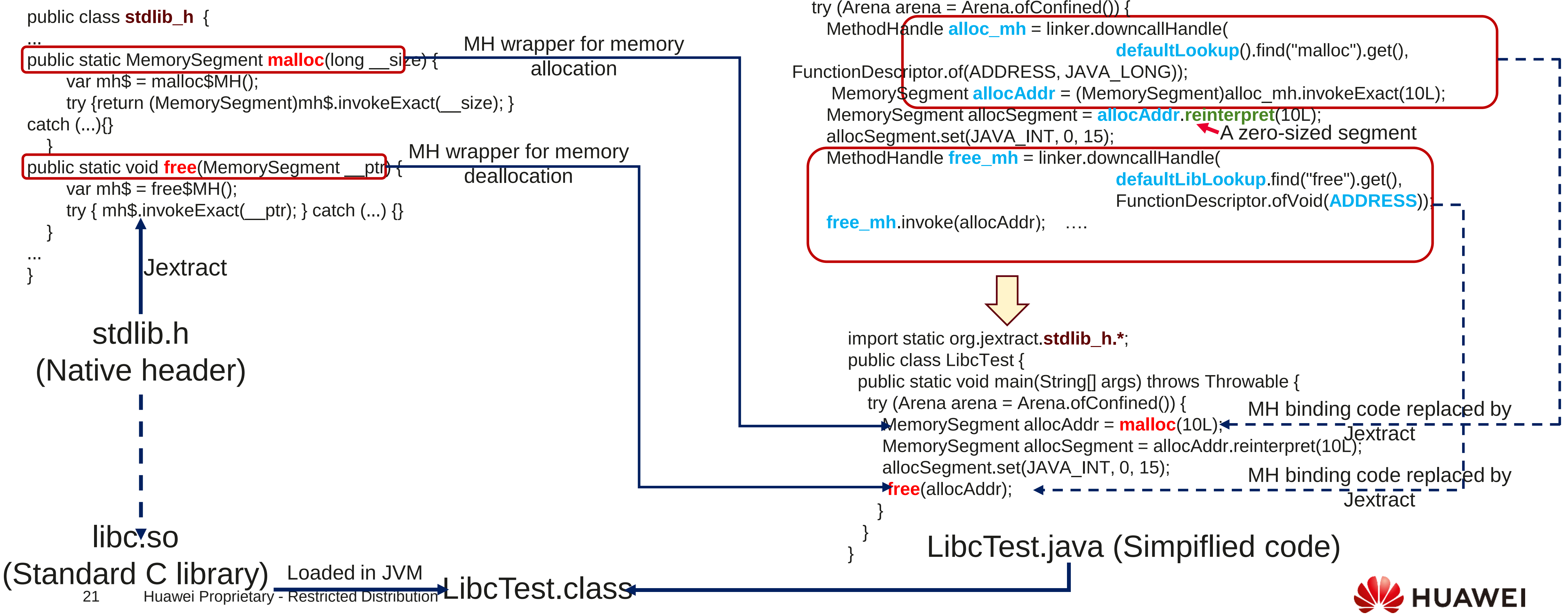
```
public class LibcTest {
  private static Linker linker = Linker.nativeLinker();
  private static final SymbolLookup defaultLibLookup = linker.defaultLookup();

  public static void main(String[] args) throws Throwable {
    try (Arena arena = Arena.ofConfined()) {
      MethodHandle alloc_mh = linker.downcallHandle(
                                    defaultLookup().find("malloc").get(),
FunctionDescriptor.of(ADDRESS, JAVA_LONG));
       MemorySegment allocAddr = (MemorySegment)alloc_mh.invokeExact(10L);
      MemorySegment allocSegment = allocAddr.reinterpret(10L);
      allocSegment.set(JAVA_INT, 0, 15);
      MethodHandle free_mh = linker.downcallHandle(
                                    defaultLibLookup.find("free").get(),
                                    FunctionDescriptor.ofVoid(ADDRESS));
      free_mh.invoke(allocAddr);    ….
```

A zero-sized segment

org/jextract/stdlib_h.java (MH binding code)

```
public class stdlib_h  {
...
public static MemorySegment malloc(long __size) {
        var mh$ = malloc$MH();
        try {return (MemorySegment)mh$.invokeExact(__size); }
catch (...){}
    }
public static void free(MemorySegment __ptr) {
        var mh$ = free$MH();
        try { mh$.invokeExact(__ptr); } catch (...) {}
    }
...
}
```


MH wrapper for memory allocation
MH wrapper for memory deallocation
Jextract
stdlib.h
(Native header)
libc.so
(Standard C library)
Loaded in JVM
LibcTest.class
MH binding code replaced by Jextract
MH binding code replaced by Jextract

```
import static org.jextract.stdlib_h.*;
public class LibcTest {
  public static void main(String[] args) throws Throwable {
    try (Arena arena = Arena.ofConfined()) {
      MemorySegment allocAddr = malloc(10L);
      MemorySegment allocSegment = allocAddr.reinterpret(10L);
      allocSegment.set(JAVA_INT, 0, 15);
      free(allocAddr);
    }
  }
}
```

LibcTest.java (Simpified code)

Downcall with FFM

Native invocation for Struct (Linux/x86_64)

StruTest.java (Java source code)

```
public class StruTest {
… …
 static  GroupLayout structLayout = MemoryLayout.structLayout(JAVA_INT.withName("elem1"),
JAVA_INT.withName("elem2"));
 static VarHandle intHandle1 = structLayout.varHandle(PathElement.groupElement("elem1"));
 static VarHandle intHandle2 = structLayout.varHandle(PathElement.groupElement("elem2"));

   public static void main(String[] args) throws Throwable {
    try (Arena arena = Arena. ofConfined()) {
    SymbolLookup nativeLibLookup = SymbolLookup.libraryLookup(path, arena);
    MemorySegment functionSymbol = nativeLibLookup.find("add2Structs").get();
    FunctionDescriptor fd =   FunctionDescriptor.of(structLayout, structLayout, structLayout);
    MethodHandle mh = linker.downcallHandle(functionSymbol, fd);

     MemorySegment structSegmt1 = arena.allocate(structLayout);
     intHandle1.set(structSegmt1, 1);     intHandle2.set(structSegmt1, 2);
     MemorySegment structSegmt2 = allocator.allocate(structLayout);
     intHandle1.set(structSegmt2, 3);     intHandle2.set(structSegmt2, 4);
     MemorySegment resultSegmt = (MemorySegment)mh.invoke((SegmentAllocator)arena, structSegmt1, structSegmt2);
    }
  }
}
```

Access the struct elements via VarHandle

Determine the element layout of struct by its name

Allocate native memories for structs

Allocate the native memory for the returned struct

StruTest.c (Native code)

```
typedef struct stru_2Ints {
  int elem1;
  int elem2;
} stru_2Ints;

stru_2Ints add2Structs(stru_2Ints arg1, stru_2Ints arg2) {
    stru_2Ints sum;
    sum.elem1 = arg1.elem1 + arg2.elem1;
    sum.elem2 = arg1.elem2 + arg2.elem2;
     return sum ;
}
```

Compiled & Generated by GCC

libffitest.so

Load the shared library

StruTest.class

Upcall with FFM

Upcall for Struct (Linux/x86_64)

StruTest.java (Java source code)

```
public class StruTest {
… …
  static  StructLayout structLayout = MemoryLayout.structLayout(JAVA_INT.withName("elem1"),
JAVA_INT.withName("elem2"));
  static VarHandle intHandle1 = structLayout.varHandle(PathElement.groupElement("elem1"));
  static VarHandle intHandle2 = structLayout.varHandle(PathElement.groupElement("elem2"));

static MemorySegment add2StructsByUpcall(MemorySegment arg1, MemorySegment arg2) {
   MemorySegment resultSegmt = = Arena.global().allocate(structLayout);
   intHandle1.set(resultSegmt, (int)intHandle1.get(arg1) + (int)intHandle1.get(arg2));
    intHandle2.set(resultSegmt, (int)intHandle2.get(arg1) + (int)intHandle2.get(arg2));
    return resultSegmt;
}

  public static void main(String[] args) throws Throwable {
   try (Arena arena = Arena. ofConfined()) {
… …
    MethodHandle mh_upcall = MethodHandles.lookup().findStatic(StruTest.class, "add2StructsByUpcall",
           methodType(MemorySegment.class, MemorySegment.class, MemorySegment.class));
    FunctionDescriptor fd_upcall = FunctionDescriptor.of(structLayout, structLayout, structLayout);
    MemorySegment upcallStubPtr = linker.upcallStub(mh_upcall,fd_upcall, arena);
    MethodHandle mh = linker.downcallHandle(functionSymbol, fd_upcall.appendArgumentLayouts(ADDRESS));

    MemorySegment structSegmt1 = arena.allocate(structLayout);
    intHandle1.set(structSegmt1, 1);     intHandle2.set(structSegmt1, 2);
    MemorySegment structSegmt2 = arena.allocate(structLayout);
    intHandle1.set(structSegmt2, 3);     intHandle2.set(structSegmt2, 4);
    MemorySegment resultSegmt = (MemorySegment)mh.invoke((SegmentAllocator)arena,
                                                                        structSegmt1, structSegmt2,
                                                                        upcallStubPtr);
    }
  }
}
```

StruTest.c (Native code)

```
typedef struct stru_2Ints {
  int elem1;
  int elem2;
} stru_2Ints;

stru_2Ints add2Structs(stru_2Ints arg1, stru_2Ints
arg2, stru_2Ints (* upcallStubPtr)(stru_2Ints,
stru_2Ints)) {
  return (*upcallStubPtr)(arg1, arg2);
}
```

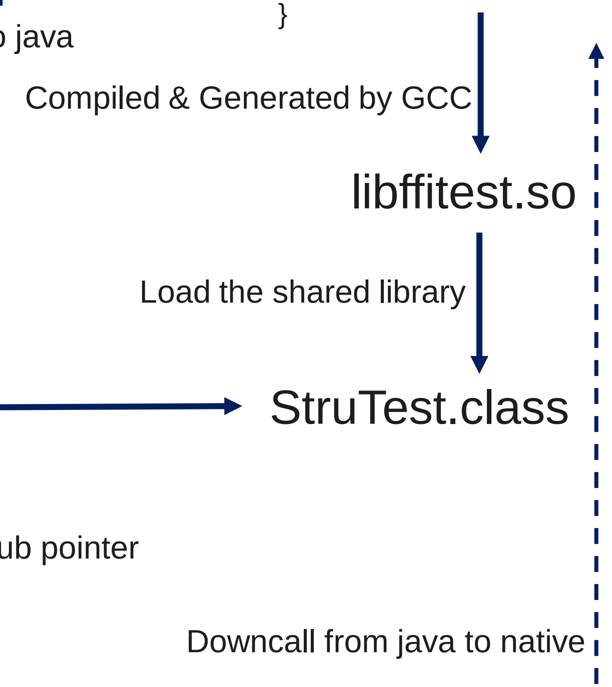
Access the struct elements via VarHandle

Upcall from native to java

Upcall method

Allocate Segments for structs

Layout for the upcall stub pointer



Wrap-up

Pros

- A new generation of FFI framework in JDK that replaces JNI to better support the native library development
- One-stop solution featured with pure Java APIs (No extra rules/protocols mandated in native)
- A better interoperability with the existing/legacy libraries (C/C++)
- A brand new & user-friendly programming pattern supported by Jextract

Cons

- Overall evaluation on various scenarios
- Performance-wise concerns (as compared to JNI/Unsafe)
- Coexistence & interoperability of JNI and FFM
- Leftover issues to be addressed in the follow-up JEPs:
  - Support for unsigned/complex types (needs Valhalla)
  - Mapping between structs and records
  - Integration between Linker and JNI
  - Structured arenas (depends on StructuredTaskScope)
  - Pinning of heap segments

  https://mail.openjdk.org/pipermail/panama-dev/2023-July/019510.html: FFM API summer update

Thank you.

Bring digital to every person, home and organization for a fully connected, intelligent world.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.