MES Wadia College of Engineering Pune-01

Department of Computer Engineering

Name of Student:	**Class:**
Semester/Year:	**Roll No:**
Date of Performance:	**Date of Submission:**
Examined By:	**Experiment No: Group A-03**

Group A ASSIGNMENT NO: 03

AIM: Write a program to create Dynamic Link Library for any mathematical operation and write an application program to test it. (Java Native Interface / Use VB or VC++).

OBJECTIVES:

- To study how to create and use dynamic link library in a java program by using java native interface(JNI).

PRE-REQUISITES:

1. Basics of dynamic linking.
2. Any Java Native Interface.

APPARATUS:

THEORY:

Linking:

Linking is the process of bringing external programs together required by the one we write for its successful execution. Static and dynamic linking are two processes of collecting and combining multiple object files in order to create a single executable.

Static link library:

In computer science, a static library or statically-linked library is a set of routines, external functions and variables which are resolved in a caller at compile-time and copied into a target application by a compiler, linker, or binder, producing an object file and a stand-alone executable. This executable and the process of compiling it are both known as a static build of the program. Historically, libraries could only be *static*. Static libraries are either merged with other static libraries and object files during building/linking to form a single executable or loaded

at run-time into the address space of their corresponding executable at a static memory offset determined at compile-time/link-time.

Dynamic loading is a mechanism by which a computer program can, at run time, load a library (or other binary) into memory, retrieve the addresses of functions and variables contained in the library, execute those functions or access those variables, and unload the library from memory.

Following are the major differences between static and dynamic linking:

	Static Linking	**Dynamic Linking**
1	Static linking is the process of copying all library modules used in the program into the final executable image. This is performed by the linker and it is done as the last step of the compilation process. The linker combines library routines with the program code in order to resolve external references, and to generate an executable image suitable for loading into memory. When the program is loaded, the operating system places into memory a single file that contains the executable code and data. This statically linked file includes both the calling program and the called program.	In dynamic linking the names of the external libraries (shared libraries) are placed in the final executable file while the actual linking takes place at run time when both executable file and libraries are placed in the memory. Dynamic linking lets several programs use a single copy of an executable module.
2	Static linking is performed by programs called linkers as the last step in compiling a program. Linkers are also called link editors.	Dynamic linking is performed at run time by the operating system.
3	Statically linked files are significantly larger in size because external programs are built into the executable files.	In dynamic linking only one copy of shared library is kept in memory. This significantly reduces the size of executable programs, thereby saving memory and disk space.
4	In static linking if any of the external programs has changed then they have to be recompiled and re-linked again else the changes won't reflect in existing executable file.	In dynamic linking this is not the case and individual shared modules can be updated and recompiled. This is one of the greatest advantages dynamic linking offers.
5	Statically linked program takes constant load time every time it is loaded into the memory for execution.	In dynamic linking load time might be reduced if the shared library code is already present in memory.
6	Programs that use statically-linked libraries are usually faster than those that use shared	Programs that use shared libraries are usually slower than those that use

	libraries.	statically-linked libraries.
7	In statically-linked programs, all code is contained in a single executable module. Therefore, they never run into compatibility issues.	Dynamically linked programs are dependent on having a compatible library. If a library is changed (for example, a new compiler release may change a library), applications might have to be reworked to be made compatible with the new version of the library. If a library is removed from the system, programs using that library will no longer work.

Java Native Interface (JNI)

At times, it is necessary to use native codes (C/C++) to overcome the memory management and performance constraints in Java. Java supports native codes via the Java Native Interface (JNI). JNI is difficult, as it involves two languages and runtimes.

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
  - ❍ Code reusability
    - ❒ Reuse existing/legacy code with Java (mostly C/C++)
  - ❍ Performance
    - ❒ Native code used to be up to 20 times faster than Java, when running in interpreted mode
    - ❒ Modern JIT compilers (HotSpot) make this a moot point
  - ❍ Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - ❍ Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - ❍ `javah` automatically includes this file in the application header files

JNI Basics

JNI defines the following JNI types in the native system that correspond to Java types:

1. Java Primitives: `jint`, `jbyte`, `jshort`, `jlong`, `jfloat`, `jdouble`, `jchar`, `jboolean` for Java Primitive of `int`, `byte`, `short`, `long`, `float`, `double`, `char` and `boolean`, respectively.
2. Java Reference Types: `jobject` for `java.lang.Object`. It also defines the following *sub-type*s:
    1. `jclass` for `java.lang.Class`.
    2. `jstring` for `java.lang.String`.
    3. `jthrowable` for `java.lang.Throwable`.
    4. `jarray` for Java array. Java array is a reference type with eight primitive array and one `Object` array. Hence, there are eight array of primitives `jintArray`, `jbyteArray`, `jshortArray`, `jlongArray`, `jfloatArray`, `jdoubleArray`, `jcharArray` and `jbooleanArray`; and one object array `jobjectArray`.

The native programs:

1. Receive the arguments in JNI type (passed over by the Java program).
2. For reference JNI type, convert or copy the arguments to local native types, e.g., `jstring` to a C-string, `jintArray` to C's `int[]`, and so on. Primitive JNI types such as `jint` and `jdouble` do not need conversion and can be operated directly.
3. Perform its operations, in local native type.
4. Create the returned object in JNI type, and copy the result into the returned object.
5. Return.

JNI with C

Java code for loading the libraries:

```
static {

    System.loadLibrary("myLibrary");
}
```

Step 1: Write a Java Class that uses C Codes - HelloJNI.java

```
1  public class HelloJNI {
2     static {
3        System.loadLibrary("hello"); // Load native library at runtime
4                                     // hello.dll (Windows) or libhello.so
5  (Unixes)
6     }
7
8     // Declare a native method sayHello() that receives nothing and returns
9  void
10    private native void sayHello();
```

```
11    // Test Driver
12    public static void main(String[] args) {
13       new HelloJNI().sayHello();  // invoke the native method
14    }
   }
```

The static initializer invokes `System.loadLibrary()` to load the native library "`Hello`" (which contains the native method `sayHello()`) during the class loading. It will be mapped to "`hello.dll`" in Windows; or "`libhello.so`" in Unixes. This library shall be included in Java's library path (kept in Java system variable `java.library.path`); otherwise, the program will throw a `UnsatisfiedLinkError`. You could include the library into Java Library's path via VM argument `-Djava.library.path=`*path_to_lib*.

Next, we declare the method `sayHello()` as a native instance method, via keyword `native`, which denotes that this method is implemented in another language. A native method does not contain a body. The `sayHello()` is contained in the native library loaded.

The `main()` method allocate an instance of `HelloJNI` and invoke the native method `sayHello()`.

Compile the "`HelloJNI.java`" into "`HelloJNI.class`".

```
> javac HelloJNI.java
```

Step 2: Create the C/C++ Header file - HelloJNI.h

Run `javah` utility on the class file to create a header file for C/C++ programs:

```
> javah HelloJNI
```

The output is `HelloJNI.h` as follows:

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class HelloJNI */
4
5  #ifndef _Included_HelloJNI
6  #define _Included_HelloJNI
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:     HelloJNI
12  * Method:    sayHello
13  * Signature: ()V
14  */
15 JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);
16
17 #ifdef __cplusplus
18 }
19 #endif
20 #endif
```

The header declares a C function `Java_HelloJNI_sayHello` as follows:

```
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);
```

The naming convention for C function is `Java_{package_and_classname}_{function_name}(JNI arguments)`. The dot in package name shall be replaced by underscore.

The arguments:

- `JNIEnv *env`: Is a pointer that points to another pointer pointing to a function table (array of pointer). Each entry in this function table points to a JNI function. These are the functions we are going to use for type conversion
- The second argument is different depending on whether the native method is a static method or an instance method
  - ❒ Instance method: It will be a `jobject` argument which is a reference to the object on which the method is invoked
  - ❒ Static method: It will be a `jclass` argument which is a reference to the class in which the method is define

We are not using these arguments in the hello-world example, but will be using them later. Ignore the macros `JNIEXPORT` and `JNICALL` for the time being.

The `extern "C"` is recognized by C++ compiler only. It notifies the C++ compiler that these functions are to be compiled using C's function naming protocol (instead of C++ naming protocol). C and C++ have different function naming protocols as C++ support function overloading and uses a name mangling scheme to differentiate the overloaded functions.

Step 3: C Implementation - HelloJNI.c

```
1 #include <jni.h>
2 #include <stdio.h>
3 #include "HelloJNI.h"
4
5 // Implementation of native method sayHello() of HelloJNI class
6 JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
7    printf("Hello World!\n");
8    return;
9 }
```

$ **gcc -shared -fPIC -I/usr/lib/jvm/default-java/include -I/usr/lib/jvm/default-java/include/linux HelloWorld.c -o libHelloWorld.so**

`-I`: for specifying the header files directories. In this case "`jni.h`" (in "`<JAVA_HOME>\include`") and "`jni_md.h`" (in "`<JAVA_HOME>\include\linux`"), where `<JAVA_HOME>` denotes the JDK installed directory. Enclosed the directory in double quotes if it contains spaces.

- `-shared`: to generate share library.
- `-o`: for setting the output filename "`libHelloWorld.so`".

How to Load a Java Native/Shared Library (.so)

1. Call **System.load** to load the .so from an explicitly specified absolute path.
2. Copy the shared library to one of the paths already listed in **java.library.path**
3. Modify the LD_LIBRARY_PATH environment variable to include the directory where the shared library is located.
4. Specify the java.library.path on the command line by using the **-D option**.

Steps to run a program on terminal

admin1@admin1:~/HelloWorld$ **javac HelloWorld.java**

admin1@admin1:~/HelloWorld1$ **javah -classpath . HelloWorld**

admin1@admin1:~/HelloWorld1$ **gcc -shared -fPIC -I/usr/lib/jvm/default-java/include -I/usr/lib/jvm/default-java/include/linux HelloWorld.c -o libHelloWorld.so**

admin1@admin1:~/HelloWorld1$ **java -classpath . -Djava.library.path=. HelloWorld**
Hello World!

CONCLUSION:

QUESTIONS:

1. Explain linking with example.
2. Explain loading with example.
3. What are the advantages of dynamic linking library?
4. What are the advantages and disadvantages of static linking library?