Schema Sharing between Android Runtime (ART) and Bionic

Our plan is to pass a pointer object from `java_vm_ext.cc` inside `art` project to `linker_phdr.cpp` inside `bionic` project. The pointer object is supposed to be pointed to by a static member of `linker_phdr.cpp` as a way of remembering it. As the pointer is initialized inside `art` can be pass along to other `Runtime` instances and be manipulated for its values.

Adding Modified Functions

ART Java VM Hooks

1. Go to `JNI_CreateJavaVM` function inside `art/runtime/jni/java_vm_ext.cc` and find `android::InitializeNativeLoader();`. Comment it out and insert new code as shown below:

```
// Initialize native loader. This step makes sure we have
// everything set up before we start using JNI.

// android::InitializeNativeLoader();

// ADD THIS
char* wei_ptr = (char*) "Initialized in java_vm_ext.cc";
android::InitializeNativeLoader2(wei_ptr);
// END OF ADD
```

ART Native Loader

1. Go to `art/libnativeloader/include/nativeloader/native_loader.h` and find the declaration `void InitializeNativeLoader();`. Below it, add the declaration for new function `void InitializeNativeLoader2(char* wei_ptr);` as shown below:

```
// README: the char** error message parameter being passed
// to the methods below need to be freed through calling
NativeLoaderFreeErrorMessage.
// It's the caller's responsibility to call that method.

__attribute__((visibility("default")))
void InitializeNativeLoader();

// ADD THIS
__attribute__((visibility("default")))
void InitializeNativeLoader2(char* wei_ptr);
// END OF ADD
```

2. Go to `art/libnativeloader/include/nativeloader/native_loader.cpp` and find the definition of `InitializeNativeLoader`. Below it, add a new definition for new function `InitializeNativeLoader2` as shown below:

```
void InitializeNativeLoader() {
  ALOGW("[weiminn] native_loader.cpp: InitializeNativeLoader1");
#if defined(ART_TARGET_ANDROID)
  std::lock_guard<std::mutex> guard(g_namespaces_mutex);
  g_namespaces->Initialize();
#endif
}

// ADD THIS
void InitializeNativeLoader2(char* wei_ptr) {
  ALOGW("[weiminn] native_loader.cpp: InitializeNativeLoader2 | Received
data: %s", wei_ptr);
#if defined(ART_TARGET_ANDROID)
  std::lock_guard<std::mutex> guard(g_namespaces_mutex);
  g_namespaces->Initialize2(wei_ptr);
#endif
}
// END OF ADD
```

3. Go to `art/libnativeloader/library_namespaces.h` and find the declaration `void Initialize();`. Below it, add the declaration for new function `void Initialize2(char* wei_ptr);` as shown below:

```
void Initialize();

// ADD THIS
void Initialize2(char* wei_ptr);
// END OF ADD
```

4. Go to `art/libnativeloader/library_namespaces.cpp` and find the definition of `LibraryNamespaces::Initialize`. Below it, add a new definition for new function `LibraryNamespaces::Initialize2` as shown below:

```
void LibraryNamespaces::Initialize() {
  // Once public namespace is initialized there is no
  // point in running this code - it will have no effect
  // on the current list of public libraries.
  if (initialized_) {
    return;
  }

  // Load the preloadable public libraries. Since libnativeloader is in the
  // com_android_art namespace, use OpenSystemLibrary rather than dlopen to
```

```
  // ensure the libraries are loaded in the system namespace.
  //
  // TODO(dimitry): this is a bit misleading since we do not know
  // if the vendor public library is going to be opened from /vendor/lib
  // we might as well end up loading them from /system/lib or /product/lib
  // For now we rely on CTS test to catch things like this but
  // it should probably be addressed in the future.
  for (const std::string& soname :
android::base::Split(preloadable_public_libraries(), ":")) {
    void* handle = OpenSystemLibrary(soname.c_str(), RTLD_NOW |
RTLD_NODELETE);
    LOG_ALWAYS_FATAL_IF(handle == nullptr,
                        "Error preloading public library %s: %s",
soname.c_str(), dlerror());
  }
}


// ADD THIS
void LibraryNamespaces::Initialize2(char* wei_ptr) {
  ALOGW("[weiminn] library_namespaces.cpp: Initialize2");

  if (initialized_) {
    return;
  }

  for (const std::string& soname :
android::base::Split(preloadable_public_libraries(), ":")) {
    ALOGW("[weiminn] library_namespaces.cpp: Initialize2 is opening %s",
soname.c_str());

    void* handle = OpenSystemLibrary2(soname.c_str(), RTLD_NOW |
RTLD_NODELETE, wei_ptr);

    LOG_ALWAYS_FATAL_IF(handle == nullptr,
                        "Error preloading public library %s: %s",
soname.c_str(), dlerror());
  }
}
// END OF ADD
```

ART Native Bridge

1. Go to `art/libnativebridge/include/nativebridge/native_bridge.h` and find the declaration `void* OpenSystemLibrary(const char* path, int flags);`. Below it, add the declaration for new function `void* OpenSystemLibrary2(const char* path, int flags, char* wei_ptr);` as shown below:

```
// Loads a shared library from the system linker namespace, suitable for
// platform libraries in /system/lib(64). If linker namespaces don't exist
(i.e.
```

```
// on host), this simply calls dlopen().
void* OpenSystemLibrary(const char* path, int flags);

// ADD THIS
void* OpenSystemLibrary2(const char* path, int flags, char* wei_ptr);
// END OF ADD
```

2. Go to `art/libnativebridge/native_bridge.cc` and find the definition of `OpenSystemLibrary`. Below it, add a new definition for new function `InitializeNativeLoader2` as shown below:

```
void* OpenSystemLibrary(const char* path, int flags) {
#ifdef ART_TARGET_ANDROID
  // The system namespace is called "default" for binaries in /system and
  // "system" for those in the Runtime APEX. Try "system" first since
  // "default" always exists.
  // TODO(b/185587109): Get rid of this error prone logic.
  ALOGW("[weiminn] native_bridge.cc: OpenSystemLibrary %s", path);
  android_namespace_t* system_ns =
android_get_exported_namespace("system");
  if (system_ns == nullptr) {
    system_ns = android_get_exported_namespace("default");
    LOG_ALWAYS_FATAL_IF(system_ns == nullptr,
                        "Failed to get system namespace for loading %s",
path);
  }
  const android_dlextinfo dlextinfo = {
      .flags = ANDROID_DLEXT_USE_NAMESPACE,
      .library_namespace = system_ns,
  };
  return android_dlopen_ext(path, flags, &dlextinfo);
#else
  return dlopen(path, flags);
#endif
}

// ADD THIS
void* OpenSystemLibrary2(const char* path, int flags, char* wei_ptr) {
#ifdef ART_TARGET_ANDROID
  // The system namespace is called "default" for binaries in /system and
  // "system" for those in the Runtime APEX. Try "system" first since
  // "default" always exists.
  // TODO(b/185587109): Get rid of this error prone logic.
  ALOGW("[weiminn] native_bridge.cc: From Initializer: OpenSystemLibrary
%s", path);
  android_namespace_t* system_ns =
android_get_exported_namespace("system");
  if (system_ns == nullptr) {
    system_ns = android_get_exported_namespace("default");
    LOG_ALWAYS_FATAL_IF(system_ns == nullptr,
                        "Failed to get system namespace for loading %s",
```

```
path);
  }

  // char* wei_ptr = (char*) "Assigned in native_bridge";
  const android_dlextinfo dlextinfo = {
      .flags = ANDROID_DLEXT_USE_NAMESPACE,
      .library_namespace = system_ns,
      .weiminn_msg = wei_ptr
  };
  return android_dlopen_ext(path, flags, &dlextinfo);
  // return wei_ptr;
#else
  ALOGW("[weiminn] [placeholder workaround]: %s", wei_ptr); // work around
to prevent -Wunused-parameter exceptions
  return dlopen(path, flags);
#endif
}
// END OF ADD
```

Bionic Libc

1. Go to `bionic/libc/include/android/dlext.h` and find the `alias` definition for `android_dlextinfo`. Inside it, add a new member `char* weiminn_msg` as shown below:

```
/** Used to pass Android-specific arguments to `android_dlopen_ext`. */
typedef struct {
  /** A bitmask of `ANDROID_DLEXT_` enum values. */
  uint64_t flags;

  /** Used by `ANDROID_DLEXT_RESERVED_ADDRESS` and
`ANDROID_DLEXT_RESERVED_ADDRESS_HINT`. */
  void*   reserved_addr;
  /** Used by `ANDROID_DLEXT_RESERVED_ADDRESS` and
`ANDROID_DLEXT_RESERVED_ADDRESS_HINT`. */
  size_t  reserved_size;

  /** Used by `ANDROID_DLEXT_WRITE_RELRO` and `ANDROID_DLEXT_USE_RELRO`. */
  int     relro_fd;

  /** Used by `ANDROID_DLEXT_USE_LIBRARY_FD`. */
  int     library_fd;
  /** Used by `ANDROID_DLEXT_USE_LIBRARY_FD_OFFSET` */
  off64_t library_fd_offset;

  /** Used by `ANDROID_DLEXT_USE_NAMESPACE`. */
  struct android_namespace_t* library_namespace;

  // ADD THIS
  char* weiminn_msg;
  // END OF ADD
```

```
} android_dlextinfo;
```

Bionic Linker

1. Go to `bionic/linker/dlfcn.cpp` and find the definition of `__loader_android_dlopen_ext`. Inside it, put additional code for printing message and calling a function of `linker_phdr.cpp` as shown below:

```
void* __loader_android_dlopen_ext(const char* filename,
                           int flags,
                           const android_dlextinfo* extinfo,
                           const void* caller_addr) {

  // ADD THIS
  if(extinfo->weiminn_msg != nullptr){
    LD_LOG(kLogErrors, "[weiminn] dlfcn.cpp RECEIVED MESSAGE ART!!!: %s |
Opening: %s", static_cast<char*>(extinfo->weiminn_msg),filename);
    weiminn_linker_phdr_print2(extinfo->weiminn_msg);
  } else {
    LD_LOG(kLogErrors, "[weiminn] dlfcn.cpp Opening: %s",filename);
  }
  // END OF ADD

  return dlopen_ext(filename, flags, extinfo, caller_addr);
}
```

2. Go to `bionic/linker/linker_phdr.cpp` and at the bottom of the file add this function:

```
void weiminn_linker_phdr_print2(char* wei_ptr){
  DL_ERR_AND_LOG("[weiminn] linker_phdr.cpp *****ACCOMPLISHED***** | called
from dlfcn.cpp | %s", wei_ptr);
}
```

Fix ABI Compliance

Because we changed the format of `android_dlextinfo` inside header file included by `libdl` library, and changes to `libdl` (and also all other shared system libraries) are being monitored strictly by the Android build system (See ABI Compliance), we will encounter error if we go ahead and `make` the project. To fix this, we need to redump ABI references of `libdl` by running following command:

```
python3 development/vndk/tools/header-
checker/utils/create_reference_dumps.py -l libdl
```

Afterwards, you can go ahead with compiling, running and logging the AOSP build:

```
$ make
$ adb reboot bootloader && fastboot -w flashall
```

Immediately after the new OS finished installing into the phone, run:

```
$ adb root && adb shell setprop debug.ld.all dlerror,dlopen && rm
logcat.txt && adb logcat >> logcat.txt
```