Android Systems Programming Using the Java Language

Linux Plumbers 2016

Elliott Hughes (Google)

How did this start?

- You may know me as the “Android native tools/libraries” guy, but...
- I joined Android in the Donut/Eclair timeframe to work on “libcore”: Android’s core Java libraries (java.net, java.util, et cetera)
- Moved down a layer to work on ART when that project started
- Moved down a layer to work on native tools / libraries when the first release of ART was basically done
- But one of my first jobs on Android was cleaning up the native code...

Libcore native code rewrite #1

- What was wrong with the C?
  - Resource leaks (“goto fail”)
  - Native crashes (or, as we’d call them today, “security vulnerabilities”)
  - Bad error reporting (throwing away all information on the way back up into Java)
  - Hard to debug (Java programmers [and their tools] largely give up at the JNI boundary)
- Switched to C++
- RAII solved the leaks and crashes
- Still lots of logic implemented in native code (differences between Java and Unix semantics)

Libcore native code rewrite #2

- Rather than implement Java semantics in native code…
- ...expose “POSIX” as native methods callable from Java
- Native code just marshalls arguments/results
- Logic moves up into Java, making debugging easier
- Best of all: exposing low-level primitives vastly increases possible use cases

Example implementations

- Helpers translate int<->FileDescriptor, “return -1 and set errno” failures
- Dup2 trivial:

```
static jobject Posix_dup2(JNIEnv* env, jobject, jobject javaOldFd, jint newFd) {
   int oldFd = jniGetFDFromFileDescriptor(env, javaOldFd);
   int fd = throwIfMinusOne(env, "dup2", TEMP_FAILURE_RETRY(dup2(oldFd, newFd)));
   return (fd != -1) ? jniCreateFileDescriptor(env, fd) : NULL;
 }
```

- setuid is an actual one-liner:

```
 static void Posix_setuid(JNIEnv* env, jobject, jint uid) {
   throwIfMinusOne(env, "setuid", TEMP_FAILURE_RETRY(setuid(uid)));
 }
```

- https://android.googlesource.com/platform/libcore/+/master/luni/src/main/native/libcore_io_Posix.cpp

What happened next?

- Became popular outside libcore...
- First used in frameworks to replace old JNI or avoid writing new JNI
  - JNI is unnecessarily painful
  - Very few people know how to write correct JNI
  - Luckily, no one really wants to!
- Then folks writing unbundled apps wanted this too

Making POSIX generally available

- Added `android.system` package to API 21 (Android 5.0 Lollipop)
- https://developer.android.com/reference/android/system/package-summary.html
- Os : big bag of static methods (such as `prctl`)
- OsContants : big bag of static constants (such as `PR_GET_DUMPABLE`)
- Handful of “structs” (such as StructStat for `struct stat`)
- ErrnoException : thrown if an Os method’s implementation sets errno

Design choices: translating C types

- Signed integers easy, unsigned not available
- char*/char*+size_t out parameters -> return String
- char*/char*+size_t/iovec[] in parameters -> Object
  - Public API has two overloads: byte[] and ByteBuffer
  - Non-direct ByteBuffer passed to native code as underlying byte[]
  - Direct ByteBuffer handled separately
  - Helper makes most of this transparent on native side
- Structs get equivalent Java classes
  - New instances returned, rather than filling out existing ones

Design choices: ErrnoException

- A previous smaller-scale version used -errno style
- Exceptions work better with string/struct return values
- Most calling code doesn't handle each failure individually ("goto fail")
- Exceptions work well with finally
- ErrnoException exposes errno for code like if (e.errno == EEXIST) …
- Also include string name of function
- Leads to messages like "open failed: ENOENT (No such file or directory)"
- Not a checked exception

Design choices: OsConstants

- Papers over the fact that Linux “constants” vary between architectures
- Simple Java trick keeps syntax clean:

```
public static final int O_RDONLY = placeholder();
private static int placeholder() { return 0; } /* Prevents javac from inlining. */
private static native void initConstants(); /* JNI can assign final fields! */
static { initConstants(); }
```

- Works well with import static
- Leads to Java code that looks very like C:

```
FileDescriptor s = Os.socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

Design choices: misc

- Implicit TEMP_FAILURE_RETRY (but asynchronous close() monitoring).
- Return new structs: safer and “cheap enough”.
- Public fields versus constructors: most structs have lots of fields, hard to make meaningful constructors.
- Could have overloaded stuff like ioctl/setsockopt but they’re confusing enough already.
- Great StrictMode coverage (warnings about blocking the UI event thread) because syscalls act as bottlenecks to really check “are you touching network/disk” (fun with things like SO_LINGER).

Example Uses

- Lots of networking uses: netlink, DHCP, advanced socket types (SOCK_SEQPACKET), fine setsockopt control.
- Lots of dup/lseek calls on FileDescriptors in media code.
- Extended attributes (getxattr/setxattr).
- stat/lstat (and S_ISLNK) popular with code that needs to traverse files.
- Access to errno turns out to be very popular: knowing whether you got EACCES or ENODEV, say.
- Very useful in CTS tests to see what's really going on/set up unusual state.
- ...and of course implementing java.* functionality.

Thanks!

Kernel topics from a bionic perspective

- The uapi headers are not hermetic.
- The uapi headers are not fine-grained enough.
- Thoughts on which syscalls to add bionic wrappers for, and when?