Debugging Native Android Platform Code

This page contains a summary of useful tools and related commands for debugging, tracing, and profiling native Android platform code. The pages within this section contain detailed information on other debugging tools for use during development of platform-level features.

For example, you may learn how to explore system services with Dumpsys (https://source.android.com/devices/tech/debug/dumpsys.html) and evaluate network (https://source.android.com/devices/tech/debug/netstats.html) and RAM (https://source.android.com/devices/tech/debug/procstats.html) use. See the subpages for tools and methods not described below.

debuggerd

The `debuggerd` process dumps registers and unwinds the stack. When a dynamically-linked executable starts, several signal handlers are registered that connect to `debuggerd` (or `debuggerd64`) in the event that signal is sent to the process.

It's possible for `debuggerd` to attach only if nothing else is already attached. This means that using tools like `strace` or `gdb` will prevent `debuggerd` from working. Also, if you call `prctl(PR_SET_DUMPABLE, 0)` you can prevent `debuggerd` from attaching. This can be useful if you wish to explicitly opt out of crash reporting.

Here is example output (with timestamps and extraneous information removed):

```
*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint: 'Android/aosp_flounder/flounder:5.1.51/AOSP/enh08201009:eng/test-keys'
Revision: '0'
ABI: 'arm'
pid: 1656, tid: 1656, name: crasher  >>> crasher <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
Abort message: 'some_file.c:123: some_function: assertion "false" failed'
    r0 00000000  r1 00000678  r2 00000006  r3 f70b6dc8
    r4 f70b6dd0  r5 f70b6d80  r6 00000002  r7 0000010c
    r8 ffffffed  r9 00000000  sl 00000000  fp ff96ae1c
    ip 00000006  sp ff96ad18  lr f700ced5  pc f700dc98  cpsr 400b0010
backtrace:
    #00 pc 00042c98  /system/lib/libc.so (tgkill+12)
    #01 pc 00041ed1  /system/lib/libc.so (pthread_kill+32)
    #02 pc 0001bb87  /system/lib/libc.so (raise+10)
    #03 pc 00018cad  /system/lib/libc.so (__libc_android_abort+34)
    #04 pc 000168e8  /system/lib/libc.so (abort+4)
    #05 pc 0001a78f  /system/lib/libc.so (__libc_fatal+16)
    #06 pc 00018d35  /system/lib/libc.so (__assert2+20)
    #07 pc 00000f21  /system/xbin/crasher
    #08 pc 00016795  /system/lib/libc.so (__libc_init+44)
    #09 pc 00000abc  /system/xbin/crasher
Tombstone written to: /data/tombstones/tombstone_06
```

This can be pasted into `development/scripts/stack` to get a more detailed unwind with line number information (assuming the unstripped binaries can be found).

Some libraries on the system are built with `LOCAL_STRIP_MODULE := keep_symbols` to provide usable backtraces directly from `debuggerd`. This makes your library or executable slightly larger, but not nearly as large as an unstripped version.

Note also the last line of `debuggerd` output — in addition to dumping a summary to the log, `debuggerd` writes a full "tombstone" to disk. This contains a lot of extra information that can be helpful in debugging a crash, in particular the stack traces for all the threads in the crashing process (not just the thread that caught the signal) and a full memory map.

For more information about diagnosing native crashes and tombstones, see Diagnosing Native Crashes (https://source.android.com/devices/tech/debug/native-crash.html)

Native Debugging with GDB

Debugging a running app

To connect to an already-running app or native daemon, use `gdbclient`.

Current versions of gdbclient just require the process ID (PID). So to debug a process with PID 1234, simply run:

```
$ gdbclient 1234
```

The script will set up port forwarding, start the appropriate `gdbserver` on the device, start the appropriate `gdb` on the host, configure `gdb` to find symbols, and connect `gdb` to the remote `gdbserver`.

Debugging a native process as it starts

If you want to debug a process as it starts, you'll need to use `gdbserver` or `gdbserver64` manually, but that's easy too:

```
$ adb shell gdbserver :5039 /system/bin/my_test_app
Process my_test_app created; pid = 3460
Listening on port 5039
```

Identify the app's PID from the `gdbserver` output, and then in another window:

```
$ gdbclient <app pid>
```

Then enter **continue** at the `gdb` prompt.

Note that to debug a 64-bit process, you'll need to use **gdbserver64**. The error messages from **gdb** if you made the wrong choice are unhelpful (along the lines of **Reply contains invalid hex digit 59**).

Debugging processes that crash

If you want **debuggerd** to suspend crashed processes so you can attach **gdb**, set the appropriate property:

```
$ adb shell setprop debug.db.uid 999999                # <= M
$ adb shell setprop debug.debuggerd.wait_for_gdb true   # > M
```

At the end of the usual crash output, **debuggerd** will give you instructions on how to connect **gdb** using the typical command:

```
$ gdbclient <pid>
```

Debugging without symbols

If you don't have symbols, sometimes **gdb** will get confused about the instruction set it is disassembling (ARM or Thumb). The instruction set that is chosen as the default when symbol information is missing can be switched between ARM or Thumb like so:

```
$ set arm fallback-mode arm   # or 'thumb'
```

Other tools

Valgrind

The following steps show you how to use Valgrind (http://valgrind.org/) on Android. This tool suite contains a number of tools including Memcheck for detecting memory-related errors in C and C++.

Android platform developers usually use AddressSanitizer (ASan) (https://source.android.com/devices/tech/debug/asan.html) rather than valgrind.

1. To build Valgrind, run:

   ```
   $ mmma -j6 external/valgrind
   ```

2. Set up the temporary directory:

   ```
   $ adb shell mkdir /data/local/tmp
   $ adb shell chmod 777 /data/local/tmp
   ```

3. Run the system server with Valgrind:

   ```
   $ adb shell setprop wrap.system_server "logwrapper valgrind"
   $ adb shell stop && adb shell start
   ```

4. For debug symbols, push unstripped libraries to **/data/local/symbols**:

   ```
   $ adb shell mkdir /data/local/symbols
   $ adb push $OUT/symbols /data/local/symbols
   ```

5. To use Valgrind during boot up, edit **out/target/product/XXXX/root/init.rc** and change:
   service example /system/bin/foo --arg1 --arg2
   to:
   service example /system/bin/logwrapper /system/bin/valgrind /system/bin/foo --arg1 --arg2
   To see the effects, you need to create a **boot.img** and reflash the device.

Systrace

See Systrace on developer.android.com (https://developer.android.com/tools/help/systrace.html) for deriving execution times of applications and other Android system processes.

Last updated March 27, 2017.