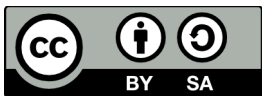
Hands-On Exercises for

Android Debugging and Performance Analysis

v. 2018.10

WARNING:
The order of the exercises does not always follow the same order of the explanations in the slides. When carrying out the exercises, carefully follow the exercise requirements. Do **NOT** blindly type the commands or code as found in the slides. Read every exercise **in its entirety** before carrying out the instructions.



These exercises created by: Karim Yaghmour

Originals at: www.opersys.com/training/android-debug-and-performance

---

Delivered and/or customized by:

Class Preparation for HiKey board

1. Install Android Studio:
https://developer.android.com/studio/index.html
This URL should give you access to all the instructions needed to get this installed and working.

2. Install the required packages to build Android -- See the "Installing required packages" instructions for the relevant Ubuntu version here:
https://source.android.com/setup/initializing

3. Fetch the AOSP:
We'll be using 8.1/Oreo. Note that the new lines that start with ">" are printed by the shell. Adding a ">" to the input you type will break the following commands. Assuming you have a directory called "android" in your home directory:

```
$ cd ~/android
$ mkdir aosp-8.1.0_r18
$ cd aosp-8.1.0_r18
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-8.1.0_r18
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

4. Patch 8.1.0_r18 to fix it for HiKey board. These instructions need to be done in the root of the AOSP just downloaded above:

```
$ wget http://opersys.com/downloads/hikey-8.1.0_r18-181008.patch
$ patch -p1 < hikey-8.1.0_r18-181008.patch
```

5. Fetch the binaries for the HiKey board (HDMI support). These instructions are also done in the AOSP root:

```
$ wget \
> https://dl.google.com/dl/android/aosp/linaro-hikey-20170523-4b9ebaff.tgz
$ tar xzf linaro-hikey-20170523-4b9ebaff.tgz
$ ./extract-linaro-hikey.sh
```

You will need to read through the license and type "I ACCEPT" at the end for the binaries to extract.

6. Apply changes for the hands-on exercises:
For the hands-on sessions, we're going to use some custom modifications to the AOSP in order to add our own instrumentation and measure performance. Note that the new lines that start with ">" are printed by the shell. Adding a ">" to the input you type will break the following commands. Assuming you're still in the "`aosp-8.1.0_r18/`" directory:

```
$ mkdir -p ../orig-files/frameworks-base-services-core-jni/
$ cp device/linaro/hikey/{device-common.mk, init.common.rc, \
```

```
> ueventd.common.rc} ../orig-files
$ cp system/sepolicy/private/{file_contexts, hwservice_contexts, \
> service_contexts} ../orig-files
$ cp system/sepolicy/public/init.te ../orig-files
$ cp frameworks/base/Android.mk ../orig-files
$ cp frameworks/base/services/core/jni/{Android.mk, onload.cpp} \
> ../orig-files/frameworks-base-services-core-jni/
$ cp frameworks/base/services/java/com/android/server/SystemServer.java \
> ../orig-files/
$ cd ..
$ wget \
> http://opersys.com/downloads/libhw-opersys-hikey-8.1-full-180912.tar.bz2
$ tar xvjf libhw-opersys-hikey-8.1-full-180912.tar.bz2
$ find libhw-opersys-hikey-8.1-full-180912 -exec touch {} \;
$ cp -r libhw-opersys-hikey-8.1-full-180912/* aosp-8.1.0_r18
```

7. Make sure you have “mtools” installed:

```
$ sudo apt-get install mtools
```

8. Build the AOSP:

```
$ cd ~/android/aosp-8.1.0_r18
$ . build/envsetup.sh
$ lunch hikey-eng
$ make -j8
```

9. Ensure that the build finishes successfully.

10. Flash your board with your AOSP:
    (a) Plug your board into the power jack and your computer through the USB cable. The board already has a system image on it that it'll boot with.
    (b) Make sure your workstation recognizes the device and gives you access to it. Have a look at the instructions here for more information: https://developer.android.com/studio/run/device.html#setting-up
    (c) Put the board in “fastboot” mode:

```
$ cd ~/android/aosp-8.1.0_r34
$ . build/envsetup.sh
$ lunch hikey-eng
$ adb reboot bootloader
```

    (d) The board should reboot and you should be able to see it using this command – example output provided:

```
$ fastboot devices
76B8809900099B63  fastboot
```

    (e) Reflash with your own images:

```
$ fastboot flashall
target reported max download size of 134217728 bytes
------------------------------------------
```

```
Bootloader Version...: 0.4
Baseband Version.....: 0.4
Serial Number........: 76B8809900099B63
----------------------------------------
checking product...
OKAY [  0.001s]
checking version-bootloader...
OKAY [  0.001s]
sending 'boot' (24380 KB)...
OKAY [  0.818s]
writing 'boot'...
...
OKAY [  3.097s]
writing 'system' 7/7...
OKAY [  3.819s]
rebooting...

finished. total time: 66.887s
```

(f) Check that your device booted properly into Android mode:

```
$ adb devices
List of devices attached
76B8809900099B63  device
```

(g) Log into your device:

```
$ adb shell
hikey:/ #
```

(h) Check that the build description matches your build date/time:

```
hikey:/ # cat system/build.prop | grep -i description
# Do not try to parse description, fingerprint, or thumbprint
ro.build.description=hikey-eng 8.1.0 OPM5.171019.017 eng.karim.20180308.124948 test-keys
```

AOSP sources and symbolic debugging

1. Generate and run the idegen script

2. Add the Android's sources as a Java project

3. Explore Android Studio's Java sources browsing capabilities (i.e. try out the shortcuts suggested a developer.android.com)

4. With the system_server process selected in the Android Device Monitor, set a breakpoint in frameworks/base/services/core/java/com/android/server/statusbar/StatusBarManagerService.java:expandNotificationsPanel(). Use “service call statusbar 1” to expand the status bar. Eclipse should now break into the debug view at the breakpoint you selected. Now, you can step in the Status Bar Manager's code for expanding the notifications panel.

5. Do the same as the previous exercise but with OpersysService.java. Use “service call opersys 2 s16 foobar” to send the “foobar” string to the Opersys system service. User “service call opersys 1 i32 45” to read 45 bytes from the Opersys system service.

6. Use gdbclient on the host to step from the JNI side of the system service all the way down the driver call. IOW, add a breakpoint to init_native(), read_native() and write_native() and follow the calls all the way to the corresponding open(), read() and write() occurring in the opersyshw HAL.

7. Full system service debugging:

- With the system_server process attached to using Android Studio, set breakpoints in the Opersys system service added earlier, namely in read() and write()
- Use gdb to set breakpoints in the system service's JNI code: init_native(), read_native() and write_native()
- Have the system service invoked using “service call opersys ...”
- Step through the code starting from the system service's java code all the way down to the HAL module's writing to the device driver.

Tracing with ftrace

1. Reconfigure the kernel to support ftrace, if it's not already enable:

a. The “function” and “function_graph” tracers. Make sure you enable CONFIG_DYNAMIC_FTRACE. On a Qualcomm board, if need be, disable CONFIG_STRICT_MEMORY_RWX.

b. Modules and module unloading. You don't need to enable forced module unloading.

Here's a reminder on how to get the kernel configuration that uses the default “hikey” configuration as its starting point:

```
$ cd kernel/hikey-linaro/
$ make ARCH=arm64 hikey_defconfig
$ make ARCH=arm64 menuconfig
```

2. Rebuild, reflash, reboot. To rebuild the kernel, you need to do the following from the “kernel/hikey-linaro/” directory -- note the “...” which you'll have to replace with the path to your AOSP:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-android- -j8
```

In order for the AOSP to take your newly built kernel into account, you'll need to do this -- the following assuming that you've run the appropriate “. build/envsetup.sh” and “lunch” from the toplevel:

```
$ croot
$ cd device/linaro/hikey-kernel/
$ cp hi6220-hikey.dtb-4.9 hi6220-hikey.dtb-4.9-orig
$ cp Image-dtb-4.9 Image-dtb-4.9-orig
$ croot
```

```
$ cd kernel/hikey-linaro
$ cp arch/arm64/boot/dts/hisilicon/hi6220-hikey.dtb \
> ../../device/linaro/hikey-kernel/hi6220-hikey.dtb-4.9
$ cp arch/arm64/boot/Image-dtb \
> ../../device/linaro/hikey-kernel/Image-dtb-4.9
```

3. Start ftrace using the “function” tracer and monitor its output. Careful: once you start ftrace by echoing “1” into tracing_on, it'll stay on until you stop it (i.e. echo “0” into same file).

4. Use ftrace to monitor the following:

- Scheduling change events
- CPU frequency scaling events
- Calls to __kmalloc
- Calls to the “brk” system call
- Calls to both __kmalloc and the “brk” system call
- All events occurring while the “system_server” process is scheduled
- All mcount calls occurring while the “system_server” process is scheduled

5. Use systrace/atrace to collect and view Android events

6. Add the circular buffer device driver located at http://www.opersys.com/downloads/circular-driver-111207.tar.bz2 to your board's BSP (i.e. device/linaro/hikey/) and build it into a .ko file. The AOSP used for the class had a stub .ko file in device/linaro/hikey/circular-driver/ directory, it's safe to replace it with the real driver. To built the driver, you'll first need to modify the Makefile included in the driver's directory to point to the kernel found in the kernel/hikey/ directory. To compile the driver, you will need to go to its directory and type a command such as -- note the “[AOSP]” which you'll have to replace with the path to your AOSP:

```
$ export TARGET=[AOSP]/prebuilts/gcc/linux-x86/aarch64/aarch64-linux-android-4.9/bin/aarch64-linux-androidkernel
$ make ARCH=arm64 CROSS_COMPILE=${TARGET}-
```

7. Driver testing:

- Use “adb push” to get the module copied on the device
- Use “insmod” to load the driver on the target
- The driver will appear as “/dev/circchar”
- Use “echo string > /dev/circchar” to write to the circular buffer
- Use “cat /dev/circchar” to read the content of the buffer

8. Now that you've tested the driver indepently, rebuild your AOSP with the new driver, reflash and reboot. The Opersys system service should be able to now use the driver directly. You can check the logcat to verify that it does.

9. Modify the driver to add a static tracepoint to monitor each read() and write() operation to it. Use trace_printk() to achieve that, remember that you'll have to use MODULE_LICENSE(“GPL”) in order to have access to this symbol. Rebuild your driver and reload it on the device. Start ftrace tracing and make sure you can see the output in the ftrace output at runtime when you do a “cat” or “echo” as above.

10. Replace your trace_printk() statements in your driver with custom-defined TRACE_EVENT() events and monitor those events with ftrace.

11. Instrument the native parts of the Opersys system service to log to ftrace's “trace_marker” file. Namely use the ATRACE_* macros to instrument (use “godir” to find the files):

- opersyshw_hikey.c
- Opersys.cpp
- com_android_server_OpersysService.cpp

12. Update your target and rerun it with ftrace enabled and check that the instrumentation added in the previous exercise is reflected in the traces generated. Make sure you review the slides on how to enable tracing for the AOSP parts.

13. Instrument the Java side of the system service, namely OpersysService.java, to log to ftrace's trace_marker file.

13. Update your system and check that you can use ftrace to follow all the tracepoints from the java layer, to the native layer, and into the driver.

14. Create a kprobe that catches all calls to the ioctl() system call and logs them into ftrace using trace_printk(). Build the kprobe as a driver, load it on your device and verify that you can see the output in ftrace's traces.

Using perf for performance analysis

1) Patch your kernel with the following patch to enable support for perf:
http://opersys.com/downloads/perf-hikey-4.9-181009.patch

2) New kernel with perf support – see slides and earlier exercises for details:
- Reconfigure your kernel to support perf,
- Rebuild the kernel
- Reinstall the kernel and the DTB at the proper location in the AOSP
- Rebuild the circular character driver
- Rebuild the AOSP from top level to take new files/images into account
- Reflash newly built AOSP to board

3) Check that simpleperf works as expected. The output of “simpleperf list” should show hardware events. Example:

```
hikey:/ # simpleperf list
List of hw-cache events:
  L1-dcache-loads
  L1-dcache-load-misses
  L1-dcache-stores
...

List of hardware events:
  cpu-cycles
  instructions
  cache-references
  cache-misses
  branch-instructions
...
```

4) Use simpleperf to generate statistics about the overall system's behavior for 10 seconds

5) Use simpleperf to generate statistics about the system_server's behavior for 10 seconds

6) Use simpleperf to count the number of events of each of the following type for 10 seconds of the overall system (separately):
- Context switches
- Branch instructions
- Cache misses
- Page faults
- Binder ioctl calls:
  - While system is idle
  - Running “service call” in parallel while simpleperf is recording events

7) Use simpleperf to record profiles for each of the following types of events for 10 seconds of the overall system (separately):

- Context switches
- Branch instructions
- Cache misses
- Page faults
- Binder ioctl calls:
  - While system is idle
  - Running “service call” in parallel while simpleperf is recording events

Use “simpleperf record” to record and “simpleperf report” to report back
NOTE: You will notice that some events are **NOT** working properly on the Hikey board.

BCC and eBPF

1) Install androdeb in your AOSP and then onto your device

2) Check that andordeb works by doing an: “adeb shell”

3) Exercise the following BCC commands within androdeb – you will likely need to shell into another session on the device using “adb shell” to run some commands in parallel to the BCC commands in order to see proper output:

- tplist
- execsnoop
- filetop
- opensnoop
- softirqs

For the following, refer to the BCC reference guide:
https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

4) Create your own BCC command based on bcc-master/examples/hello_world.py that attached to the sched_switch trace event using TRACEPOINT_PROBE instead of the “kprobes__” semantic.

5) Using “syncsnoop.py” as an example, create your own BCC command that snoops on Binder ioctl() calls: bindersnoop.py. Try recording and reporting the PID of the caller in addition to a timestamp.

6) Use the previously-coded command to monitor binder communication on the following commands:

- service list
- service call statusbar 1
- service call opersys 2 s16 foobar
- dumpsys

Additional tools

1) Use strace to monitor “service list”

2) Try the following Android tools:
- librank
- procrank
- showmap
- schedtest
- cpueater

3) Extend the Opersys system service to provide a dumpsys() function with at least one parameter: a string. Print the provided string as a parameter as part of the dumpsys() output.

4) Install and try Binder Explorer on your device:
- See https://github.com/opersys/binder-explorer-web/releases
- Put it in “external/”
- Use “mm” to build/install BE into your output images
- Reflash
- Use “OsysBE” to start BE
- You need to do an “adb forward tcp:3000 tcp:3000” on your host
- Connect to “localhost:3000/index.html” on the host to see the output

5) Create a board-specific dumpstate extension for the Hikey. Refer to:
- hardware/interfaces/dumpstate:
  - HIDL interface definition
  - Default implementation
- device/*:
  - Several devices have a dumpstate HAL
  - Pay attention to the manifest.xml entries for the dumpstate HALs