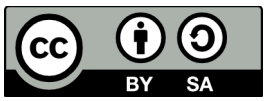
Hands-On Exercises for

Android Debugging and Performance Analysis

v. 2020.02

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 are made available to you under a Creative Commons Share-Alike 3.0 license. The full terms of this license are here:

https://creativecommons.org/licenses/by-sa/3.0/

Attribution requirements and misc.:

- This page must remain as-is in this specific location (page #2), everything else you are free to change; including the logo :-)
- Use of figures in other documents must feature the below “Originals at” URL immediately under that figure and the below copyright notice where appropriate.
- You are free to fill in the space in the below “Delivered and/or customized by” section as you see fit.

(C) Copyright 2010-2021, Opersys inc.

These exercises created by: Karim Yaghmour

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

---

Delivered and/or customized by:

Class Preparation Reference / Reminder

This section contains material that should have been used prior to class to prepare ahead of time. It's provided here as a reference/reminder.

Class Preparation for emulator

WARNING: The list of prerequisites for Android development changes over time

1. Install Android Studio:
https://developer.android.com/tools/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: http://source.android.com/source/initializing.html

3. Fetch the AOSP:
We'll be using 11.0.0 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 android-11.0.0_r27
$ cd android-11.0.0_r27
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-11.0.0_r27
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

4. Build the AOSP:

```
$ cd ~/android/android-11.0.0_r27
$ . build/envsetup.sh
$ lunch aosp_x86_64-eng
$ make -j8
```

5. Give the AOSP a spin:
From the same directory where you build the AOSP, run the emulator:

```
$ emulator &
```

At this point, an emulator should start with your custom-built Android running inside. It shouldn't take more than a minute or two to start.

AOSP sources and symbolic debugging

In this section you will learn how to:

- Import the AOSP as a project into Android Studio
- Use Studio to explore the AOSP
- Debug the AOSP's java code using Studio
- Use gdb/gdbserver to debug C/C++ code
- Step from a Java system service all the way down to a HAL module with Studio and gdb

1. Generate and run the idegen script:

```
$ cd ~/android/android-11.0.0_r27
$ . build/envsetup.sh
$ lunch 31
$ make idegen && development/tools/idegen/idegen.sh
```

This should result in your having a "android.ipr" file at the top level of your AOSP. Note that the build may complain about permission issues with some "root" directories. You can ignore this.

2. Add the Android's sources as a Java project by opening the "android.ipr" file as existing project in Android Studio. A few gotchas:

- You must make sure you have the SDK support for API 30
- Watch out for memory limit messages. You may need to increase the size of the heap attributed to Androi Studio. Click on the warning message to be taken to the proper menu.
- Watch out for filesystem watch limit. You may need to increase the watch limits by following the steps outlined in the link in the error message.
- Ignore messaging regarding git indexing.
- Accept converting the android.ipr to a new format if prompted.
- Do NOT accept migrating the project to Gradle if prompted.

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 the “gdbclient.py” on the host to step from the JNI side of the system service all the way down the driver call. In other words, 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. For Android 11, there is a problem with the default gdbclient provided in the AOSP and you will need to replace it for debugging to work:

```
$ cd prebuilts/misc/gdbserver/android-x86_64/
$ mv gdbserver64 gdbserver64-orig
$ wget https://opersys.com/downloads/gdbserver64-10.0.0_r39
$ mv  gdbserver64-10.0.0_r39  gdbserver64
$ chmod 755 gdbserver64
$ croot
$ cd out/target/product/generic_x86_64/
$ rm -rf system*
$ croot
$ make -j8
```

Once you restart your emulator, you should be able to use “gdbclient.py”:

```
$ gdbclient.py -n system_server
```

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.

Kernel Basics

In this section you will learn how to:

- Check what kernel version you are using
- Fetch, build and use a set of kernel sources “the Android way”
- Replace a default AOSP-provided kernel with a custom one

1. Check kernel version on the device (Shell into the device and type “cat /proc/version”). Your output should look something like this:

```
generic_x86_64:/ # cat /proc/version
Linux version 5.4.47-01061-g22e35a1de440 (android-build@abfarm-east4-070)
(Android (6443078 based on r383902) clang version 11.0.1
(https://android.googlesource.com/toolchain/llvm-project
b397f81060ce6d701042b782172ed13bee898b79), LLD 11.0.1
(/buildbot/tmp/tmp6_m7QH b397f81060ce6d701042b782172ed13bee898b79)) #1 SMP
PREEMPT Fri Jun 19 01:58:49 UTC 2020
```

This information is useful to know what version of the kernel you are relying on.

2. Download the kernel and associated toolchain:

a. Create directory for working on kernel – assuming this is outside your the top-level of the AOSP:

```
$ cd ~/android/
$ mkdir aosp11-kernel
$ cd aosp11-kernel
```

b. Get the kernel itself and its tools:

```
$ repo init -u https://android.googlesource.com/kernel/manifest \
> -b common-android11-5.4
$ repo sync
```

The final command will take a bit of time to download everything that you need. Once it’s done, you’ll then have a set of kernel sources and the tools to build them. More information about using this technique of fetching the kernel and the following instructions on building the kernel can be found here: https://source.android.com/setup/build/building-kernels

3. Make sure you have libncurses5-dev and libssl-dev installed:

```
$ sudo apt-get install libncurses5-dev libssl-dev
```

These packages are required for building the kernel. If you omit them then you will typically get errors at kernel build time.

4. By default, the kernel build script attempts to generate an LZ4-compressed kernel image at build time. If you are using Ubuntu 18.04 or earlier, the tool for generating this image is either

absent or won't work. Hence, we need to reconfigure the kernel to use BZ2 compression instead. To do so, open the "common/arch/x86/configs/gki_defconfig" file from the toplevel of the kernel repository and replace the LZ4 configuration. Here is what a "git diff" looks like on that file once it has been modified (changes in bold):

```
diff --git a/arch/x86/configs/gki_defconfig
b/arch/x86/configs/gki_defconfig
index 5f6b1e4dd7f3..5ec13a27f3f1 100644
--- a/arch/x86/configs/gki_defconfig
+++ b/arch/x86/configs/gki_defconfig
@@ -1,5 +1,5 @@
 CONFIG_UAPI_HEADER_TEST=y
-CONFIG_KERNEL_LZ4=y
+CONFIG_KERNEL_BZIP2=y
 # CONFIG_USELIB is not set
 CONFIG_AUDIT=y
 CONFIG_NO_HZ=y
```

"CONFIG_" variables are the means by which a kernel is configured.

5. Build the kernel and then the kernel modules required to boot the emulator. To do so, issue these commands at the toplevel of the kernel repository you downloaded earlier:

```
$ cd ~/android/aosp11-kernel/
$ SKIP_MRPROPER=1 BUILD_CONFIG=common/build.config.gki.x86_64 \
> build/build.sh
$ SKIP_MRPROPER=1 \
> BUILD_CONFIG=common-modules/virtual-device/build.config.goldfish.x86_64 \
> build/build.sh
```

The "build.sh" script is the primary build script for kernels and it takes parameters in the form of environment variables set prior to its execution. You can look at the script for additional options. Beware that the use of the options is not always clear and you may need to inspect the actual code to understand what's going on.

Once the build is over, you can check the "out/android11-5.4/dist/" directory for the output. It should contain a "bzImage" file, which is the kernel image, as well as many "*.ko" files, which are the kernel modules required for this kernel.

6. Replace the kernel and modules provided by default in the AOSP for the emulator with the kernel and modules you just built, and rebuild the AOSP:

```
$ cd ~/android/android-11.0.0_r27
$ cd prebuilts/qemu-kernel/x86_64/
$ mv 5.4 5.4-orig
$ mkdir -p 5.4/ko
$ cd ~/android
$ cp aosp11-kernel/out/android11-5.4/dist/bzImage \
> android-11.0.0_r27/prebuilts/qemu-kernel/x86_64/5.4/kernel-qemu2
$ cp aosp11-kernel/out/android11-5.4/dist/*.ko \
> android-11.0.0_r27/prebuilts/qemu-kernel/x86_64/5.4/ko
```

```
$ cd android-11.0.0_r27
$ . build/envsetup.sh
$ lunch aosp_x86_64-eng
$ rm out/target/product/generic_x86_64/kernel-ranchu
$ make -j8
```

The default emulator kernel and modules are provided under “prebuilts/qemu-kernel”. The commands above replace those files with the ones built earlier. For real devices, the AOSP generally stores kernel modules under a “device/VENDOR/DEVICE-kernel” directory. BSPs you get from silicon vendors might operate in a completely different way.

7. Restart the emulator and check the new kernel's version (“cat /proc/version”):

```
$ emulator -no-cache &
$ adb shell cat /proc/version
Linux version 5.4.61-android11-1-00003-g8d3d29f0729a-dirty (build-user@build-
host) (Android (6443078 based on r383902) clang version 11.0.1
(https://android.googlesource.com/toolchain/llvm-project
b397f81060ce6d701042b782172ed13bee898b79), LLD 11.0.1
(/buildbot/tmp/tmp6_m7QH b397f81060ce6d701042b782172ed13bee898b79)) #1 SMP
PREEMPT 2020-10-17 11:13:17
```

The “-no-cache” option of the emulator is very important because it’ll avoid using a cached previous emulator run. Notice that the version information printed out is different from the one you last checked it.

Linux Device Driver

In this section you will learn how to:

- Integrate a vendor driver to the Google Kernel repository
- Compile a vendor driver using Google's kernel build tools
- Modify the AOSP to take into account a new device in the native filesystem
- Integrate a driver with native Android user-space components for proper operation
- Check if a driver has been properly-loaded at boot time

NOTES:

- The exercises in this section assume that you have a kernel that was downloaded and built according to the "Kernel Basics" exercise section.
- The tarball we'll use for our exercise is here: http://www.opersys.com/downloads/vendor-circular-driver-201020.tar.bz2
- The driver we use in these exercises implements a circular buffer over the read/write file-operations. When read, if there is not data available, it returns 0 bytes.
- The driver is, however, simplistic. It doesn't implement blocking/waking semantics nor does it use proper locking primitives. It's ok for an exercise, but wouldn't be something you would ship to anyone.
- Also, the driver relies on misc_register() instead of register_chrdev() and will therefore register your char dev and fire off a hotplug event.
- For more information on device driver writing in Linux, have a look at the Linux Device Drivers book (dated as it may be): http://lwn.net/Kernel/LDD3/
- For reference, there's another driver sample here: http://tldp.org/LDP/lkmpg/2.6/html/x569.html

1. Download and extract the sample driver within the toplevel directory of the previously-downloaded kernel repository:

```
$ cd ~/android/aosp11-kernel/
$ wget http://www.opersys.com/downloads/vendor-circular-driver-210204.tar.bz2
$ tar xvjf vendor-circular-driver-210204.tar.bz2
```

Have a look at the content of the "vendor/" directory to see what's inside for building the driver. Note that Google has no known documentation at the time of this writing that explains how to build "vendor" modules alongside its GKI kernels. Hence, while the recipe here works, it's not necessarily the final way by which Google will want or instruct vendors to build their modules against GKI kernels.

2. Build the driver using Google's build script and copy it to the proper location in

```
$ SKIP_MRPROPER=1 BUILD_CONFIG=common/build.config.gki.x86_64 \
> EXT_MODULES="vendor/circular-driver" build/build.sh
$ cp out/android11-5.4/dist/circular-char.ko \
> ~/android/android-11.0.0_r27/prebuilts/qemu-kernel/x86_64/5.4/ko
```

The build process will generate a "circular-char.ko" under out/android11-5.4/dist/. This is the driver module. We need to copy it to the proper location under "prebuilts" in the AOSP in order for it to be: a) taken into account by the AOSP build system, and b) automatically loaded at start from vendor-image.

3. To have ueventd create the proper entry in /dev at runtime, you'll need to modify a ueventd.rc to have a proper entry for the module. You can modify the device/generic/goldfish/ueventd.ranchu.rc file to add:

```
/dev/circchar        0666  system     system
```

Without this line, the driver will be owned by the "root" user and will be inaccessible to a system service running as the "system" user.

4. To allow the device driver to be accessible by the system server, you'll need to modify the selinux policies to have a rule for the /dev entry. To do so, add "/dev/circchar" to the list of files allowed by the system server in the following file:
"device/generic/goldfish/sepolicy/common/file_contexts":

```
/dev/circchar          u:object_r:tty_device:s0
```

As before, this file contains the SELinux access rules for files in the filesystem. The rule we just added states that "/dev/circchar" belongs to the "tty_device" domain and will be accessible subject to the rules set for any process that has access to that domain.

5. Recompile your AOSP and restart your emulator (again, don't forget "-no-cache"). You should then shell into your device and see the module loaded and the device under /dev:

```
$ adb shell
generic_x86_64:/ # lsmod
Module                  Size  Used by
virtio_wifi            24576  0
virtio_pmem            16384  0
virtio_pci             28672  0
...
circular_char          16384  0
generic_x86_64:/ # dmesg | grep circ
[    0.663562] init: Loading module /lib/modules/circular-char.ko with args
[    0.664258] circular_char: loading out-of-tree module taints kernel.
[    0.704016] init: Loaded kernel module /lib/modules/circular-char.ko
generic_x86_64:/ # ls -al /dev/circchar
```

```
crw-rw-rw- 1 system system 10,  59 2020-10-19 08:10 /dev/circchar
```

Notice how the entry under /dev/ belongs to the “system” user and group.

6. You can now use “cat” and “echo” commands to read and write to your device:

```
generic_x86_64:/ # echo foobar > /dev/circchar
generic_x86_64:/ # cat /dev/circchar
foobar
generic_x86_64:/ # cat /dev/circchar
generic_x86_64:/ #
```

Notice how the 2nd “cat” command returns nothing. This is important because it means that the bytes have been “consumed”. The driver is also fairly verbose in the kernel logs. In a typical driver this would be wrong to do, but the exercise is meant to be academic:

```
generic_x86_64:/ # dmesg
[  272.584741] device_open called
[  272.586177] device_write called
[  272.587692] Writing f
[  272.588957] Writing o
[  272.590306] Writing o
[  272.591285] Writing b
[  272.592183] Writing a
[  272.592995] Writing r
[  272.593807] Writing \x0a
[  272.595215] device_release called
[  276.354275] device_open called
[  276.354555] device_read called
[  276.354794] Reading o
[  276.354989] Reading o
[  276.355158] Reading b
[  276.355328] Reading a
[  276.355497] Reading r
[  276.355666] Reading \x0a
[  276.355929] Reading \x00
[  276.356125] device_read called
[  276.356707] device_release called
[  282.449302] device_open called
[  282.449536] device_read called
[  282.449756] device_release called
```

Tracing with ftrace

In this section you will learn how to work with ftrace and Android's integration of it to get detailed information about the system's operation.

1. Use ftrace to monitor the following:
   - Scheduling change events
   - All scheduling-related events
   - All events occurring while the "system_server" process is scheduled

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

3. Modify the circular character 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.

4. 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.c
   - Opersys.cpp
   - com_android_server_OpersysService.cpp

5. 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.

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

7. 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.

8. Replace your trace_printk() statements in your driver with custom-defined

TRACE_EVENT() events and monitor those events with ftrace.

9. Create a kprobe module 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

NOT FOR EMULATOR

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:
- Installing “node.js” binary:
  - See https://github.com/fdgonthier/Aosp-Node-Prebuilts
  - Retrive Aosp-Node-Prebuilts into the “external/” directory
  - Build it for ARM64: “make arm64”
  - Install into the “system” image: “mm”
- Installing Binder Explorer:
  - See https://github.com/opersys/binder-explorer-web/releases
  - Put it in “external/”
  - Use “mm” to build/install BE into your output images
  - Reflash
- Operating BE:
  - 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