Hands-On Exercises for

Android Debugging and Performance Analysis

v. 2015.04

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/community/docs

Delivered and/or customized by:

Working with the AOSP sources

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)

Kernel tools and capabilities

1. Look for the system_server's memory usage in the /proc filesystem
2. Reconfigure the kernel to support the full range of tracers (function, function_graph, etc. CONFIG_*_TRACER). Make sure you enable CONFIG_DYNAMIC_FTRACE and disable CONFIG_STRICT_MEMORY_RWX.
3. Rebuild, reflash, reboot
4. 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).
5. Use perf-stat to measure standard counters for:
   - 60 seconds of the SurfaceFlinger execution while idle
   - 60 seconds of the system_server process while idle
6. Use perf-record/report/annotate to profile the same
7. Use ftrace's function profiling capability to profile the same
8. Compare the results of the previous 3 exercises
9. Try using the systrace/atrace combo to monitor Android
10. Use ftrace to capture from the command line the effects of the call to the collapse and expand of the status bar -- you may need to expand the ftrace buffer size:

```
# service call statusbar 1
# service call statusbar 2
```

11. User perf to monitor the statistics of the system_server process while you start the browser. Make sure you're monitoring just the system_server process.
12. Use perf to count just the number of task switches that occur when you start the calculator app
13. Use perf to record the execution of system_server while you open the gallery app
14. 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/qcom/msm8960/) and build it into a .ko file. The AOSP used for the class had a stub device/qcom/msm8960/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 out/target/product/msm8960/obj/KERNEL_OBJ/ directory. To compile the driver, you will need to go to its directory and type a command such as -- note the “...” which you'll have to replace with the path to your AOSP:

```
$ make ARCH=arm CROSS_COMPILE=/home/.../prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin/arm-eabi-
```

15. 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

16. 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.
17. 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.
18. 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.

Native debugging/profiling tools

1. Use strace to monitor the system calls made by the SurfaceFlinger during the startup of an app
2. Use strace to monitor the system_server
3. Use gdb on your host to step through a call to “service list”. You'll need to start the command with “gdbserver” and attach to it from the host.
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:
    - opersyshw_msm8960.c
    - 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. Use gdb 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.

Java tools

1. With the device connected to your host, start ddms and check out the threads run by the system_server and its heap information
2. Use traceview and dmtracedump to monitor the execution the system service added in the

previous section
3. Compare those results with the information collected by the ftrace instrumentation
4. Configure Eclipse to connect to ddms for your AOSP project imported earlier
5. With the system_server process selected in ddms, set a breakpoint in frameworks/base/services/core/java/com/android/server/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.
6. With the launcher process selected in ddms, set a breakpoint in packages/apps/Launcher2/src/com/android/launcher2/Launcher.java:showAllApps(). Click on the show all apps button on Android's home screen. Eclipse should now break into the debug view at the breakpoint you selected. You can now step through the Launcher's code for showing all apps.
7. Instrument the Java side of the system service, namely OpersysService.java, to log to ftrace's trace_marker file.
8. 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.
9. Full system service debugging:

- With the system_server process selected in ddms, 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.

Extra

1. Add a glibc-based filesystem to your AOSP by following the exercises for the Embedded Android class (see courseware at http://www.opersys.com/training/embedded-android)
2. Use EGL trace and/or apitrace to monitor the calls made by a couple of 3D benchmarks and their effects
3. Modify ftrace's functionality to record CPU counters on function entry/exit and report the results in the traces. Have a look at register_ftrace_function/unregister_ftrace_function.
4. Try getting the following to work with your AOSP:
   ktap
   Linaro's uprobes patch
   The BPF triggering patches

Some of the benchmarking tools available for Android, for reference:

- Apps:
  - 0xbench
  - AnTuTu
  - Passmark
  - Vellamo
  - Geekbench2
  - GLBenchmark
  - Quadrant Standard Edition
  - Linpack
  - Neocore
  - 3DMark
  - Epic Citadel
  - Androbench
  - CF-bench
  - SD Tools
  - RL Benchmark: SQL
  - Benchmark & Tunning
  - A1 SD Bench
  - Quick Benchmark Lite
  - 3DRating benchmark
  - Smartbench 2011
  - NenaMark
  - An3DBenchXL
  - CaffeineMark
  - NBench
  - AndEBench
  - SmartBench 2012
  - RealPi
  - Rowboperf
- Browser-based:
  - Rightware Browsermark
  - Octane
  - V8 browser perf
  - SunSpider
  - Methanol
- Native/CLI:
  - SPEC2000, SPEC2006
  - netperf

LMbench
microbench
cpueater