Hands-On Exercises for

Android Debugging and Performance Analysis

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.

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 Cuttlefish

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. Make sure you have the “repo” tool available:
See the “Installing Repo” here: https://source.android.com/setup/develop
You may need to log out and back in if the “~/bin” directory wasn’t already in your path.

4. Fetch the AOSP:
We'll be using 12.0.0 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-12.0.0_r2
$ cd android-12.0.0_r2
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-12.0.0_r2
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

5. Check that you got the right AOSP version:

```
$ cat .repo/manifests/default.xml | grep revision
  <default revision="refs/tags/android-12.0.0_r2"
```

Make sure it says “android-12.0.0_r2” in this output.

6. Download the Android GKI kernel:
Note that we’re just downloading the kernel for now, we aren’t building it.
a. Create directory for working on kernel – assuming this is outside your the top-level of the AOSP:

```
$ cd ~/android/
$ mkdir android12-5.10-kernel
$ cd android12-5.10-kernel
```

b. Get the kernel itself and its tools:

```
$ repo init -u https://android.googlesource.com/kernel/manifest \
> -b common-android12-5.10
```

```
        $ repo sync
```

7. Download and setup the Cuttlefish tools. Follow the instructions at the following URL to install the tools needed to control Cuttlefish instances:
https://android.googlesource.com/device/google/cuttlefish/

8. Build the AOSP:
Return to the AOSP directory and build it. We use "-j32" as 2X the number of CPUs (assumed 16 here as seen by typing "nproc" on the command line). Generally on a modern AOSP, you shouldn't need to specify "-jX", it'll be automatically figured out.

```
        $ cd ~/android/android-12.0.0_r2
        $ . build/envsetup.sh
        $ lunch aosp_cf_x86_64_phone-eng
        $ make -j32
```

You **MUST** check that the build completed successfully. Generally, you'll see this at the end of a successful build in the output (typically in green):

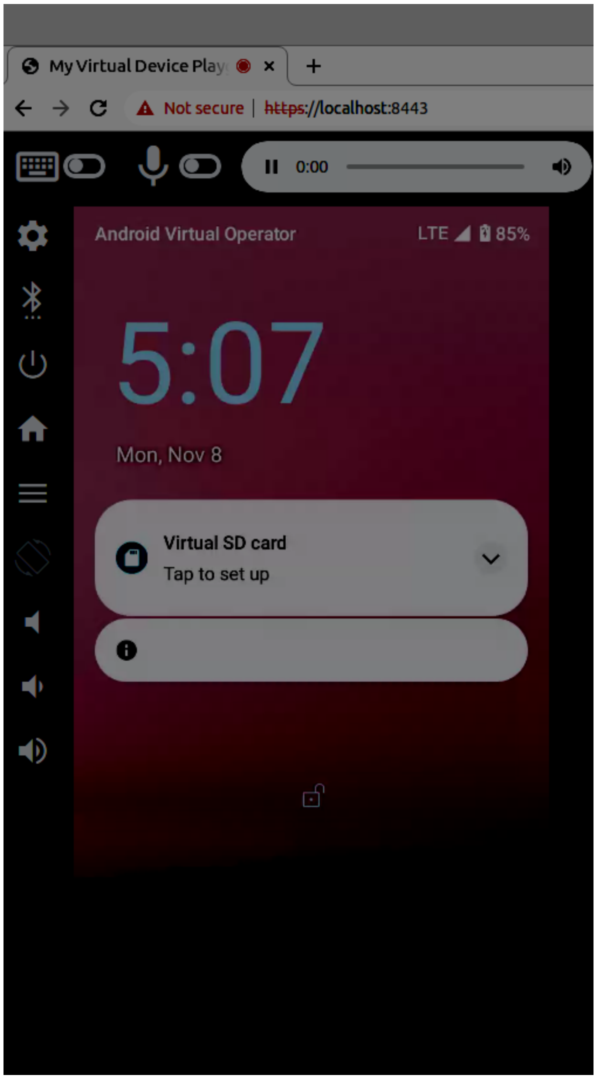
```
#### build completed successfully (...) ####
```

9. Give the AOSP a spin -- From the same directory where you build the AOSP, start the Cuttlefish target:

```
        $ HOME=${PWD}/out launch_cvd
```

At this point, the Cuttlefish target will start with your custom-built Android running inside and print messages to the command line[1]. However Cuttlefish itself won't be visible. You'll need to use a Chrome-based browser (Firefox will NOT work) and direct it to localhost:8443. You will likely need to accept some security exceptions to access the web page on which you'll see a device listed since it doesn't have a proper certificate. Once you click on the device you'll then see the Cuttlefish device interface.

[1] If you are seeing errors regarding "kvm" then see the next section.



You can stop Cuttlefish on the command line as well:

```
$ HOME=${PWD}/out stop_cvd
```

Extended Stack Addition to AOSP

For the purpose of the exercises we will do in class, you will need to add several components to create a new system service, a new HAL and a corresponding HAL module and driver. The details of these additions are covered in Opersys' Embedded Android class material (https://www.opersys.com/training/embedded-android-training/). The archive we use for the present class is tailored for the present exercise set. Hence, while you may want to refer to the Embedded Android class material for background, we suggest you continue referring to the present exercises for the hands-on portions of the present class.

1) Backup your AOSP's existing kernel (watch out for the differences between "x86_64" and "x86-64") and remove the original kernel build artifacts from the output:

```
$ cd ~/android/android-12.0.0_r2
$ cd kernel/prebuilts/5.10/
$ mv x86_64 x86_64-orig
$ mkdir x86_64
$ git init x86_64/
$ cp x86_64-orig/Android.bp x86_64/
$ mv x86_64-orig/Android.bp x86_64-orig/Android.bp-disable
$ cd ../common-modules/virtual-device/5.10
$ mv x86-64 x86-64-orig
$ mkdir x86-64
$ git init x86-64/
$ cp x86-64-orig/Android.bp x86-64/
$ mv x86-64-orig/Android.bp x86-64-orig/Android.bp-disable
$ cd ~/android/android-12.0.0_r2/out/target/product/vsoc_x86_64/
$ rm kernel
$ rm -rf \
>./obj/PACKAGING/depmod_VENDOR_RAMDISK_intermediates/lib/modules/0.0/
$ rm -rf obj/PACKAGING/depmod_vendor_ramdisk_stripped_intermediates/
```

2) Download and apply the Opersys patch for adding a system service, a HAL, corresponding HAL module and driver, and replacement kernel for Cuttlefish.

```
$ cd ~/android/
$ wget opersys.com/downloads/halext-opersys-cuttlefish-12.0-full-211123.tar.bz2
$ tar xvjf halext-opersys-cuttlefish-12.0-full-211123.tar.bz2
$ find halext-opersys-cuttlefish-12.0-full-211123 -exec touch {} \;
$ cp -r halext-opersys-cuttlefish-12.0-full-211123/* android-12.0.0_r2
```

3) Rebuild your AOSP:

```
$ cd ~/android/android-12.0.0_r2
$ . build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-eng
$ make -j32
```

4) Restart Cuttlefish to take into account the new additions (don't forget the "--noresume" parameter):

```
$ HOME=${PWD}/out launch_cvd --noresume
```

Cuttlefish should restart with no noticeable difference in the immediate.

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-12.0.0_r2
$ . build/envsetup.sh
$ lunch aosp_cp_x86_64_phone-eng
$ 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:

- Accept converting the android.ipr to a new format if prompted.
- Do NOT accept migrating the project to Gradle if prompted.
- Watch out for memory limit messages. You may need to increase the size of the heap attributed to Android 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.
- You must make sure you have the SDK support for API 31
- Ignore messaging regarding git indexing

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 (i.e. “system_process” as shown in the list) 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 “lldbclient.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 read_native() and write_native() and follow the calls all the way to the corresponding open(), read() and write() occurring in the opersyshw HAL.

```
$ lldbclient.py -n system_server
```

A few quick lldb commands, which resemble those of gdb by the way – you'll easily find many tutorials on the internet with a lot more details:

- cont: Continue running the program – you’ll need to do this after setting breakpoints
- b: Set a breakpoint, ex.: b com_android_server_OpersysService.cpp:75
  - This command supports TAB-completion for filenames
  - “:75” is the line number
- n: Step over next line
- s: Step into function
- p: Print variable
  - p var: prints “var”
  - p &var: prints dereferenced “var” -- useful is, say, “var” is a pointer.

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 lldb 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:

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

1. Download the kernel and associated toolchain (Presumably this was already done as part of class preparation):

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

    ```
    $ cd ~/android/
    $ mkdir android12-5.10-kernel
    $ cd android12-5.10-kernel
    ```

    b. Get the kernel itself and its tools:

    ```
    $ repo init -u https://android.googlesource.com/kernel/manifest \
    > -b common-android12-5.10
    $ 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

2. 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.

3. [THIS IS LIKELY NOT NEEDED WITH Ubuntu 20.04 and Android 12] 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.

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

```
$ cd ~/android/android12-5.10-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.virtual_device.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/android12-5.10/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.

5. Prepare the “kernel/prebuilts/” directory to copy your custom-built-kernel – this directory contains the default kernels and modules for Cuttlefish:

```
        $ cd ~/android/android-12.0.0_r2
        $ . build/envsetup.sh
        $ lunch aosp_cf_x86_64_phone-eng
        $ cd kernel/prebuilts/5.10/
        $ mv x86_64 x86_64-opersys
        $ mkdir x86_64
        $ git init x86_64/
        $ cp x86_64-opersys/Android.bp x86_64/
        $ mv x86_64-opersys/Android.bp x86_64-opersys/Android.bp-disable
        $ cd ../common-modules/virtual-device/5.10
        $ mv x86-64 x86-64-opersys
        $ mkdir x86-64
        $ git init x86-64/
        $ cp x86-64-opersys/Android.bp x86-64/
        $ mv x86-64-opersys/Android.bp x86-64-opersys/Android.bp-disable
        $ croot
```

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.

6. Clean up the output directory of the kernel built artifacts – you’ll need to do this every time you replace the kernel material under “kernel/prebuilts/”, otherwise your updates may not be taken into account (it’s not clear why such a drastic approach is required, but we haven’t found a way around this – if you find one, we’re happy to learn from you):

```
        $ cd out/target/product/vsoc_x86_64/
        $ rm kernel
        $ rm -rf \
        >./obj/PACKAGING/depmod_VENDOR_RAMDISK_intermediates/lib/modules/0.0/
        $ rm -rf obj/PACKAGING/depmod_vendor_ramdisk_stripped_intermediates/
        $ croot
```

7. Copy the kernel and modules you just built for Cuttlefish to your AOSP and rebuild it:

```
        $ cd ~/android
        $ cp android12-5.10-kernel/out/android12-5.10/dist/* \
        > android-12.0.0_r2/kernel/prebuilts/5.10/x86_64/
        $ cp android12-5.10-kernel/out/android12-5.10/dist/bzImage \
        > android-12.0.0_r2/kernel/prebuilts/5.10/x86_64/kernel-5.10
        $ cp android12-5.10-kernel/out/android12-5.10/dist/*.ko \
        > android-12.0.0_r2/kernel/prebuilts/common-modules/virtual-device/5.10/x86-64/
        $ cd android-12.0.0_r2
        $ make -j32
```

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

```
$ HOME=${PWD}/out launch_cvd --noresume &
$ adb shell cat /proc/version
Linux version 5.10.66-android12-9-00041-gfa9c9074531e (build-user@build-host)
(Android (7284624, based on r416183b) clang version 12.0.5
(https://android.googlesource.com/toolchain/llvm-project
c935d99d7cf2016289302412d708641d52d2f7ee), LLD 12.0.5
(/buildbot/src/android/llvm-toolchain/out/llvm-project/lld
c935d99d7cf2016289302412d708641d52d2f7ee)) #1 SMP PREEMPT Fri Nov 12 11:36:25
UTC 2021
```

The “--noresume” option of Cuttlefish is important because it’ll avoid using a cached previous Cuttlefish run.

Linux Device Driver

In this section you will learn how to:

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

NOTES:

❿ The exercises in this section assume that you have a kernel that was downloaded and built according to the “Kernel Basics” exercise section.
❿ The tarball we’ll use for our exercise is here:
http://www.opersys.com/downloads/vendor-circular-driver-211206.tar.bz2
❿ 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.
❿ 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.
❿ Also, the driver relies on misc_register() instead of register_chrdev() and will therefore register your char dev <u>and</u> fire off a hotplug event.
❿ 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/
❿ 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/android12-5.10-kernel
$ wget http://www.opersys.com/downloads/vendor-circular-driver-211206.tar.bz2
$ tar xvjf vendor-circular-driver-211206.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/android12-5.10/dist/circular-char.ko \
> ~/android/android-12.0.0_r2/kernel/prebuilts/common-modules/virtual-device/5.10/x86-64/
```

The build process will generate a “circular-char.ko” under out/android12-5.10/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/google/cuttlefish/shared/config/ueventd.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/google/cuttlefish/shared/sepolicy/vendor/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 Cuttlefish (again, don't forget "--noresume"). You should then shell into your device and see the module loaded and the device under /dev:

```
      $ adb shell
      vsoc_x86_64:/ # lsmod
      Module                  Size  Used by
      zram                   28672  0
      ...
      circular_char          16384  0
      ...
      vsoc_x86_64:/ # ls -al /dev/circchar
      crw-rw-rw- 1 system system 10, 123 2021-11-17 14:31 /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:

```
      vsoc_x86_64:/ # echo foobar > /dev/circchar
      vsoc_x86_64:/ # cat /dev/circchar
      foobar
      vsoc_x86_64:/ # cat /dev/circchar
      vsoc_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:

```
vsoc_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 from the command line to monitor the following:
   - ❿ Scheduling change events
   - ❿ All scheduling-related events
   - ❿ All events occurring while the “system_server” process is scheduled
   - ❿ Android HAL-related events
   - ❿ Android Graphics, View and Window Manager events simultaneously

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):
   - ❿ opersyshw.c
   - ❿ Opersys.cpp
   - ❿ 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 CUTTLEFISH

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.

Tracing with eBPF/BCC

In this section, you'll learn how to use Google's AndroDeb to get a chroot-hosted Debian distribution running inside Android in order to use to get access to the full range of eBPF/BCC tracing functionality.

1. How to use AndroDeb:

❿ Start by replacing the adeb directory under external/:

```
$ cd external
$ mv adeb adeb-orig                    (THIS MAY NOT BE NEEDED)
$ git clone https://github.com/joelagnel/adeb.git
$ cd adeb
```

❿ Download an AMD64 image for use with adeb. The image is at the following URL https://drive.google.com/file/d/1Dy2HKC5y11BNgZ01Z1kkM244DQWH7kfc/ and you must place it under external/adeb/. Then you can prepare it:

```
$ ./adeb prepare --archive amd64_androdeb.tar.gz
```

❿ Now mount the tracing directory under /sys to adeb's root filesystem – this will make the formal /sys/kernel/tracing directory from Android's root filesystem available in the Debian chroot jail:

```
$ adb shell mount --bind /sys/kernel/tracing/ \
> /data/androdeb/debian/sys/kernel/tracing
```

❿ Finally, you can shell into AndroDeb:

```
$ ./adeb shell
root@localhost:/#
```

We highly suggest you stop using "--noresume" from this point onwards as any changes you make in your AndroDeb environment will be destroyed if you do so.

2. Update and install missing packages for AndroDeb ("root@localhost:/#" omitted for brevity and ease of copy-pasting):

```
apt update
apt-get install arping bison clang-format cmake dh-python \
  dpkg-dev pkg-kde-tools ethtool flex inetutils-ping iperf \
  libbpf-dev libclang-dev libedit-dev libelf-dev \
  libfl-dev libzip-dev linux-libc-dev llvm-dev libluajit-5.1-dev \
  luajit python3-netaddr python3-pyroute2 python3-distutils python3
```

3. Clone, compile and install the BPF Compiler Collection from source:

```
cd
git clone https://github.com/iovisor/bcc.git
mkdir bcc/build; cd bcc/build
cmake ..
make -j4
make install
```

4. Fix BCC libraries to take into account Android 12's tracing filesystem transition from /sys/kernel/debug/tracing to /sys/kernel/tracing. Here, we're going to be using a "dirty" hack to replace the hard-coded paths in AndroDeb libraries with updated paths. WARNING: the extra "/" characters in the perl command have to be accounted for exactly to replace the number of characters in "debug". All of the following commands are done in the AndroDeb shell – the "root@localhost:/" part of the string is removed for brevity:

```
cd /usr/lib/x86_64-linux-gnu

      cp libbcc.so.0.23.0 libbcc.so.0.23.0-orig
```

```
perl -pe 's:kernel/debug:kernel//////:g' libbcc.so.0.23.0 > cheat.so
cp cheat.so libbcc.so.0.23.0
rm libbcc.so.0
ln -s libbcc.so.0.23.0 libbcc.so.0

cp libbcc_bpf.so.0.23.0 libbcc_bpf.so.0.23.0-orig
perl -pe 's:kernel/debug:kernel//////:g' libbcc_bpf.so.0.23.0 > cheat.so
cp cheat.so libbcc_bpf.so.0.23.0
rm libbcc_bpf.so.0
ln -s libbcc_bpf.so.0.23.0 libbcc_bpf.so.0
```

5. Fix the Python library paths as well:

```
cd /usr/lib/python2.7/dist-packages/bcc

cp __init__.py __init__.pyc-orig
perl -pe 's:kernel/debug:kernel/:g' __init__.py > __init__.py-fix1
perl -pe 's:../kprobes:../debug/kprobes:g' __init__.py-fix1 > __init__.py-fix2
cp __init__.py-fix2 __init__.py
```

6. Commit the changes to disk:

```
sync
```

Again, from this point onwards you’ll need to STOP using “--noresume”. Otherwise everything you did up to here in AndroDeb will be lost.

7. Mount the tracing and debugfs filesystems:

```
cd /
mount -t tracefs none /sys/kernel/tracing
mount -t debugfs none /sys/kernel/debug
```

8. The BCC tools are found under /usr/share/bcc/tools/. Some of them might need path fixing for “/sys/kernel/tracing” instead of “/sys/kernel/debug/tracing”. You’ll need to judge as you run them. But, there are some very insightful utilities available in BCC. We strongly recommend you spend some time exploring them. Here are a handful we recommend:

➓ execsnoop
➓ filetop
➓ opensnoop
➓ biosnoop (may need some fixing for paths)
➓ tplist (may need some fixing for paths)
➓ softirqs

9. Snooping on Binder communication:

➓ Download “bindersnoop.py” from opersys.com/downloads/bindersnoop.py
➓ Push bindersnoop.py to the ~ directory of AndroDeb
➓ Set bindersnoop.py as executable (chmod 755)
➓ Run bindersnoop to monitor Binder communication in Android

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

10. 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.

11. Using “syncsnoop.py” as an example, create your own BCC command that snoops on your driver’s read, write and ioctl commands. Try recording and reporting the PID of the caller in addition to a timestamp.

12. Use the previously-coded command to monitor what happens when you “cat” and “echo” to “/dev/circchar”.

Additional tools

1. Use strace to monitor “service list”

2. Try the following Android tools:
- ❿ librank
- ❿ procrank
- ❿ showmap
- ❿ schedtest
- ❿ 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. Create a board-specific dumpstate extension for Cuttlefish. Refer to:
- ❿ hardware/interfaces/dumpstate:
  - ⇲ HIDL interface definition
  - ⇲ Default implementation
- ❿ device/*:
  - ⇲ Several devices have a dumpstate HAL
  - ⇲ Pay attention to the manifest.xml entries for the dumpstate HALs

5. Create a board-specific atrace extension for Cuttlefish. Refer to:
- ❿ hardware/interfaces/atrace:
  - ⇲ HIDL interface definition
  - ⇲ Default implementation
- ❿ hardware/google/pixel/atrace:
  - ⇲ Example implementation
- ❿ Modify the opersyshw.c to rely on the board-specific custom tracing
- ❿ Use the “atrace” command to record your custom vendor events

6. Install and try Binder Explorer on your device:
- ❿ 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”
- ❿ 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
- ❿ 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