

Hands-On Exercises for

Android Debugging and Performance Analysis

v. 2025.04

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 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 2010-2025, 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 Cuttlefish

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

1. Install Android Studio:

<https://developer.android.com/studio>

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 15.0.0_r5. 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:

```
$ mkdir -p ~/android/android-15.0.0_r5
$ cd ~/android/android-15.0.0_r5
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-15.0.0_r5
$ repo sync
```

Note:

- Make sure the path depth is short. For reasons we do not control, the Cuttlefish emulator will refuse to start if the path length (in terms of characters) is too long. It's our understanding that Google is working on fixing this. In the mean time, make sure you keep the path relatively short in length. Generally our experience has been that it should be less than around 100 characters.
- 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-15.0.0_r5"
```

Make sure it says “android-15.0.0_r5” 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 android15-6.6-kernel
$ cd android15-6.6-kernel
```

b. Get the kernel itself and its tools:

```
$ repo init -u https://android.goglesource.com/kernel/manifest \
> -b common-android15-6.6
$ repo sync
```

7. Download and setup the Cuttlefish tools (Original instruction at <https://android.goglesource.com/device/google/cuttlefish/>):

Notes:

- Do NOT copy “>” symbols at the beginning of a line, they are added by the shell, you don’t type them in.
- The last instruction will reboot your computer (sudo reboot). Make sure that you can safely do that without losing any information.

```
$ cd ~/android/
$ sudo apt install -y \
> git devscripts config-package-dev debhelper-compat golang curl
$ git clone https://github.com/google/android-cuttlefish
$ cd android-cuttlefish
$ git checkout 17ccfffd06139cd69d3b4cd4f0179b43b20aa573
$ for dir in base frontend; do
  cd $dir
  debuild -i -us -uc -b -d
  cd ..
done
$ sudo dpkg -i ./cuttlefish-base_*_64.deb || sudo apt-get install -f
$ sudo dpkg -i ./cuttlefish-user_*_64.deb || sudo apt-get install -f
$ sudo usermod -aG kvm,cvdnetwork,render $USER
$ sudo reboot
```

8. Build the AOSP:

Return to the AOSP directory and build it. We don’t specify any “-j” flag since on a modern AOSP it’ll be automatically figured out.

```
$ cd ~/android/android-15.0.0_r5
```

```
$ . build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ make
```

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 (...) ####
```

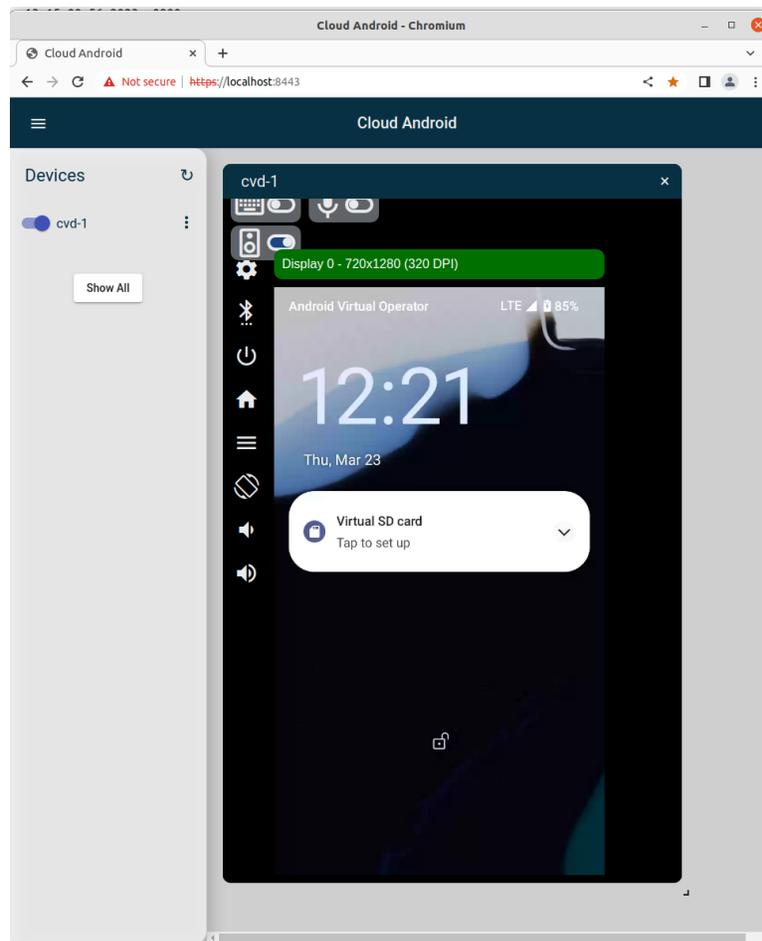
9. Give the AOSP a spin:

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 line¹. However Cuttlefish itself won't be visible. You'll need to use a Chrome-based browser (Firefox will NOT work) and direct it to <https://localhost:8443> – do not omit the “https” or it won't work. 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-15.0.0_r5
$ cd kernel/prebuilts/6.6/
$ mv x86_64 x86_64-orig
$ mkdir x86_64
$ git init x86_64/
$ cd ../common-modules/virtual-device/6.1
$ mv x86-64 x86-64-orig
$ mkdir x86-64
$ git init x86-64/
$ cd ~/android/android-15.0.0_r5/out/target/product/vsoc_x86_64/
$ rm kernel
```

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/halex-opersys-cuttlefish-15.0.0_r5-full-250401.tar.bz2
$ tar xvjf halex-opersys-cuttlefish-15.0.0_r5-full-250401.tar.bz2
$ find halex-opersys-cuttlefish-15.0.0_r5-full-250401 -exec touch {} \;
$ cp -r halex-opersys-cuttlefish-15.0.0_r5-full-250401/* android-15.0.0_r5
```

3) Rebuild your AOSP:

```
$ cd ~/android/android-15.0.0_r5
$ . build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-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 lldb to debug C/C++ code
- Step from a Java system service all the way down to a HAL module with Studio and lldb

1. Generate and run the idegen script:

```
$ cd ~/android/android-15.0.0_r5
$ . build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-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; go to Help->”Change Memory Settings” as an alternative. Set memory to 8192 in size. You'll need to restart Android Studio for changes to take effect
- 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. Check here for details: <https://youtrack.jetbrains.com/articles/IDEA-A-2/Inotify-Watches-Limit>. Use double the number on that page (i.e. 2*524288). You'll need to restart Android Studio for changes to take effect.
- You must make sure you have the SDK support for API 34. Check Tools->”SDK Manager”.
- Ignore messaging regarding git indexing
- Once indexing is done, if you aren't able to see processes when clicking on the debugging icon then you'll need to check that File->”Project Structure” has SDK 34 selected.

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. Android Studio 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 HAL AIDL side of our stack all the way down the driver call. The default one from Android 15.x doesn't work. Grab this one instead: <https://www.opersys.com/downloads/lldbclient-aosp-14-gsi.py> In other words, add a breakpoint to `read()` and `write()` in `opersys_cuttlefish.cpp` and follow the calls all the way to the corresponding `open()`, `read()` and `write()` occurring in the `opersyshw.c` legacy HAL.

```
$ ps -A opersys
system      4522  ... binder_ioctl_write_read 0 S android.hardware.opersys-service.cuttlefish
$ lldbclient.py -p 4522
```

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 opersys_cuttlefish.cpp:95`
 - This command supports TAB-completion for filenames
 - “`:95`” 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.

WARNING: The default one from Android 15.x doesn't work. Grab this one instead: <https://www.opersys.com/downloads/lldbclient-aosp-14-gsi.py>

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 HAL AIDL implementation : `read()` and `write()`
- 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:

```
vsoc_x86_64:/ # cat /proc/version
Linux version 6.6.30-android15-8-gdd9c02ccfe27-ab11987101 (kleaf@build-host)
(Android (11368308, +pgo, +bolt, +lto, +mlgo, based on r510928) clang version
18.0.0 (https://android.googlesource.com/toolchain/llvm-project
477610d4d0d988e69dbc3fae4fe86bfff3f07f2b5), LLD 18.0.0) #1 SMP PREEMPT Tue Jun
18 20:50:32 UTC 2024
```

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

2. 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 android15-6.6-kernel
$ cd android15-6.6-kernel
```

b. Get the kernel itself and its tools:

```
$ repo init -u https://android.googlesource.com/kernel/manifest \
> -b common-android15-6.6
$ 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. 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/android15-6.6-kernel
$ tools/bazel build //common:kernel_x86_64_dist
$ tools/bazel build //common-modules/virtual-device:virtual_device_x86_64_dist
```

What you are using here is the Kleaf kernel build system based on the Bazel build tools. Prior

to this, GKI kernels were built using the “build.sh” script which took parameters in the form of environment variables set prior to its execution. The script was really a bad fit for building the kernel. One of the issues is that the use of its options was not always clear and you sometimes needed to inspect the actual code to understand what’s going on.

5. Once the build is over, you can “install” the kernel and artifacts into “out/virtual_device_x86_64/dist/” by running the following command (notice the “run” instead of “build” as in the previous commands):

```
$ tools/bazel run //common:kernel_x86_64_dist
$ tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist
```

Once the build is over, you can check the “out/virtual_device_x86_64/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. Prepare the “kernel/prebuilts/” directory to copy your custom-built-kernel – this directory contains the default kernels and modules for Cuttlefish (mind the different uses of “-” (dash) vs. “_” (underscore), it’s very confusing ... sorry, not our fault ;)):

```
$ cd ~/android/android-15.0.0_r5
$ cd kernel/prebuilts/6.6/
$ mv x86_64 x86_64-opersys
$ mkdir x86_64
$ git init x86_64/
$ cd ../common-modules/virtual-device/6.6
$ mv x86-64 x86-64-opersys
$ mkdir x86-64
$ git init x86-64/
$ 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. Note the “useless” “git init”. If you don’t do that then the next time you run “repo” it’ll complain about the missing git repositories that have been moved around. Creating an empty git repository keeps “repo” happy.

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

```
$ cd ~/android
$ cp android15-6.6-kernel/out/kernel_x86_64/dist/* \
> android-15.0.0_r5/kernel/prebuilts/6.6/x86_64/
$ cp android15-6.6-kernel/out/kernel_x86_64/dist/bzImage \
> android-15.0.0_r5/kernel/prebuilts/6.6/x86_64/kernel-6.6
$ cp android15-6.6-kernel/out/kernel_x86_64/dist/*virtio* \
> android-15.0.0_r5/kernel/prebuilts/common-modules/virtual-device/6.6/x86-64
$ cp android15-6.6-kernel/out/virtual_device_x86_64/dist/*.ko \
> android-15.0.0_r5/kernel/prebuilts/common-modules/virtual-device/6.6/x86-64/
$ cp android15-6.6-kernel/out/virtual_device_x86_64/dist/initramfs.img \
> android-15.0.0_r5/kernel/prebuilts/common-modules/virtual-device/6.6/x86-64/
$ cd android-15.0.0_r5
$ 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 6.6.57-android15-8-maybe-dirty (kleaf@build-host) (Android
(11368308, +pgo, +bolt, +lto, +mlgo, based on r510928) clang version 18.0.0
(https://android.googlesource.com/toolchain/llvm-project
477610d4d0d988e69dbc3fae4fe86bff3f07f2b5), LLD 18.0.0) #1 SMP PREEMPT Thu Jan
1 00:00:00 UTC 1970
```

The “--noresume” option of Cuttlefish is important because it’ll avoid using a cached previous Cuttlefish 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-241218.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/android15-6.6-kernel
$ wget http://www.opersys.com/downloads/vendor-circular-driver-241218.tar.bz2
$ tar xvjf vendor-circular-driver-241218.tar.bz2
```

Have a look at the content of the "vendor/" directory to see what's inside for building the driver. Google has been refining their concept of Driver Development Kit (DDK) over the past few years. At present, the method used to build this "vendor" module alongside a GKI kernel seems to be the most up-to-date. Expect this recipe to evolve as Google continues to refine its DDK.

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

```
$ tools/bazel build //vendor:vendor_x86_64_dist
$ tools/bazel run //vendor:vendor_x86_64_dist
$ cp out/virtual_device_x86_64/dist/circular-char.ko \
> ~/android/android-15.0.0_r5/kernel/prebuilts/common-modules/virtual-device/6.6/x86-64/
```

The build process will generate a "circular-char.ko" under `out/virtual_device_x86_64/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. **[NOT NEEDED IF YOU FOLLOWED UP TO HERE]** 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. **[NOT NEEDED IF YOU FOLLOWED UP TO HERE]** 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:serial_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
...
[ 95.172494] circchar: circchar_open called
[ 95.172779] circchar: circchar_write called
[ 95.172942] circchar: Wrote f
```

```
[ 95.173099] circchar: Wrote o
[ 95.173251] circchar: Wrote o
[ 95.173403] circchar: Wrote b
[ 95.173684] circchar: Wrote a
[ 95.173878] circchar: Wrote r
[ 95.174030] circchar: Wrote

[ 95.174269] circchar: circchar_release called
[ 97.935331] circchar: circchar_open called
[ 97.935611] circchar: circchar_read called
[ 97.935760] circchar: Read f
[ 97.935916] circchar: Read o
[ 97.936080] circchar: Read o
[ 97.936271] circchar: Read b
[ 97.936446] circchar: Read a
[ 97.936621] circchar: Read r
[ 97.936812] circchar: Read

[ 97.937085] circchar: circchar_read called
[ 97.937290] circchar: circchar_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. Refer to slides 69-84 for details.

1. Use ftrace from the command line to monitor the following:

- Scheduling change events
- All scheduling-related events
- All events occurring while the "system_server" process is scheduled
- Android HAL-related events
- Android Graphics, View and Window Manager events simultaneously

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

3. Use ui.perfetto.dev to create a command to record Android traces and visualize the result.

Here's an example of how to manually collect info for Perfetto:

```
$ external/perfetto/tools/record_android_trace -o mytrace-250401-1 -t 30s gfx
view am wm sm hal sched irq
```

4. 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.

5. 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_cuttlefish.cpp
- Opersys.cpp
- com_android_server_OpersysService.cpp

You'll need to edit the relevant Android.bp files in the directories containing those files and ensure that they have the following libraries as dependencies: libcutils (for both C and C++) and libutils (for C++). For reference, to write/read from Opersys system service:

```
Write: # service call opersys 2 s16 foobar
Read: # service call opersys 1 i32 1234123
```

6. 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.

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

- Make the following modifications to the vendor/BUILD.bazel file in your kernel sources – the modifications are highlighted in grey:

```
...
filegroup(
    name = "vendor_circular_char_sources",
    srcs = [
        "circular-driver/circchar.h",
        "circular-driver/circular-char.c",
    ],
)

ddk_module(
    name = "vendor_x86_64_external_modules",
    srcs = [":vendor_circular_char_sources"],
    out = "circular-char.ko",
    kernel_build = "//common:kernel_x86_64",
    deps = [":common_headers"],
    includes = ["."],
)
...
```

- As a starting point for creating and using TRACE_EVENT(), look for existing TRACE_EVENT declarations in the kernel, such as:

```
common/kernel/sched/core.c
common/include/trace/events/sched.h
```

10. 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 set up 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. Create the base Debian system root directory structure (this is inspired from the full explanation here <https://wiki.debian.org/ChrootOnAndroid>) :

- Create a directory for your Debian work on Android:

```
$ mkdir ~/android/debian
$ cd ~/android/debian
```
- Create the Debian bootstrap filesystem:

```
$ sudo apt install debootstrap
$ sudo debootstrap --arch=amd64 -variant=minbase \
> stable ~/android/debian/rootfs http://deb.debian.org/debian
$ sudo tar -C ~/android/debian/rootfs \
> -cvzplf ~/android/debian/debian.tar.gz .
```
- Copy the kernel headers from the earlier GKI build:

```
$ cp ~/android/android15-6.6-kernel/out/kernel_x86_64/dist/kernel-headers.tar.gz .
```
- Get our script for running Debian:

```
$ wget https://opersys.com/downloads/run-debian
```
- Now push the content we generated and copied from elsewhere to Android:

```
$ adb push debian.tar.gz /data/local/tmp
$ adb push run-debian /data/local/tmp
$ adb push kernel-headers.tar.gz /data/local/tmp
```
- You can now shell into Android to continue the setup:

```
$ adb shell
vsoc_x86_64:/ # mkdir /data/debian
vsoc_x86_64:/ # cd /data/debian
vsoc_x86_64:/ # tar xvzf ../local/tmp/debian.tar.gz
vsoc_x86_64:/ # cp ../local/tmp/run-debian .
vsoc_x86_64:/ # chmod 755 run-debian
vsoc_x86_64:/ # tar xvzf ../local/tmp/kernel-headers.tar.gz
vsoc_x86_64:/ # ./run-debian
root@localhost:/# echo "nameserver 8.8.8.8" > /etc/resolv.conf
root@localhost:/# mkdir -p /data/local/tmp/
```

We highly suggest you stop using "--noresume" from this point onwards as any changes you make in your Debian 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 git dialog dh-python \
dpkg-dev pkg-kde-tools ethtool flex inetutils-ping iperf \
libbpf-dev libclang-dev libedit-dev libelf-dev libclang-cpp-dev \
libfl-dev libzip-dev linux-libc-dev llvm-dev liblua5.1-dev \
lua5.1 python3-netaddr python3-pyroute2 python3-distutils \
python3-setuptools python3 zip libpolly-14-dev linux-headers-generic \
python-is-python3 build-essential
```

You can also add whichever other Debian packages you want, including editors, etc.

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

```
cd
git clone https://github.com/iovisor/bcc.git
cd bcc
git checkout f954eb1ec60a34ddc59535646be58085163e568f
mkdir bcc/build; cd bcc/build
cmake ..
make -j4
make install
```

Note that we tried several commits before finding one that “works”. How? ... just trial and error :)

4. 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.

5. Mount the tracing and debugfs filesystems:

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

6. Set the kernel header location for BCC:

```
export BCC_KERNEL_SOURCE=/kernel-headers/
```

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:

- `execsnoop`
- `filetop`
- `opensnoop`
- `filegone`
- `tplist`
- `softirqs`

Some of these may fail because they’re trying to attach to kprobes that aren’t readily available on the kernel you are using. At the time of this writing, for example, with GKI 15-6.6 `execsnoop` and `opensnoop` don’t work out of the box. Both try to attach to system calls not available. The quick fix is to go to `/usr/share/bcc/tools/`, make a copy of the files (like copy `execsnoop` to `execsnoop2`) and edit those to make your fixes.

For example, here's a fix for execsnoop2:

```
...
#execve_fname = b.get_syscall_fname("execve")
execve_fname = "__x64_sys_execve"
...
```

And here's a fix for opensnoop2:

```
#fname_open = b.get_syscall_prefix().decode() + 'open'
#fname_openat = b.get_syscall_prefix().decode() + 'openat'
#fname_openat2 = b.get_syscall_prefix().decode() + 'openat2'
fname_open = 'do_sys_open'
fname_openat = '__x64_sys_openat'
fname_openat2 = 'do_sys_openat2'
```

Also, note that the scripts based on mcount-type source of kernel instrumentation will not work. Generally these will complain about "No such file or directory" for `/sys/kernel/debug/tracing/available_filter_functions`.

9. Snooping on Binder communication:

- Download "bindersnoop.py" from opsys.com/downloads/bindersnoop.py
- Push bindersnoop.py to the ~ directory of your Debian chroot directory
- Set bindersnoop.py as executable (chmod 755)
- 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 using the “cmd” command to check the command line capabilities of some system services.
3. Try the following Android tools:
 - librank
 - procrank
 - showmap
 - schedtest
4. 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.
5. Create a board-specific dumpstate extension for Cuttlefish. Refer to:
 - hardware/interfaces/dumpstate:
 - AIDL interface definition
 - Default implementation
 - device/*:
 - Several devices have a dumpstate HAL
 - Pay attention to the manifest.xml entries for the dumpstate HALs
6. Create a board-specific atrace extension for Cuttlefish. Refer to:
 - hardware/interfaces/atrace:
 - AIDL interface definition
 - Default implementation
 - hardware/google/pixel/atrace:
 - Example implementation
 - Modify the opersyshw.c to rely on the board-specific custom tracing
 - Use the “atrace” command to record your custom vendor events