PSCG

Building a General Purpose Android Workstation

Android Builders Summit
March 2015

Ron Munitz

Founder & CEO - The PSCG
ron@thepscg.com

@ronubo

PSCG

about://Ron Munitz

- **Founder and CEO of the PSCG**
    - **The Premium Embedded/Android consulting and Training firm**
- **Founder and (former) CTO of Nubo Software**
    - **The first Remote Android Workspace**
- **Instructor at NewCircle**
- **Senior Lecturer at Afeka College of Engineering**
- **Working on an awesome stealth startup**
- **Building up on diverse engineering experience:**
    - **Distributed Fault Tolerant Avionic Systems**
    - **Highly distributed video routers**
    - **Real Time, Embedded, Server bringups**
    - Operating Systems, very esoteric libraries, 0's, 1's and lots of them.
        - Linux, Android ,VxWorks, Windows, iOS, devices, BSPs, DSPs,...

Disclaimer

This talk will include ~~building~~/flashing[dd-ing]
/modifying/rebooting/testing/running.

- Some of those things may take some time.
- Some of those things may not be visible [depending on HDMI/External VGA on boot time)

Please do use that time for questions.

Agenda

- Demo [yes, that comes first]
  - TL;DR what makes Android an Android
  - End of day lecture -see if you want to leave or bring all your friends.
- Android and X86 - History.
  - Theory, what makes Android and Android
  - Challenges in transforming an Android into a GP platform
  - Approaches to porting complexities
- Break and laptop restart back to Linux
  - How stuff work - Code walkthrough.

PART I - Theory, Motivation

TL, (Do Read);

Theorem of Android Execution

∀ (Android Build Systems, boot methods, virtualization mechanisms, hardware, startup mechanisms) ,

∃ (A reasonable and straightforward explanation that makes them all behave quite the same)

*** I have a truly marvellous proof of this, which this margin is too narrow to contain - so from now and on everything is going to be really simple.*

What makes Android an Android

Let’s make things as simple as possible,
and then over-simplify them

(and then also explain in detail…..)

What makes Android an Android

- Over-simplified Android is:
  - ANDROID-ized kernel
  - Androidized /init (in some Androidized ramdisk)

- Without being too much of a smart@$$, this is the bare truth (@see next slide)

Justifying the over-simplification

- Mount *rootfs* with whatever modules you may need
  - can also just be the ramdisk
- Mount additional partitions with whatever you may need
  - */system* et. al (zygote, linkers, hals, binaries, pretty much everything.)

So in other words

- It really doesn’t matter what you do before you run init
- The important thing, is that you run init
- can also just be the ramdisk
- Mount additional partitions with whatever you may need
  - */system* et. al (zygote, linkers, hals, binaries, pretty much everything.)

PSCG

What comes next

We’re going to:

- Take the last couple of slides
- Explain them in greater detail
- Select a system that implements it
- Explain it
- Dissect it

Android Partition Layout

Android ROM components

Traditional terminology – whatever lies on the read-only partitions of the device's internal flash memory:

- Recovery Mode:
  - Recovery Image (kernel + initrd)
- Operational Mode:
  - Boot Image (kernel + initrd)
  - System Image
- The magical link between the two:
  - Misc

What is ***not*** a part of the ROM?

- User data: /data, /cache, /mnt/sdcard/...

Android ROM storage layout

Since Android is Linux at its core, we can examine its storage layout via common Linux tools:

```
shell@android:/ $ df
Filesystem              Size   Used   Free   Blksize
/dev                    487M    32K   487M   4096
/mnt/secure             487M     0K   487M   4096
/mnt/asec               487M     0K   487M   4096
/mnt/obb                487M     0K   487M   4096
/system                 639M   464M   174M   4096
/cache                  436M     7M   428M   4096
/data                     5G     2G     3G   4096
/mnt/shell/emulated       5G     2G     3G   4096
```

Android ROM storage layout - Standard Linux

PSCG

```
shell@android:/ $ mount
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
```

Output of **mount** continues in next slide

Android ROM storage layout: Standard Android

PSCG

none /acct cgroup rw,relatime,cpuacct 0 0

tmpfs /mnt/secure tmpfs rw,relatime,mode=700 0 0

tmpfs /mnt/asec tmpfs rw,relatime,mode=755,gid=1000 0 0

tmpfs /mnt/obb tmpfs rw,relatime,mode=755,gid=1000 0 0

none /dev/cpuctl cgroup rw,relatime,cpu 0 0

/dev/block/platform/sdhci-tegra.3/by-name/APP **/system ext4 ro**,relatime,user_xattr,acl,barrier=1, data=ordered 0 0

/dev/block/platform/sdhci-tegra.3/by-name/CAC /cache ext4 rw,nosuid,nodev,noatime,errors=panic, user_xattr,acl,barrier=1,nomblk_io_submit,data=ordered,discard 0 0

/dev/block/platform/sdhci-tegra.3/by-name/UDA /data ext4 rw,nosuid,nodev,noatime,errors=panic, user_xattr,acl,barrier=1,nomblk_io_submit,data=ordered,discard 0 0

/dev/fuse /mnt/shell/emulated fuse rw, nosuid, nodev, relatime,user_id=1023,group_id=1023, default_permissions,allow_other 0 0

Android ROM storage layout

```
shell@android:/ $ cat /proc/partitions
 major   minor  #blocks  name
 179        0    7467008 mmcblk0
 179        1      12288 mmcblk0p1
 179        2       8192 mmcblk0p2
 179        3     665600 mmcblk0p3
 179        4     453632 mmcblk0p4
 179        5        512 mmcblk0p5
 179        6      10240 mmcblk0p6
 179        7       5120 mmcblk0p7
 179        8        512 mmcblk0p8
 179        9    6302720 mmcblk0p9
```

Mapping blocks devices to ROM functionalities

PSCG

Some BSP's are kind enough to provide a mapping between the mapped partitions, and their purpose.
An example of an Nvidia Tegra based SoC follows:

```
shell@android:/ $ ls -l /dev/block/platform/sdhci-tegra.3/by-name/
lrwxrwxrwx root     root   2013-02-06 03:54 APP -> /dev/block/mmcblk0p3
lrwxrwxrwx root     root   2013-02-06 03:54 CAC -> /dev/block/mmcblk0p4
lrwxrwxrwx root     root   2013-02-06 03:54 LNX -> /dev/block/mmcblk0p2
lrwxrwxrwx root     root   2013-02-06 03:54 MDA -> /dev/block/mmcblk0p8
lrwxrwxrwx root     root   2013-02-06 03:54 MSC -> /dev/block/mmcblk0p5
lrwxrwxrwx root     root   2013-02-06 03:54 PER -> /dev/block/mmcblk0p7
lrwxrwxrwx root     root   2013-02-06 03:54 SOS -> /dev/block/mmcblk0p1
lrwxrwxrwx root     root   2013-02-06 03:54 UDA -> /dev/block/mmcblk0p9
lrwxrwxrwx root     root   2013-02-06 03:54 USP -> /dev/block/mmcblk0p6
```

Legend: APP is system, SOS is recovery, UDA is for data...

Why should we care about it?!

For a couple of reasons:

- Backup
- Recovery
- Software updates
- Error checking
- Board design
- Curiosity
- ...
- **Because up Android for a workstation is really just building an Android device!**

X86 Android Projects

PSCG

Android Projects

Various forks to the Android Open Source Project:

- **AOSP**
  - The root of all (good?)
- Android-X86
- Android-IA
- CyanogenMod
  - Need to raise funds? Ask them how…
- And many others. Not all are known or open sourced.

Android Projects

Since most workstations are running X86, we will concentrate on the X86 architecture (that includes 64 bit of course)

The same techniques can be easily applied to any ARM/MIPS/* based machine.

Android and X86

X86 ROMs (by chronological order):

- Android-X86 (Debut date: 2009)
  - http://android-x86.org
- Emulator-x86 (Debut date: 2011)
  - http://source.android.com
- Android-IA (Debut date: 2012)
  - https://01.org/android-ia

AOSP

The common reference, having the most recent version of the Android platform (Userspace) versions.

Provides the QEMU based ***Android Emulator***:

+ Works on any hosted OS

+ Supports multiple architectures

- But slow on non X86 ones

- Performs terribly if virtualized

  +/- Advances in nested virtualization help

- Has no installer for X86 devices

- Older kernel

+ Lollipop now provides a QEMU based target. This should help in porting

+/- An **emulator**. For better and for worse.

PSCG

Android-X86

+ Developed by the open source community
+ Developer/Linux user friendly
+ Multi-Boot friendly
+ Supports legacy boot and UEFI
+ Generally supports many Intel and AMD devices

+/- But of course requires specific work on specific HW

+ VM friendly
+ Mature, Recognized and stable

- Delays in new releases (You can help!)
  - Latest version (5.0) is still a bit buggy

  ? Any MESA developers here?

Android-IA

+ Installer to device
+ Relatively new versions of android and kernel
+ Works great on some Intel devices
- Development for devices based on intel solutions only
- Very unfriendly to other, non Windows 8/10/? OS's
- Not developer friendly – unless they make it such
- Community work can be better.
+ Unknown roadmap:
  + Made impressive progress in early 2013
  - But suspended development at Android 4.2.2 for months!
  + Back on track in April 2014
  - And then again - no Lollipop support for non MinnowBoard's.
  ? Any Intel OTC guys here?

Android is Linux

- Android is Linux
  - Therefore the required minimum to run it would be:
    - A Kernel
    - A filesystem
    - A ramdisk/initrd... Whatever makes you happy with your kernel's init/main.c's run_init_process() calls.
      See http://lxr.linux.no/linux+v3.6.9/init/main.c
  - This means that we can achieve full functionality with
    - A kernel (+ramdisk)
    - A rootfs where Android system/ will be mounted (ROM)
    - Some place to read/write data

Android IA is Android

Android-IA is, of course, Linux as well.
However, it was designed to conform to Android OEM's partition layout, and has no less than 9 partitions:

- boot - flashed boot.img (kernel+ramdisk.img)
- recovery - Recovery image
- misc - shared storage between boot and recovery
- system - flashed system.img - contents of the System partition
- cache - cache partition
- data - data partition
- install - Installation definition
- bootloader - A vfat partition containing android syslinux bootloader **(<4.2.2)**
  - A GPT partition containing gummiboot **(Only option in 4.2.2)**
- fastboot - fastboot protocol (flashed droidboot.img)

Note: Since android-ia-4.2.2-r1, the bootable live.img works with a single partition, enforcing EFI. It still has its issues - but it is getting there.

Android-IA boot process

- Start bootloader (e.g. EFI ***gummiboot***)
- The bootloader starts the combined kernel + ramdisk image (boot.img flashed to /boot)
- At the end of kernel initialization Android's
- /init runs from ramdisk
- File systems are mounted the Android way – using ***fstab.common*** that is processed (*mount_all* command*)* from in ***init.<target>.rc***

Android-X86 is Linux

- One partition with two directories
  - First directory – grub (bootloader)
  - Second directory – files of android (SRC)
    - kernel
    - initrd.img
    - ramdisk.img
  - system
  - data
- This simple structure makes it very easy to work and debug

Note: Also comes with a live CD/installer, and iso/efi bootable.

Very convenient.

Android-X86 boot process

- Start bootloader (GRUB)
- bootloader starts kernel + initrd (minimal linux) + kernel command line
- At the end of kernel initialization
  - run the ***/init*** script from initrd.img
  - load some modules, etc.
  - At the end ***change root*** to the *Android* file system
- Run the **/init** binary from ramdisk.img
  - Which parses init.rc, and starts talking "Android-ish"

We will examine Android-X86's /init after the break

Android-X86 vs. Android-IA : Which one is better?

PSCG

It depends what you need:

- Developer options?
- Debugging the init process?
- Support for Hardware?
- Support for OTA?
- Licensing?
- Participating in project direction?
- Upstream features?
- ...

There is no Black and White.

An hybrid approach

- Use Android-X86 installer system
- And put your desired android files (***matching*** kernel/ramdisk/system) in the same partition.
- Use the Android-X86 **chroot** mechanism
  - Critics: Does redundant stuff
  - But that's just a hack anyway – devise specific solutions for specific problems
- This way, we can multiple boot various projects:
  - Android-IA
  - AOSP
  - Any other OS...
    - On Multi-OS containers… See future talk.

Note: You can also use chroot mechanism on any Linux Distribution, from userspace! But this is *significantly* harder...

Booting your system

- Operating Systems do not start themselves.
  - Please don’t start a “what is an OS” discussion
- Kernels do not start themselves
  - Please don’t start a “what is a Kernel” discussion…
- Bootloaders, however do start them. Let’s see how:
  - With Legacy GRUB
    - Android-X86’s default.
    - Doesn’t support UEFI/GPT
  - With GRUB 2
    - “hopefully” your bootloader (it’s a Linux conference after all)
    - Supports whatever has to be supported.

Note: It’s really not about the bootloaders themselves. They are merely discussed as an example.

3 strikes for Android-IA

- You can’t really tell what is going on and when to expect stuff
- Too much proprietary stuff going on (more details later)
- The Lollipop version only supports MinnowBoard
  - With all the respect… that is very disappointing.

⇒ Rest of the talk will concentrate on Android-X86.

Android Multi-Booting

Legacy GRUB multi-boot recipe (simplified)

PSCG

Repartition existing Linux partition (Don't do that...)

Install Android-X86

Add entries to GRUB

Reboot to Android-X86 debug mode

Copy Android-IA files from a pendrive or over SCP

- For the former: cp /mnt/USB/A-IA/ /mnt && sync
- /mnt is the root of Android-X86 installed partition (e.g. (hd0,1)/...

Update GRUB entries and update GRUB

Voila :-)

Less simplified procedure: Debug GRUB... :-(

** **Note**: Replace *Android-IA* with *AOSP* to boot AOSP built files (system.img / kernel / ramdisk.img) on your target device.

Legacy GRUB multi-boot recipe (simplified)

PSCG

- Repartition existing Linux partition (Don't do that...)
- Create a mount point for your multi-booting android
  - Can make a partition per distribution, it doesn't really matter.
  - For this example let's assume all Android distributions will co exist on the same partition, and that it is mounted to /media/Android-x86
- Build your images
  - AOSP: Discussed before
  - Android-x86:
    - . build/envsetup.sh && lunch android_x86-<variant> \
      && make iso_img # OR make efi_img
  - ~~Android-IA: replace bigcore with ivy/sandy/who knows when intel will support bayrail et al...~~
    - ~~. build/envsetup.sh && lunch core_mesa-<variant> \
      && make allimages~~
    - ~~. build/envsetup.sh && lunch bigcore-<variant> && make allimages~~

** ***<variant>*** is either one of the following: *user, userdebug, eng*

Legacy GRUB multi-boot recipe (simplified)

PSCG

- Create directories for your projects (e.g. jb-x86, A-IA, AOSP) under your mount point (e.g. /media/Android-x86)
- From Android-X86's out/product/target: Copy ***initrd.img*** to all projects.
  - Can of course only copy ramdisk to one location.
- From all projects – copy ***kernel***, ***ramdisk.img***, ***system/*** and ***data/*** to to the corresponding directory under your mount point.
- Add entries to GRUB and update grub.
  - # e.g. sudo vi /etc/grub.d/40_custom && update-grub

Multi-boot recipe with GRUB2 - A “numerical” example

PSCG

```
$ df
Filesystem  1K-blocks      Used Available Use% Mounted on
/dev/sda5  451656948 394848292  34199920  93% /
udev          1954628         4   1954624   1% /dev
tmpfs          785388      1072    784316   1% /run
none             5120         0      5120   0% /run/lock
none          1963460      2628   1960832   1% /run/shm
/dev/sda1    15481360   5165416   9529464  36% /media/Android-
x86
```

A “numerical” example (cont.) - /etc/grub.d/40_custom

PSCG

```
#### JB-X86
menuentry 'jb-x86' --class ubuntu --class gnu-linux --class gnu --class os {
recordfail
insmod gzio
insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
echo    'Loading Android-X86'
linux /jb-x86/kernel quiet androidboot.hardware=android_x86 video=-16 SRC=/jb-x86
initrd /jb-x86/initrd.img
}
```

A “numerical” example (cont.) - /etc/grub.d/40_custom

PSCG

```
### android-IA
menuentry 'Android-IA' --class ubuntu --class gnu-linux --class gnu --class os {
recordfail
insmod gzio
insmod part_msdos
insmod ext2
set root='(hd0,msdos1)'
echo    'Loading Android-IA'
linux /A-IA/kernel console=ttyS0 pci=noearly console=tty0 loglevel=8 androidboot.hardware=ivb SRC=/A-IA
initrd /A-IA/initrd.img
}
```

Cloud bringup techniques

- The same technique would work also for bringing up Android on any cloud provider or VM.
- An example (taken from my 2014 sessions at the MWC and AnDevCon follows in the next slide)
  - Temporarily available in http://thepscg.com/talks was too short in time to provide a link, but if you stayed so far it might interest you...

PSCG

Using Android from within Linux

- A couple of excellent options for the non-virtualized Host (assuming Intel VT/AMD-V and the likes)
  - The AOSP X86 emulator/AOSP on a Virtual Machine
  - Android-X86 on a Virtual Machine
  - Android-IA on a Virtual Machine
- **Problem**: Can't run a VM within a VM!
- There are two elegant solutions for this problem…
  - Nested Virtualization
  - chroot-ing

PSCG

Android on AWS (teaser)

- **AWS Cloudroid recipe:**
  - Choose “Local” server with HW characteristics similar to the target VM
  - Bring up Android-X86 on XEN
    - You can use other distributions too for the chroot part
    - In fact - in many of my projects I use the AOSP
  - Create an AMI out of that Android-X86 instance
  - Set up a new AWS instance with that AMI
- Sounds simple, right?
  - Well, it’s not. If you are up for the challenge, I would definitely recommend hiring a top-notch, competent Linux bringup superstar.
- There is a **a bit** simpler alternative...

So, what is it good for?

- Assuming you have Android running on your device.
    -
- With Fully fledged command line tools
    - e.g. crosstool-ng to build gcc etc. etc.
- You can use Linux Containers / Hypervisors to multi-use OS’s:
    1. OS 1 / Display Protocol Server 1
    2. OS 2 / Display Protocol Server 2
    3. OS 3 / native apps for OS 3
    4. OS 3 / Display protocol clients for other OS’s...

PSCG

Releasing an Android from a chroot jail
in two quick steps:

1. Run “Standard” Linux
2. chroot <Android ramdisk.img> <Android’s /init>

That’s pretty much the same thing Android-X86 does on its init.

The problem here is who owns the display… On a server, it is actually very elegant and allows multi-Android instance scaling (If you have a display protocol…)

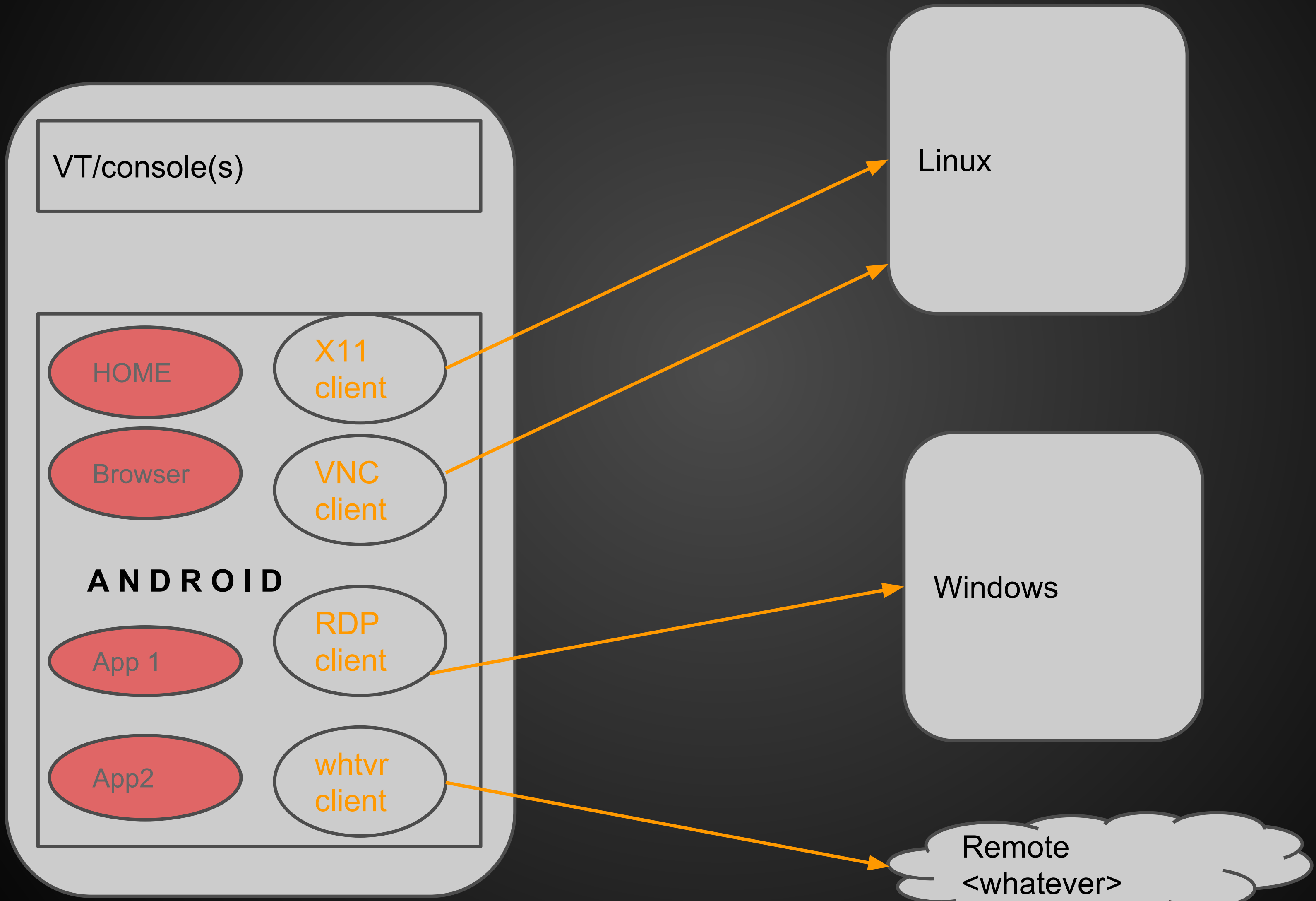
PSCG

Motivation

- Goal: Run Android anywhere. In particular - where a GPOS would usually run.
- Problem: How do you handle the porting complexities?
- Solution: Extend the work of others

PSCG

CVM (Collaborative VM)



PART II - Technical, Operation

Command Line Android/Linux Capabilities

Command Line Linux

- Android does not provide a terminal emulator as we know it.
- Neither does it provide binary compatibility due to ABI differences, libc differences
    - Can be worked around by recompiling
    - Or with providing glibc and LD_PRELOAD-ing
- In the next couple of slides we will see approaches to enabling our favorite terminal work.

Command line Linux approaches

- Via VT
  - Enable VT’s + key bindings
- Via terminal emulator apps
  - Modify framework to support split windows
    - There has been support in the AOSP since KitKat.
    - Modify code to run as a (bound) service:
      - Think of onPause(), onStop(),...
      - Think of Linux without nohup …
- VT primer follows
  - Can be skipped for ELC
  - Can be bullet-speed presented in ABS

Virtual Terminals

- A ***virtual console*** (VC), also known as a **virtual terminal** (VT), is a conceptual combination of the keyboard and display for a computer user interface. (Source: Wikipedia)
- Usually in Linux, the first 6 virtual consoles provide a text terminal with a login prompt. The ***graphical X Window*** System starts in the 7th virtual console. You can have up to ***63*** such terminals.
- But Android is not exactly linux. **There is No X!**
  - ***Surface Flinger*** is the graphic architecture.
  - No support for VT's in "vanilla" Android
  - Frame buffers are used as in Linux.
    - Depending on HW... In our case they are.

Virtual Terminals

- The keyboard shortcuts ***Alt+Fx*** and ***Crtl+Alt+Fx*** are implemented in kernel.
- Switching VT's using the keyboard shortcuts is supported upon explicitly setting permissions
- So the trivial solution would be just to call these ioctl's on the Surface flinger initialization service
  - @see frameworks/native/services/***surfaceflinger***/DisplayHardware/DisplayDevice.cpp
  - @see Android-X86 commit 640221175d9957b5d5bcddc83b4726a4da057cdd
  -

Virtual Terminals

- Well that is simple in Theory.
  - In real life, nothing works at first shot...
- **Problem**: simultaneous display of android applications and text messages from terminal.
- **Root Cause**: You have video driver for terminal works well and your graphic console is tty1.
- **Fix**:
  - Disable video driver of terminal (quite extremist...)
  - Use in another terminal for graphics (i.e. good old tty7

Virtual Terminals - legend

- Nodes:
  - ***/dev/tty*** – current terminal (like Xterm or virtual terminal)
  - ***/dev/tty0*** – current virtual terminal
- Commands:
  - ***openvt*** – open virtual terminal
  - ***chvt*** – switch to another virtual terminal
    - Do try this at home (sudo chvt 1 / chvt 7 in your Linux distro)

Now see Android-X86 /init script to see how virtual consoles are eventually set (mknod /dev/tty , openvt)

(@see bootable/newinstaller/initrd/init)

PSCG

Desktop /Laptop / Workstation Hardware Capabilities

Making HW/HAL’s work

Desktop alternatives

- Now that you have your Android, you need to be able to expose hardware to it.
- This can be done in the exact same way as in any other Android Bringup
  - e.g. overlays/HAL device registration/<have-feature> tags, etc.
- But it really doesn’t have to. Most of the components are already supported by the framework.
  - Except for the HAL’s...

Desktop/laptop BLOBs

- A reasonable set of Android-X86 supported hardware is listed below
  - Network (wireless, wire) - kernel + UCode firmware
  - Graphics
  - Audio
  - Extension cards via USB/PCI* (e.g. cellular modem)
  - Keyboard / Pointer device
  - More pointer devices [e.g. touchpad].
  - …
- Making some work is easy. Making others work is… Less easy.

PSCG

Desktop/laptop HW handling

- We'll start with the ones that are usually easier:
  - Extension cards via USB/PCI* (e.g. cellular modem)
    - Kernel responsibility to announce hotplug devices
    - add to ueventd if necessary
  - Keyboard / Pointer device
    - Enable relevant drivers in kernel, use the inputattach framework, add .kl if necessary
  - More pointer devices [e.g. touchpad].
    - Enable relevant drivers in kernel - add .idc if necessary

Desktop/laptop HW handling - Network

PSCG

- Network - Ethernet:
  - Configure your device in the kernel
    - e.g. CONFIG_E1000…
- Network - Wireless:
  - Configure your device in the kernel
  - And make sure the firmware for it is available under /lib/firmware/
    - e.g. /lib/firmware<your_blob>.ucode for the intel drivers

Graphical Android Capabilities

Desktop/laptop HW handling - Graphics

- Graphics
  - **Problem**. Hardware acceleration
  - “Solution”: Use MESA for the GPU hal
- Unfortunately MESA does not support all chipsets. “noveau” is traditionally experimental, “PowerVR” is not supported - so if you don’t have support from the manufacturers - you’re in problem
- Also, newer Open GLES versions may need a porting of MESA
- Best shot: Use i915/965 intel based chipsets.

Server/VM graphics

- Graphics:
  - **Problem**: Hardware acceleration
  - **Another Problem:** No GPU at all
  - **Another problem:** Graphics in a VM
- These are all really instances of the same problem.
- **Solution**: In the next slide.

Server VM/Graphics.

- If you want to use Graphic Acceleration - you must have the mechanisms.
- This is a very painful issue in virtualization
- To workaround it:
  - Implement drivers that support OpenGL ES 2.0/3.0 for Android.
  - Add offloading of OpenGL to host.
    - Not always possible.
    - See AOSP emulator and Genymotion
  - Add kernel flag 'qemu=1' and disable Hardware Acceleration, use software implementation instead

Android Emulator and Graphics

- As a GPU-less virtual machine, the Android emulator suffers the same problems, and offers two types of solutions:
  - Software Implementation (libGLES_android.so)
    - frameworks/native/opengl/libagl
  - Target → Host GL commands translation:
    - external/qemu/distrib/android-emugl (not part of the default manifest in Lollipop!)
  - @see frameworks/native/opengl/libs/EGL/Loader, @see Loader::load_driver(...)

PSCG

Androidizing your Android

Androidizing your Android

- Making your device “Google”
  - Seriously, WTH happened to goo.im ?!
  - TL;DR: get the right versions and push to the right places. (/system/app , system/priv-app/, … (
- Insisting to run ARM apps on on x86* arch
  - If the Android emulator can do it
  - **So can we.**
  - How?
  - **User mode qemu.**
  - Reference: Bluestacks, Intel’s libhoudini.

ARMing your Android

- FIrst of all: This should not really be relevant to almost anyone.
    - If someone doesn’t APP_ABI := <your arch> - they are probably not worth the install…
    - Yes, there are “legacy” excuses.
    - There are also HW excuses
    - **Bottom line is that this is not perfect.**
    - But if you still want to use it...

ARMing your Android - usermode QEMU

PSCG

- Theory of operation:
  - Whenever an .so is dlopen()-ed
  - If it has an arm* ABI - use **another dynamic loader**
  - The other dynamic loader is essentially a user-mode QEMU translator

ARMing your Android - usermode QEMU

- Challenges:
  - Modify the dynamic loader code (patch @libnativehelper, art, dalvik)
  - Get the QEMU user-mode blobs (.so’s).
    - And here comes the infamous proprietary blobs problem..

ARMing your Android

- Some companies have done tremendous work on integrating QEMU user-mode within X86 Android instances
- And have shipped devices that enable dual (or triple) architecture
- But have not open sourced it...

ARMing your Android

- Fighting against closed-source blobs:
  "If you can't win them - rip them":
    - Root the phone ["when there is a will - there is (usually) a way"]
    - Get the relevant .so's
    - **push them wherever they have to be pushed**
- This works for Android HAL blobs (@see about everything in XDA developers, @see how CyanogenMod "brunches")
- Works for Google's GLES pipe translator
- And this works for the user-mode qemu blobs as well.

ARMing your Android

- This works for Android HAL blobs
  - @see about everything in XDA developers
  - @see how CyanogenMod "brunches"
- This works for Google's Android Emulator Open GLES configurations (SW emulation/host translation)
  - Not ripped as it's open source. But definitely "pushed" using the build
- And this also works for the user-mode qemu blobs as well.

PSCG

Thank You

Questions/Consulting/Training requests:

ron@thepscg.com