1 Obtain a Linux-based operating system

A Linux-based system is an efficient and hassle-free environment for kernel development and Android application development. You have two options: use a Debian/Ubuntu distribution in a virtual machine on a Windows/OSX/Linux host or use a native Linux installation on your hardware or on one of CMU Linux clusters.

Throughout this guide (except in the following Option A section), we will refer to the Linux-based distribution where all development will happen as the *host* and to the Nexus 7 as the *target*.

1.1 Option A: Virtual Machine

1. Get a virtual machine with Ubuntu 12.04 (Precise): make your own, re-use one you have, or download a ready-made one from `http://timex.ece.cmu.edu/18648/precise.7z` (837 Mb)
2. Install latest Oracle Virtual Box from `http://virtualbox.org/`
3. Install VirtualBox Extension Pack from the same website (needed for USB device support)
4. Extract `precise.7z` (an archiver is available at `http://www.7-zip.org/` or in `p7zip-full` package if you're on Debian/Ubuntu but still chose to use a VM)
5. Launch VirtualBox, add `precise.vbox` using the `Machine > Add` menu, and configure it to match available resources on your hardware
6. To redirect the Nexus 7 USB device to the virtual machine:
   (a) Open virtual machine settings dialog and select the USB tab on the left
   (b) Enable the USB Controller and the EHCI Controller checkmarks
   (c) Plugin your Nexus 7 into a USB port on your computer
   (d) Add a *filter* by clicking the `Add filter from device` button on the right and choosing the device from the list
   (e) Edit the filter you just created: delete all fields except Name and Vendor ID. The Nexus 7 device has many faces: it presents itself as different devices depending on the mode (such as the bootloader mode) and we want the filter to catch all of them.
   (f) Unplug the Nexus 7, start the machine, and after the machine finishes booting, when you plug in the Nexus 7, the device should attach to the guest OS.
7. Login as `ubuntu` (no password), then switch to `root` (password `root`), and create a user for yourself. This guide assumes that you use your Andrew ID as the username. Also, add to `admin` group to allow gaining super user privileges with `sudo`.

```
$ su - root
enter root as the password
$ USERNAME=andrewid0
$ useradd -m -d /home/$USERNAME -s /bin/bash -G admin $USERNAME
$ passwd $USERNAME
set a password for yourself (required)
```

From now on you should not ever need to login as `root`, login as yourself and use the `sudo` utility to gain privileges when necessary. Note that when `sudo` asks for a password it is asking for *your* password, not `root` password, in order to verify that the right person is in front of the screen.

8. Customize the system to your taste! You'll spend lots of time in it. Among other things, install your favorite editor/IDE.

1.2 Option B: Native

Congratulations, you don't need to do any of the above. Also, enjoy significantly faster kernel compilations.

This guide was composed on Ubuntu 12.04 (Precise), but it should be applicable to any recent Debian-based distribution without significant modifications.

2 Setup shell environment

The rest of this guide assumes you have set the `HARCH` environment variable used to compose commands that work on both 32-bit and 64-bit hosts:

```
$ echo 'export HARCH=`echo $(uname -m) | sed "s/i./x/g"`' >> ~/.bashrc
$ . ~/.bashrc
```

3 Get essential tools

This guide assumes that your package cache is up to date and you have the following packages:

```
$ PACKAGES="wget tar unzip cpio gzip patch usbutils git"
$ if [ "$HARCH" = "x86_64" ]; then PACKAGES="$PACKAGES ia32-libs"; fi
$ sudo apt-get update
$ sudo apt-get install $PACKAGES
```

4 Device connectivity

4.1 USB connectivity

Verify the Nexus 7 device is recognized correctly by your host by checking that it shows up in the list of connected USB devices:

```
$ lsusb
```

Troubleshooting: if you are using a virtual machine, make sure the device shows up as attached to guest in the VirtualBox interface (`Devices > USB Devices > Google/Nexus 7 device` checkmark is set). If it doesn't, check your filter and try replugging the device into another USB port to give Windows a refresh request.

4.2 Permissions

To run the tools that access the Nexus 7 via USB as yourself (as opposed to as root), you need to grant yourself permissions to the USB device. We do this by adding a `udev` rule that matches to the vendor id of Google Inc. (reported by `lsusb`) and grants ownership to the current user:

```
$ RULE='SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", MODE="0666", OWNER="'`whoami`'"'
$ sudo sh -c "echo␣'$RULE'␣>␣/etc/udev/rules.d/50-nexus.rules"
$ sudo udevadm control --reload-rules
```

5 Clone repositories

5.1 Git

You will be using the Git *distributed* version control system to collaborate on your code and lab writeups as well as to submit your work. If you're familiar with Git, glance over the resources below, and skip to next section.

Git is an extremely powerful tool and, as a consequence, has a significant learning curve. Its principles differ fundamentally from *centralized* version control systems, such as SVN or CVS. Fortunately, you can get by with just a couple commands which you can learn in a few minutes from any of the many tutorials. However, to extract productivity gains from this tool, you will need to invest time into exploring it both before you begin (to understand the principles) and during your work (to expand your "bag-of-tricks").

Resources:

- Bare minimum interactive tutorial: `http://try.github.io/`
- Detailed tutorial: `http://git-scm.com/docs/gittutorial`
- Intro to Git for SVN users: `http://git-scm.com/course/svn.html`
- **Understanding Git Conceptually**: `http://www.sbf5.com/~cduan/technical/git/`

5.2 Create user identity

Each team's repositories, one for kernel code another one for writeups, will be hosted by a server at CMU. Each teammate needs to introduce him/herself to this server by giving it his/her public key.

The username to be associated with your key should be your Andrew ID to avoid collisions. The bot on the server pulls the username out of the *comment* in the key file (specified by the `-C` argument below)[1]. If you are re-using an existing key, please edit the comment in the `.pub` into the form *`andrewid@somehostname`*.

Login as yourself (not as `root`), generate a key pair, and send the public key to the server as follows:

```
$ sudo apt-get install openssh-client
$ ssh-keygen -C andrewid@somehostname
# enter a passphrase of your choice when prompted
$ ssh bot18648@timex.ece.cmu.edu add-student < ~/.ssh/id_rsa.pub
```

When prompted for the password for `bot18648` enter `botpleasehelpme`.

5.3 Fork/create and clone

Fortunately, you won't have to write a kernel for Nexus 7 from scratch. Instead, you will be modifying the official kernel source from the Git repository maintained by Google, whose clone is already on the class server ready to be *forked* into a clone private to your team. Note that the writeups repository starts empty (no forking) and is created automatically when you clone it.

Only ONE member from each team needs to do what's in this Section 5.3.

```
$ REPOPATH=18648/teamnumber (ex.team01)
$ REPOS="kernel␣writeups␣taskmon␣lunarlander"
$ ssh git@timex.ece.cmu.edu fork 18648/staff/kernel $REPOPATH/kernel
$ for repo in $REPOS; do git clone git@timex.ece.cmu.edu:$REPOPATH/$repo; done
```

So far only you, the creator, have access, so let your teammates in (note the line continuations (`>`)):

```
$ TEAMMATES="andrewid1 andrewid2"
$ for repo in $REPOS; do \
> for mate in $TEAMMATES; do \
> ssh git@timex.ece.cmu.edu perms $REPOPATH/$repo + TEAM $mate
> done; done
```

The cloned repositories are now in the two folders in the current directory: `kernel` and `writeups`.

Tell your teammates to set `$REPOPATH` and `REPOS` as above and clone the repos:

```
$ for repo in $REPOS; do git clone git@timex.ece.cmu.edu:$REPOPATH/$repo; done
```

5.3.1 Troubleshooting

To see which keys `ssh` is offering to the server and to print the public key corresponding to a private key:

```
$ ssh -v git@timex.ece.cmu.edu
$ ssh-keygen -y -f /path/to/private_key
```

[1] Impersonation attempts are detected by disallowing overwrites: if you succeeded in adding your key under your username, then nobody else can add their own key for that username. If your add failed due to an "aready-exists" error, then you know that somebody used your username, either accidentally or maliciously.

To list repositories to which you have permissions (a `C` permission next to a regular expression pattern means you can create a repository whose name matches that pattern):

```
$ ssh git@timex.ece.cmu.edu info -lc
```

Delete a repository you created (be careful!):

```
$ REPO=18648/teamnumber/reponame
$ ssh git@timex.ece.cmu.edu D unlock $REPO
$ ssh git@timex.ece.cmu.edu D trash $REPO
```

5.4 Submitting

As you work, please use your repository on the server to collaborate and as an extra backup copy. That is, push frequently, branch as needed. You own the repo.

To submit your work for (future) labs, simply push your commits to the master branch in the remote clone named `origin` (if this sounds cryptic, definitely see Section 5.1). The simplest workflow without any branching could be:

```
$ cd ~/kernel
# hack, hack, hack
$ git add somefile.c somefile.h
$ git status # did I add all my modified and new files?
$ git diff --staged # what am I about to commit?
$ git commit -m "A␣descriptive␣msg␣about␣the␣change"
$ git status # clean working copy, or other stuff to commit?
# repeat the above
# when ready to share your commits with your team:
$ git push origin master
```

We will grade the code/writeup from the latest commit timestamped before the deadline.

Feel free to complete the write ups in plain text, markdown, or in LaTeX. For the latter two, please also commit the compiled PDFs.

6 Install toolchain

6.1 Android SDK

To load the kernel images you compile onto the device, you'll need the `paltform-tools` that come with the Android SDK. The tools can be installed stand-alone, however since you will need to write an Andoid application in later labs, it is best to install the full ADT bundle now. If you'd like to customize your setup, please feel free to read about your options at `http://developer.android.com/sdk/`. Otherwise, this will do:

```
$ sudo apt-get install openjdk-6-jre openjdk-7-jre
$ cd # go home
$ wget http://dl.google.com/android/adt/adt-bundle-linux-$HARCH-20130729.zip
$ unzip adt-bundle-linux-$HARCH-20130729.zip
```

```
$ mv adt-bundle-linux-$HARCH-20130729 adt-bundle
```

Make the platform tools reachable from the command line:

```
$ echo 'export PATH=$PATH:$HOME/adt-bundle/sdk/platform-tools' >> ~/.bashrc
$ echo 'export PATH=$PATH:$HOME/adt-bundle/eclipse' >> ~/.bashrc
$ . ~/.bashrc
$ which fastboot adb eclipse # check that these are indeed found
```

6.2 Java development environment

The Eclipse-based ADT IDE that you just installed is the standard environment for writing Android applications. However, you could setup an existing installation of Eclipse, or a different IDE, or develop without an IDE – see `http://developer.android.com/sdk` for any of these options and more.

Check and get familiar with your setup:

1. Launch the ADT IDE by running `eclipse &`.
2. Create a Nexus 7 Virtual Device in `Window > Virtual Device Manager`.
   This will allow you to develop the Android app while your teammates are using the physical device.
3. Create a hello-world application using the wizard
4. Run the project – it will run in the virtual device you created

After you've taken control of the device (Section 9), return here and:

1. Make sure you've setup permissions to the USB device (Section 4.2)
2. Make sure USB debugging is enabled on the device (if `adb shell` works, then it is)
3. Plug in the physical Nexus 7
4. The IDE should detect the device and you should be able to find a way to run your test project on the physical device instead of on the virtual one.

6.3 Cross-compilation toolchain

To compile on one (host) architecture binaries for a different (target) architecture, you need a cross-compilation toolchain. A complete toolchain includes a compiler and a *sysroot*. The *sysroot* refers to the target system headers and core libraries (in particular, `libc`) installed on the host. To compile a userspace app (i.e. something that depends on `libc` and kernel headers) you need both a compiler and a sysroot. To compile a kernel, a sysroot is not needed.

In theory, the compiler and the sysroot can be setup separately and the compiler can be pointed to the sysroot when needed. However, in practice, some croos-compilers are not even built with sysroot support, and usually standalone toolchain packages are used which pack a cross compiler and one or more sysroots. In our particular case it is easiest to use a sysroot-less cross-compiler shipped in a Ubuntu package to compile our kernel and a toolchain shipped with Android NDK (Native Development Kit) to compile our userspace.

6.3.1 Kernel cross-compiler

Since Nexus 7 architecture is different from the host architecture, you'll need a cross-compiler. You will be using the cross-compilation toolchain from Linaro (`http://www.linaro.org/`). It is conveniently available from the package repository:

```
$ sudo apt-get update
$ sudo apt-get install gcc-arm-linux-gnueabi
# check:
$ arm-linux-gnueabi-gcc -v
# should be: 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) or later
```

6.3.2 Userspace cross-compiler (Android NDK)

```
$ cd
$ wget http://dl.google.com/android/ndk/android-ndk-r9-linux-$HARCH.tar.bz2
$ tar xf android-ndk-r9-linux-$HARCH.tar.bz2
```

We will use this to compile `busybox` in Section 8. You might also end up using this same installation of the NDK to call native apps from Java. You might have to point your Android SDK IDE to this NDK location when time comes.

7 Boot image packager

To be able to unpack, modify, and pack the ramdisk, you'll need some tools that are part of the Android source (we've made compiled versions available):

```
$ cd
$ wget http://timex.ece.cmu.edu/18648/bootimg-tools.tar.gz
$ tar xf bootimg-tools.tar.gz
$ echo 'export PATH=$HOME/bootimg-tools:$PATH' >> ~/.bashrc
$ . ~/.bashrc
# check that these are found
$ which mkbootfs mkbootimg unpackbootimg
```

8 Compile `busybox`

The `busybox` suite is a set of core userspace utilities commonly found on Linux-based systems, such as `find, grep, cut`. Without these, the Android shell is an austere place for a seasoned shell user. All utilities are packed into one binary.

1. Get the `busybox` source, the configuration file prepared by us (which defines which utilities are included), and a minor patch to handle Android differences over Linux:

```
$ cd
$ wget http://busybox.net/downloads/busybox-1.21.0.tar.bz2 --no-check-certificate
```

```
$ tar xf busybox-1.21.0.tar.bz2
$ cd busybox-1.21.0
$ wget -O .config http://timex.ece.cmu.edu/18648/busybox-android.config
$ wget http://timex.ece.cmu.edu/18648/busybox-android.patch
$ patch -b -p0 < busybox-android.patch
```

If you care to look at or customize the config: `make menuconfig`

2. Configure the compilation options for using the Android NDK toolchain and write them into the config file (the following script also available at `http://timex.ece.cmu.edu/18648/busybox-configure.bash`):

```
$ NDK=$HOME/android-ndk-r9
$ NDK_TC=arm-linux-androideabi-4.8 NDK_API=android-14 NDK_GCC=4.8
$ SYSROOT=$NDK/platforms/$NDK_API/arch-arm
$ LIBSTDCPP=$NDK/sources/cxx-stl/gnu-libstdc++
$ INCLUDES="-isystem␣$SYSROOT/usr/include␣-isystem␣$LIBSTDCPP/include␣"\
> "-isystem␣$LIBSTDCPP/$NDK_GCC/include␣"\
> "-isystem␣$LIBSTDCPP/$NDK_GCC/include/backward␣"\
> "-isystem␣$LIBSTDCPP/$NDK_GCC/libs/armeabi-v7a/include"
$ declare -A vars
$ vars[CROSS_COMPILER_PREFIX]=arm-linux-androideabi-
$ vars[SYSROOT]=$SYSROOT
$ vars[EXTRA_LDFLAGS]="-rdynamic␣--sysroot␣$SYSROOT␣-Wl,--gc-sections␣"\
> "-L$LIBSTDCPP/$NDK_GCC/libs/armeabi-v7a␣-lgnustl_static␣-lsupc++"
$ vars[EXTRA_CFLAGS]="-fsigned-char␣-march=armv7-a␣-mfloat-abi=softfp␣"\
> "-mfpu=vfp␣-fdata-sections␣-ffunction-sections␣-fexceptions␣-mthumb␣"\
> "-fPIC␣-Wno-psabi␣-DANDROID␣-D__ARM_ARCH_5__␣-D__ARM_ARCH_5T__␣"\
> "-D__ARM_ARCH_5E__␣-D__ARM_ARCH_5TE__␣-fomit-frame-pointer␣"\
> "--sysroot␣$SYSROOT␣$INCLUDES"
$ for var in CROSS_COMPILER_PREFIX SYSROOT EXTRA_LDFLAGS EXTRA_CFLAGS; do \
> sed -i "s|^CONFIG_$var=.*|CONFIG_$var=\"${vars[$var]}\"|" .config; done
$ export PATH=$NDK:$NDK/toolchains/$NDK_TC/prebuilt/linux-$HARCH/bin:$PATH
```

Please excuse the mess: we show you this in detail because in a future lab you will need to compile a userspace application and a library. When time comes you should be able to adapt the above to that task (keeping most of it unchanged). Please make sure to run this using a bash shell. Alternatively, you can copy and paste the commands into your shell.

3. Build and sanity check the architecture of the binary (should be ARM):

```
$ make
$ file busybox
```

4. Installing `busybox` involves copying the binary and creating the symbolic or hard *links* for the individual utilities packed into the binary. Via the links, you can invoke them directly by name, like `grep` instead of `busybox grep` – the utility is found by looking at `argv[0]`.

   Install links into `system` in the current directory:

```
$ make install
```

   Since the next steps happen on the target, they will have to wait until you get a root shell (Section 9.4). Once you have it:

- Option A: install to flash

```
# on the host:
$ tar cf busybox.tar system
$ adb push busybox /data/
$ adb push busybox.tar /data/
$ adb shell
# on the target:
$ mount -o remount,rw /system
$ cp /data/busybox /system/bin/
$ busybox tar xf /data/busybox.tar -C /
```

- Option B: install to ramdisk

  Adding `busybox` to ramdisk (as opposed to writing it to flash) will ensure that you always have it no matter what's currently on your flash (e.g. you'd still have it after a clean re-flash and a boot with your ramdisk). The steps are covered in Section 9.4.

9 Take control of Nexus 7

"To root" a device is a loose term used to describe several different operations. The entities in which privileges are often restricted are:

1. **bootloader**: a locked bootloader will let you boot only operating system images released by the vendor, but not ones you compile.
2. **shell**: a shell that you can get via the `adb` utility or via a terminal app by default will run as a restricted user
3. **user switching capability**: applications launched by a restricted can gain privileges if a "switch user" capability (e.g. via `su`) is available to them

While gaining the third kind of capability is of interest to most casual users, for your work it is unnecessary and a root shell will suffice, which is covered later in this section.

9.1 Enable USB debugging

1. Go to `Settings > About tablet`
2. Touch the `Build number` several times until you get a message `You are now a developer!`[2]
3. Go one level back and click the newly appeared `Developer options` and enable `USB debugging`

[2] Don't ask... At least they didn't have us pronounce a secret chant into the microphone.

9.2 Meet the platform tools

9.2.1 `adb`

The `adb` (Android Debug Bridge) utility talks to the device over USB via a custom protocol. Its featureset is includes that of SSH (shell and file transfer), an application-level debugger for Android, and some commands to hardware. In Section 9.1 you've enabled the `adb` backend on the target and can now try out some useful commands:

```
$ adb help
$ adb devices
$ adb push /host/path/to/file /target/path/to/file # try /data on target
$ adb shell
$ adb reboot-bootloader
```

9.2.2 `fastboot`

The `fastboot` utility is your interface to the bootloader (the entity responsible for loading the kernel image into memory and passing control to it). You will use it to boot the kernel images you compile.

To reboot into bootloader you could either:

- issue `adb reboot-bootloader`, or
- while holding the Volume Down button, press and hold the power button (as you would normally to turn it on)

The bootloader screen should appear: a black screen with the Android mascot.

As expected, fastboot can overwrite the kernel image in the flash storage. However, its more convenient feature is `fastboot boot`, which takes an image from the host and boots it directly from memory, i.e. without persisting it in flash. This is faster than going through flash memory and lets you keep the working original kernel untouched on the flash, so that the device is always at most a reboot away from being usable.

```
$ fastboot help
$ fastboot devices
```

Note: if fastboot devices returns `no permissions`, this means our efforts in Section 4.2 did not succeed. Check the rule file. If the `udev` rule file takes effect, then the device node corresponding to the Nexus 7 should have `admin` as the owner group:

```
$ find /dev/bus/usb -exec ls -l {} \;
```

9.3 Unlock bootloader

The bootloader on your device **might** be already unlocked: check the lock icon on the bottom of earliest bootup screen to determine whether it's been unlocked. If it is unlocked, skip this Section 9.3. Otherwise:

1. Reboot the device into bootloader (see Section 9.2.2)

2. Connect the device to a USB port, check that fastboot sees the device and unlock it:

```
$ fastboot devices
$ fastboot oem unlock
```

9.4 Obtain a root shell

The shell you get from `adb shell` runs as a restricted user by default. However, it can be configured to be run as root. The configuration is via low-level Android system-wide properties.

The set of properties of interest is loaded from a file on the *initial ramdisk*. The ramdisk (a standard Linux feature) is an in-memory root file system used during early boot before the real on-disk root file system is mounted. A ramdisk (`initrd`) is usually seen in company with the kernel binary (`vmlinux`).

While you're at it, you'll also add a property to disable the (uncalled for) need for WiFi and activation on first bootup. Otherwise after a clean flash, believe-it-or-not, you'd need a WiFi Internet connection to even check the version of the currently running kernel.

1. Get the factory image, unpack the ramdisk archive from the boot partition image, and extract it into a file tree:

```
$ cd
$ wget https://dl.google.com/dl/android/aosp/nakasi-jdq39-factory-c317339e.tgz
$ tar xf nakasi-jdq39-factory-c317339e.tgz
$ cd nakasi-jdq39
$ unzip image-nakasi-jdq39.zip
$ mkdir boot-img
$ unpackbootimg -i boot.img -o boot-img
$ cd boot-img
$ gunzip boot.img-ramdisk.gz
$ mkdir ramdisk
$ cd ramdisk
$ cpio -i < ../boot.img-ramdisk
```

2. Edit the properties of interest:

   - `ro.secure=0`: start `adb` as root
   - `ro.debuggable=1`: allow *restarting* `adb` on-demand via the `adb root` command
   - `ro.adb.secure=0`: disable public-key authentication of the host (gets rid of popup on first `adb` connection)
   - `ro.setupwizard.mode=DISABLED`: go directly to home screen without WiFi and activation

```
$ sed -i 's/ro.secure=1/ro.secure=0/g' default.prop
$ sed -i 's/ro.debuggable=0/ro.debuggable=1/g' default.prop
$ sed -i 's/ro.adb.secure=1/ro.adb.secure=0/g' default.prop
$ echo 'ro.setupwizard.mode=DISABLED' >> default.prop
```

3. **Optional** (see Section 8): install busybox to ramdisk.

```
$ BBOX=~/busybox-1.21.0/system
$ cp -a $BBOX/bin/* sbin/
$ cd sbin
$ find $BBOX/sbin -type l -printf "%f\n" | xargs -n 1 ln -s busybox
$ cd ..
```

Incidentally, if you want to add any other binaries/scripts that you would frequently use to the ramdisk, then simply copy them to `sbin/` as above.

4. Pack everything back up:

```
$ cd ..
$ mkbootfs ramdisk | gzip > boot.img-ramdisk-root.gz
```

You do **not** need to glue the kernel binary and the ramdisk together (as they were) because `fastboot` can take them separately, but you could:

```
$ mkbootimg -o ../boot-root.img \
> --kernel boot.img-zImage --ramdisk boot.img-ramdisk-root.gz \
> --base `cat boot.img-base` --cmdline boot.img-cmdline
```

5. To test your ramdisk, reboot Nexus 7 into bootloader (see Section 9.2.2), connect it to USB, and boot the stock kernel with your ramdisk as follows:

```
$ fastboot devices # make sure it's there
$ fastboot boot boot.img-zImage boot.img-ramdisk-root.gz
```

Wait for the device to boot directly into home screen and enable `adb` backend on the target:

(a) Go to `Settings > About tablet`

(b) Touch the "Build number" several times until you get a message "You are now a developer!"[3].

(c) Go one level back and click the newly appeared "Developer options" and enable "USB debugging"

Enter the root shell and run a busybox utility:

```
$ adb devices # make sure the device is there
$ adb shell
# on the target:
$ whoami
# better be 'root'
```

10 Compile and boot kernel

Extract the kernel config from the running stock kernel and cross-compile your own kernel:

[3]Don't ask... At least they didn't have us pronounce a secret chant into the microphone

```
$ cd ~/kernel # this is the repository you cloned earlier
$ adb pull /proc/config.gz
$ gunzip config.gz
$ mv config .config
$ export CROSS_COMPILE=arm-linux-gnueabi- ARCH=arm
$ make oldconfig
$ make
$ file arch/arm/boot/compressed/vmlinux
```

Check that the reported type of file is an ARM executable.

For subsequent builds you need only make sure `CROSS_COMPILE` and `ARCH` environment variables are set as above and run `make`. If at any time you want to start from scratch: `make clean`. You shouldn't need to but to see or change the kernel config: `make menuconfig`.

Connect the Nexus 7, reboot it into the bootloader (see Section 9.2.2), and boot the kernel you compiled earlier with the ramdisk you packaged earlier:

```
$ RAMDISK=~/nakasi-jdq39/boot-img/boot.img-ramdisk-root.gz
$ fastboot boot arch/arm/boot/zImage $RAMDISK
$ adb shell
# on the target:
$ uname -a
```

The commit hash and the date should reflect your kernel build.

11 Reflashing to factory state

You should **not** need this, but in case something goes very wrong and you want to get your device into a completely clean state, you can reflash it with the factory image. Do this only if you have a *valid* reason to: most of your issues will be about fixing your kernel code.

Get the factory image if you don't already have it from earlier:

```
$ cd ..
$ wget https://dl.google.com/dl/android/aosp/nakasi-jdq39-factory-c317339e.tgz
$ tar xf nakasi-jdq39-factory-c317339e.tgz
```

Boot into the bootloader (see Section 9.2.2) and:

```
$ cd ~/nakasi-jdq39
$ fastboot -w update image-nakasi-jdq39.zip
```

Do not touch the bootloader under any circumstances! That's how the device can be hard bricked with no return (no, not even JTAG). This means, **do NOT run the `flash-all` scripts!**

12 Userspace notes

12.0.1 Kernel version

To check that you are running the kernel that you think you are running check the version, git commit hash, and the date in `Settings > About tablet > Kernel version`. Same can be obtained in a root shell (assuming you've got busybox installed):

```
$ adb shell
# on the target
$ uname -a
```

12.0.2 No updates

Do not update! The kernel source tree you will be working on is based on top of the (unmodified) kernel from Android 4.2.2 (JDQ39) and can be guaranteed to be compatible only with that version of userspace. If you update the system, then most likely it won't boot on top of your own kernel.

In case of an accidental update, re-flash by following Section 11.

If you find a *reliable* and *elegant* way to disable update checks and prompts, please do let us know.

12.0.3 CMU WiFi

Connect to CMU-SECURE with Phase 2 auth MSCHAPV2 and your Andrew ID and password as the Identity.

12.0.4 Google account

Given that you disabled the Google-activation wizard in Section 9.4, it should be possible to get through the course without registering a Google account! However, many applications that you might be tempted to use (most notably, Google Play) require an account. Feel free to register a dummy one or use your own.

13 Finish

Congrats on setting up your development environment – good luck developing!