

# all-kernels

---

May 2011

How to compile several kernel versions and run a herd of Qemus

Alessandro Rubini ([rubini@gnuudd.com](mailto:rubini@gnuudd.com))

---

## Introduction

This simple package is a set of scripts that I wrote for myself, to have several kernel versions running at the same time, to test my sample code on several versions – such code doesn't need real hardware to run.

Please note that this stuff may not work for you (for example, I may invoke some of my programs from the scripts, or some dependencies may be undocumented). Similarly, my choices may be suboptimal just out of ignorance – I can't know everything and on some topics I'm pretty naive. Please give me your feedback if you use this stuff, I'm willing to fix any issues.

There are three main scripts, one to build the kernels, one to build the filesystem and fire a herd of *qemu* or *kvm* processes.

## 1 Prerequisites

In order to run this package you'll need the C language tool-chain, the Debian build programs and *texinfo* to compile the documentation.

Additionally, one script uses `wrap-tty`, which is in source form in the *tools* subdirectory of this package. Please compile it and place it in your command search path.

With regards to system configuration, you need *ext2* available on the host system, as well as *bridge* and *tun/tap*. Also, you need to have an NFS server in order to share you home with the *qemu* instances. See below for details.

## 2 Package Outline

This stuff was born just for myself, so I made it as simple as possible; I used abstractions only when they made my work easier, so some parts are generic or scalable, and most are not.

The basic idea is using environment variables for configuration and avoiding the download or recompile steps as much as possible. However, the result may be slightly buggy with regard to dependencies or such stuff.

The user is expected to set personal environment variables and then run the suggested ones. Thus, you should run the `.` (*dot*) shell command.

```
. suggested_env
```

Each environment variable in there is documented, so please read *suggested\_env* itself. You may also edit the script to activate your own defaults instead of mine, or make your own copy. The script doesn't force any variable, it only sets them if they are not already set.

All relevant environment variables have a name starting in *AK\_*, for *all-kernels*. Thus you can look at all of them with

```
env | grep '^AK_'
```

If you need to unset the variables to start afresh (for example, I do it while checking this package), you can run

```
unset $(env | grep AK | awk -F= '{print $1}')
```

Besides *suggested\_env*, there other files related to configuration, related to the three main steps; each is documented within the specific chapter.

By default, all compiled stuff is placed in `../all-kernels-build` (the kernels and their log files) and `../all-kernels-debootstrap` (the boot filesystem).

The default configuration compiles all official kernel versions from 2.6.20 to 2.6.38; it uses around 29 gigabytes of disk space and takes several hours to complete, even on a not-too-old multi-core system.

Currently I'm only running this natively, I haven't tested cross compilation yet, but I'd love to run the herd for ARM-Linux.

## 3 Compiling the Kernels

To compile all the kernels (according to the variable `AK_KLIST`), you should just run:

```
./scripts/do_compilekernels
```

As you'll notice, all main scripts are called `do_something`.

Please note that the `AK_KLIST` variable is usually made up of integer numbers (that are suffixed to "2.6.") but non-integer strings are allowed as well. For example, you can compile and run an RC kernel if you want (however, only one at a time with the current scripts, but I'm only interested in testing the latest one).

`do_compilekernels` uses *git* to checkout the kernel locally, so some description of how *git* is used is mandatory, before you start compiling.

### 3.1 Using Git and Alternates

The scripts in this package assume that a local copy of the kernel source is already available in the filesystem. The environment variable `AK_GIT_MASTER` must name the directory of your *master* repository, which is expected to include the tags for each revision you are compiling. If your master repository is not *bare*, the variable name should include the trailing `.git` directory name.

Since we need to check out the repository several times, one per version, I use the *alternate* technique, to share *git* objects among versions. To this aim I write `$AK_GIT_MASTER/objects` into `.git/objects/info/alternates` of each new repository.

After setting *alternates*, you need to build local information about the branches (even if the objects remain remote), and to this aim a *fetch* operation is needed. Such a *fetch* takes several minutes, so I allow to do it once of all.

If you create a directory called `master-template` within your `$AK_BASE_DIR` (the main directory of the project), the script copies over this directory and just checks out the files, instead of repeating the *fetch* from `AK_GIT_MASTER`.

To create you `master-template` you can use the following commands, from within the base directory:

```
mkdir master-template
cd master-template
git init
echo $AK_GIT_MASTER/objects > .git/objects/info/alternates
git fetch $AK_GIT_MASTER master:master
```

After this operation, the `.git` directory is very small (4 MB mor or less), as all real data remains in the master repository. The script will thus copy over these 4MB instead of repeating the *fetch* for each kernel version.

### 3.2 Actual Kernel Compilation

This section explains the way I set up kernel compilation, to help you understand what is going on under the hood, in case you need to open it and make your modifications.

The main script (`do_compilekernels`) checks the presence of a few mandatory environment variables, and then builds each version running `scripts/compileone`. If the file `vmlinux` is already present in the build directory for a specific version, no compilation is started at all. You can thus

force recompilation of specific versions if for example you changes some configuration parameters, but removing `vmlinux`.

This `compileone` script receives a full kernel version as first command line argument (e.g.: `2.6.35`). It creates its build directory and starts a git checkout, using either `master-template` or `AK_GIT_MASTER`. It then performs the following steps, where `2.6.35` is used as example version:

- It checks out `v2.6.35` from the repository.
- It creates a new branch `v2.6.35-localconfig`.
- It applies the default configuration, and commits it.
- It customizes the default configuration, and commits it.
- It applies a few patches.
- It compiles saving a log file
- It reports the compilation results and exits successfully.

The successful exit is something I used to check which versions compile and which do not in my own system in order to help me identifying the correct fix for each problem.

### 3.3 Example Run

This is what you might expect when running the script:

```
morgana$ scripts/do_compilekernels
Variable check:      AK_BASEDIR = /op/all-kernels
Variable check:      AK_KLIST = 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
Variable check:      AK_CONFIG = /opt/all-kernels/customconfig
Variable check:      AK_GIT_MASTER = /opt/linux-2.6-git
Variable check:      AK_PATCH_DIR = /opt/all-kernels/patches
Variable check:      AK_BUILD_DIR = /opt/all-kernels/./all-kernels-build
You may now ctrl-C for 3 seconds...
Sat Mar 12 20:18:31 CET 2011: Processing 2.6.20 (/opt/all-kernels-build/2.6.20)
Sat Mar 12 20:18:31 CET 2011: Copying template git...
Sat Mar 12 20:18:31 CET 2011: Checkout...
Switched to branch "v2.6.20-localconfig"
Sat Mar 12 20:22:09 CET 2011: Defconfig...
Sat Mar 12 20:23:18 CET 2011: Applying time-prevent-the-loop-in-timespec_add_ns
Sat Mar 12 20:23:18 CET 2011: Compile...
Sat Mar 12 20:30:00 CET 2011: Done (success)
Sat Mar 12 20:30:00 CET 2011: Processing 2.6.21 (/opt/all-kernels-build/2.6.21)
Sat Mar 12 20:30:00 CET 2011: Copying template git...
[...]
Sun Mar 13 01:09:21 CET 2011: Checkout...
Switched to branch "v2.6.37-localconfig"
Sun Mar 13 01:14:13 CET 2011: Defconfig...
Sun Mar 13 01:15:13 CET 2011: Compile...
Sun Mar 13 01:29:02 CET 2011: Done (success)
```

Your build directory, after the process is over, will contain the following files:

```
drwxrwxr-x 21 rubini staff  4096 Mar 12 20:29 2.6.20/
-rw-rw-r--  1 rubini staff 45228 Mar 12 20:30 2.6.20-make.log
-rw-rw-r--  1 rubini staff 102785 Mar 12 20:30 2.6.20.log
drwxrwxr-x 21 rubini staff  4096 Mar 12 20:42 2.6.21/
-rw-rw-r--  1 rubini staff  46622 Mar 12 20:42 2.6.21-make.log
-rw-rw-r--  1 rubini staff 105566 Mar 12 20:42 2.6.21.log
[...]
drwxrwxr-x 25 rubini staff  4096 Mar 13 01:08 2.6.36/
-rw-rw-r--  1 rubini staff  78037 Mar 13 01:08 2.6.36-make.log
-rw-rw-r--  1 rubini staff  78378 Mar 13 01:08 2.6.36.log
drwxrwxr-x 25 rubini staff  4096 Mar 13 01:28 2.6.37/
-rw-rw-r--  1 rubini staff  79055 Mar 13 01:29 2.6.37-make.log
-rw-rw-r--  1 rubini staff  79396 Mar 13 01:29 2.6.37.log
```

### 3.4 Configuring the Kernel Build

The build is configured in two ways: by selecting configuration changes, to be applied after *make defconfig*, and by selecting patches.

The configuration changes are applied by the `$AK_CONFIG` script. The default one is called `customconfig` and sets some variables that are not set by default in all version I'm interested in. In particular, it enables the *ext2* filesystem and the *8139cp* network driver. The script relies on functions defined in `scripts/functions`, in case you need to look deeper in it.

Code patches are applied by the script `scripts/applypatch`, which looks in the directory `$AK_PATCH_DIR`, where a `SERIES` file is expected to exist (a working version of this stuff is in the *patches* directory of this package, which is selected by default). What *applypatch* does is applying specific patches (according to `SERIES`) before or after a specific kernel version – so the same patch is not applied for all versions, only for versions before or after a specific point in kernel history.

## 4 Building the Filesystem

Before running the herd of emus, we need a filesystem. You can use any filesystem of your choice, although I've only used Debian or some home-made hacks. In the released code, the filesystem is expected to be an *ext2* image, but you can always change the way *qemu* or *kvm* is fired.

Thus, the next script in this set is used to build a Debian minimal filesystem. Even though *squeeze* is already released, I stick with *lenny* for the time being.

To run the script with the default parameters, just run

```
./scripts/do_debootstrap
```

### 4.1 Actual Debootstrap Run

Running *debootstrap* is pretty straightforward. The script is more or less self-explicative (see the comments), and supports the idea of an overlay (so you can add your own binaries in the filesystem).

Currently the size of the filesystem is hardwired to 200MB (`AK_FSIZE` would be good to have, but it isn't there, yet).

After running *debootstrap*, the script applies some extra scripts, to customize the filesystem.

### 4.2 Example Run

This is a typical run of `./scripts/do_debootstrap`. The first time you invoke it you'll get more output because the packages are downloaded as well – in the example shown no download is performed:

```
Variable check:      AK_BASEDIR = /opt/all-kernels
Variable check:      AK_DB_DIR = /opt/all-kernels/./all-kernels-debootstrap
Variable check:      AK_DEBS = strace,tcpdump,vim-tiny,net-tools,nfs-common,
                        udev,openssh-server,openssh-client,
                        iputils-ping,module-init-tools
Variable check:      AK_FSNAME = target-fs
Variable check:      AK_DB_OPTS = lenny target-fs http://ftp.it.debian.org/debian
Variable check:      AK_DB_FIXUP = /opt/all-kernels/fs-fixup
Variable check:      AK_DB_OVERLAY = /opt/all-kernels-overlay
You may now ctrl-C for 3 seconds...
Wed May 11 09:14:25 CEST 2011: Running debootstrap
I: Retrieving Release
I: Retrieving Packages
I: Validating Packages
```

```

I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Found additional base dependencies: adduser debian-archive-keyring
  gnupg gpgv libbz2-1.0 libedit2 libevent1 libgcrypt11 libgnutls26
  libgpg-error0 libgssglue1 libkeyutils1 libkrb53 libldap-2.4-2
  libnfsidmap2 libpcap0.8 libreadline5 librpcsecgss3 libsasl2-2
  libssl10.9.8 libtasn1-3 libusb-0.1-4 libvolume-id0 libwrap0 netbase
  openssh-blacklist portmap readline-common ucf vim-common
I: Checking component main on http://ftp.it.debian.org/debian...
I: Validating libacl1
I: Validating adduser
[...]
I: Validating vim-tiny
I: Validating zlib1g
I: Extracting libacl1...
I: Extracting libattr1...
[...]
I: Extracting zlib1g...
I: Installing core packages...
I: Unpacking required packages...
I: Unpacking libacl1...
I: Unpacking libattr1...
[...]
I: Unpacking util-linux...
I: Unpacking zlib1g...
I: Configuring required packages...
I: Configuring sysv-rc...
[...]
I: Configuring nfs-common...
I: Base system installed successfully.
Wed May 11 09:16:01 CEST 2011: Creating ext2 filesystem
[...]
Wed May 11 09:16:34 CEST 2011: Applying fix fix-activate-getty-on-ttyS0
Wed May 11 09:16:35 CEST 2011: Applying fix fix-no-rename-netif
Wed May 11 09:16:35 CEST 2011: Applying fix fix-ssh-authorize
Wed May 11 09:16:35 CEST 2011: Applying fix fix-mount-home
Wed May 11 09:16:35 CEST 2011: Applying fix fix-hostname
Wed May 11 09:16:36 CEST 2011: Filesystem is ready in /opt/all-kernels-debootstrap/target-fs-
ext2

```

### 4.3 Configuring the Debootstrap Step

The `AK_DB_OVERLAY` environment variable can be used to name a directory that is copied over the generated filesystem. The directory is the root of an overlay tree, so it will most likely include `/etc` and `/bin` as subdirectories.

The variable `AK_DEBS` lists some Debian packages that are added to the default minimal selection. The variable `AK_DB_OPTS` lists a few default arguments to `debootstrap`.

Finally, the variable `AK_DB_FIXUP` (default: `./fs-fixup/`) names a directory of late changes to be applied to the filesystem. The series applied is listed in a file called `SERIES`.

The distributed package performs the following changes:

- It activates `getty` on `/dev/ttyS0`.
- It removes the scripts to rename the network interfaces.
- It adds the user's public key in root's `authorized_keys`.
- It adds a command to mount `/home` from `10.0.0.1`.
- It changes `/etc/init.d/hostname.sh` to set `“guest-$(uname -r)”`

The last three steps are thought for easier interaction between the host and the guest system: by having your key authorized you can `ssh` as root in the guest without typing a password. Even

if the password of the superuser is empty by default in the guest system, you still have to pass it interactively unless you are authorized.

Further, the `fix-ssh-authorize` script adds the contents of the user's `.ssh/authorized_key_for_kvm` into `/root/.ssh/authorized_keys` in the target filesystem. This has been added in release 2011-05 to allow easy access to other users (I use it for my students, who can thus easily test their modules on the virtual machines).

## 5 Running the Emus

The final script in this package is

```
./scripts/do_qemus
```

But before you can run it you have to setup the host system with an appropriate network configuration.

### 5.1 Preparing the Host System

The *qemu* or *kvm* guest systems are expected to communicate over the network with the host system. I only support IPV4 so far.

Since I use 192.168.x.y on my physical network, I chose to use 10.x.y.z for my herd of emus. This won't work if you run network 10 on the physical wires and you'll have to fix IP numbers by hand (I'm sorry, this was for internal use and generalizing takes a lot of time – like documenting and publishing, which was not initially planned).

You are thus expected to set up a bridge interface on the host, running address 10.0.0.1 on the host side. You can also route to the physical network, but this is not described here.

The commands to set up the host systems are as follows, assuming you have the bridge interface configured in the host kernel:

```
sudo brctl addbr br0
sudo ifconfig br0 10.0.0.1
```

### 5.2 Actual Startup of the Herd

The guest systems are run in the background, so no error is checked or reported. you'll need to check the output of each guest engine in the file `qemu-35.log` (substitute your version for 35), in the build directory of the kernels.

The invocation of each guest is delayed 10 seconds from the previous one, as I had some overload issues on one of my hosts.

All guests mount the same filesystem in "snapshot" mode. I use the debootstrap thing I created in the previous step, or other file systems (like a busybox-only thing).

Each *qemu* uses an IP address of the form 10.2.6.x, where *x* is the integer part of the version number. The host is expected to have 10.0.0.1 on a bridge interface, whence `/home` is mounted.

Each *qemu* uses a VNC session for its VGA display (there you can login as root with no password). The number of the session is `:x`, where *x* is the integer version number (e.g.: 35).

Each *qemu* has a serial console, that you can connect to UDP or *pty* or whatever (please edit the script to uncomment the proper lines. I planned to have several serial ports with console messages, but it seems only one works).

The default in the distributed script is for the serial port to be saved to a file, so you'll get all the `kmsg` saved in case of panic. When saving to a file, there is no input available for the *qemu*, so the `getty` on the serial port is not used, but I sometimes change the serial configuration so I prefer to always have `getty` running).

If you use the UDP console, the port number is `2600+$version`, for example 2635 for the 2.6.35 kernel.

### 5.3 Example Run

```
Variable check:      AK_BASEDIR = /opt/all-kernels
Variable check:      AK_KLIST = 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
Variable check:      AK_BUILD_DIR = /opt/all-kernels/./all-kernels-build
Variable check:      AK_QEMU = qemu
You may now ctrl-C for 3 seconds...
[sudo] password for rubini:
Running qemu for version 20: IP 10.2.6.20, vnc :20
Running qemu for version 21: IP 10.2.6.21, vnc :21
Running qemu for version 22: IP 10.2.6.22, vnc :22
[...]
Running qemu for version 35: IP 10.2.6.35, vnc :35
Running qemu for version 36: IP 10.2.6.36, vnc :36
Running qemu for version 37: IP 10.2.6.37, vnc :37
```

Note that you can set `AK_QEMU` to either *kvm* (default) or *qemu*.

If all goes well, you'll be able to ssh in each of the guest systems. For example this script shows the version number reported by each of them (they are laid out differently as the default for git-hosted builds changed over time, but all of them are 2 or three commits away from the master branch as described earlier):

```
$ for n in $AK_KLIST; do ssh root@10.2.6.$n uname -r; done
2.6.20-ge37352b4
2.6.21-g00bca0ec
2.6.22-g3477da1a
2.6.23-g53e3636f
2.6.24-g32686a3c
2.6.25-00002-gc4cd0b0
2.6.26-00002-g4e4b45e
2.6.27
2.6.28
2.6.29
2.6.30
2.6.31
2.6.32
2.6.33
2.6.34
2.6.35+
2.6.36+
2.6.37+
```

### 5.4 Configuring the Qemu Run

As usual, part of the configuration is performed through environment variables. The list of kernel versions is found in `AK_KLIST` and the name of the guest engine is in `AK_QEMU`.

The `AK_KLIST` variable usually includes plain numbers like "20 28 30 35". If one name in `AK_KLIST` is for example "38-rc6", the script extracts "38" as version number. Such version number is used to build an IP address (`10.2.6.$version`), a VNC display number (`:$version`) and a UDP port number (`2600+$version`). This means you can run only one RC kernel at a time, but this is enough for me at this time.

No other configuration has been abstracted at this point. You can edit the script itself if needed.

# Table of Contents

<b>Introduction</b> .....	<b>1</b>
<b>1 Prerequisites</b> .....	<b>1</b>
<b>2 Package Outline</b> .....	<b>1</b>
<b>3 Compiling the Kernels</b> .....	<b>2</b>
3.1 Using Git and Alternates .....	2
3.2 Actual Kernel Compilation .....	2
3.3 Example Run .....	3
3.4 Configuring the Kernel Build .....	4
<b>4 Building the Filesystem</b> .....	<b>4</b>
4.1 Actual Debootstrap Run .....	4
4.2 Example Run .....	4
4.3 Configuring the Debootstrap Step .....	5
<b>5 Running the Emus</b> .....	<b>6</b>
5.1 Preparing the Host System .....	6
5.2 Actual Startup of the Herd .....	6
5.3 Example Run .....	7
5.4 Configuring the Qemu Run .....	7