debian

Embedding Debian

From Debian to Embedded

(look mom… no hands)

Oron Peled <oron@actcom.co.il>

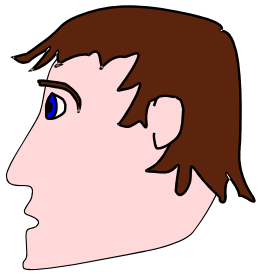
Who am I ?

- Unix veteran (BSD 4.1/4.2, SGI, HP-UX, old Solaris)
- My first Linux – Slackware (1993, kernel 0.99pl14)
- Moved to RedHat – 1995
- Migrated to Fedora – (FC2):
- All my personal servers/desktops/laptops (FC15/KDE)
- Community involvement... next to nil – shame on me...
- Debian:
  - Sporadic use through the years
  - Not a Debian Developer
  - Never maintained a single Debian package...

Why Debian for embedded systems ?

- Free Software – no hidden agendas
- Major distribution – large package selection
- Many official hardware architectures
- Also, non-official hardware architectures
- Live embedded community -- http://embedian.org
- Long release cycle (~2 years)
  - Yes, I'm talking about Debian Stable here.

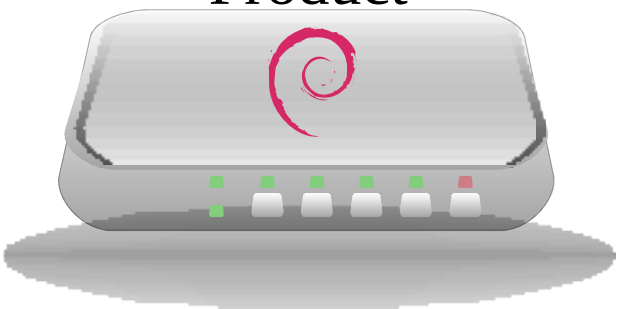
The challenge



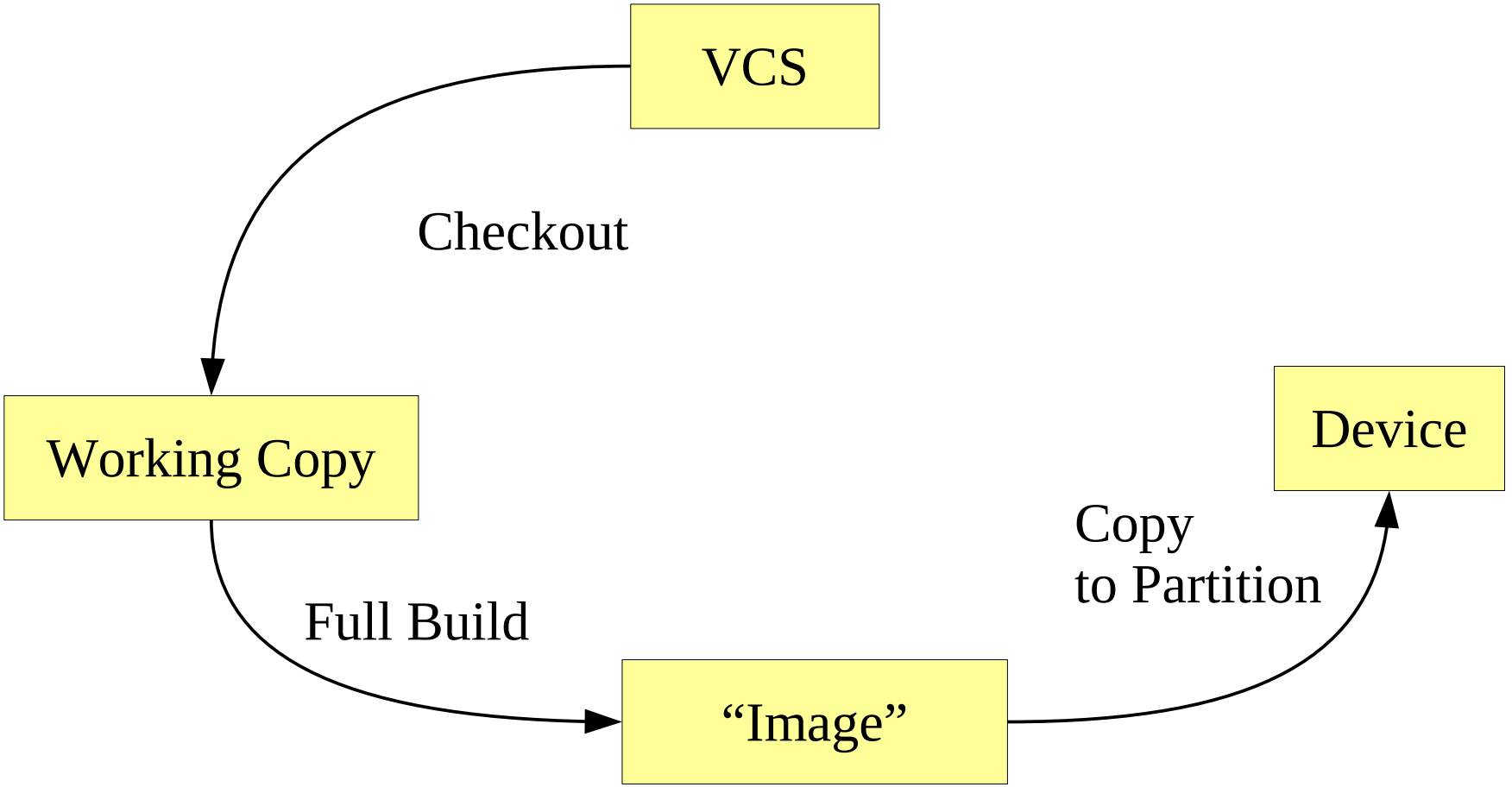
Write Software

Build, Commit, Re-Build, Install, Test...

Product



Typical embedded build



Note: Usually two partitions are used round-robin (fail-safe)

Problem 1

- Complete “image” written to device (partition)
- How much time will it take?
  - Old embedded devices had ~8-64 Mb
  - What about ~128Mb, ~256Mb, ~512Mb ?

Solution 1 - overview

- Update incrementally – only what has changed
- Yes, any Linux distro has been doing it since 1993
- It's called **packages**
- We still want a fail-safe solution:
  - Mount the other partition
  - Chroot into it
  - Run the upgrade on that partition
  - Reboot to new partition

  (if fails, reboot to old partition)

Package Management

- A system is composed of many packages
- Each package has its own version
- How do we define the “product” version?
  - Create a “*product*-release” package
  - Its version is the product version
  - It **depends** on all packages+versions of components
- So an instance of this package “defines” the product:

foobar-release-1.20.9-7

Building *product*-release

- The “source” is a manifest of packages+versions:

```
apt     0.8.10.3
bash    4.1-3
...     ...
```

- Dependencies:
  - First approximation, by installing (and then `dpkg -l`)
  - Automation by wrapping `edos-debcheck` or `apt`
  - Save results and reuse it (determinism)
- During package build:
  - A script generates `debian/substvar` from manifests
  - This is used in the “`Depends`” clause of the package

Problem 2

- Can the same build be reproduced?
- It may depend on the environment:
  - Toolchain
  - Libraries
  - Configuration
- Typical solution: A dedicated “Build-Machine”
  - Maintenance/Configuration management issues
  - Can we reproduce it ?

Solution 2 - overview

- Clean/known build environment is a must
- Any Linux distro has tools to do this
  - The key is building in chroot'ed environment
  - Fresh build environment is set-up for each build
- While we focus on Debian, a similar approach has been implemented for RPM based systems.

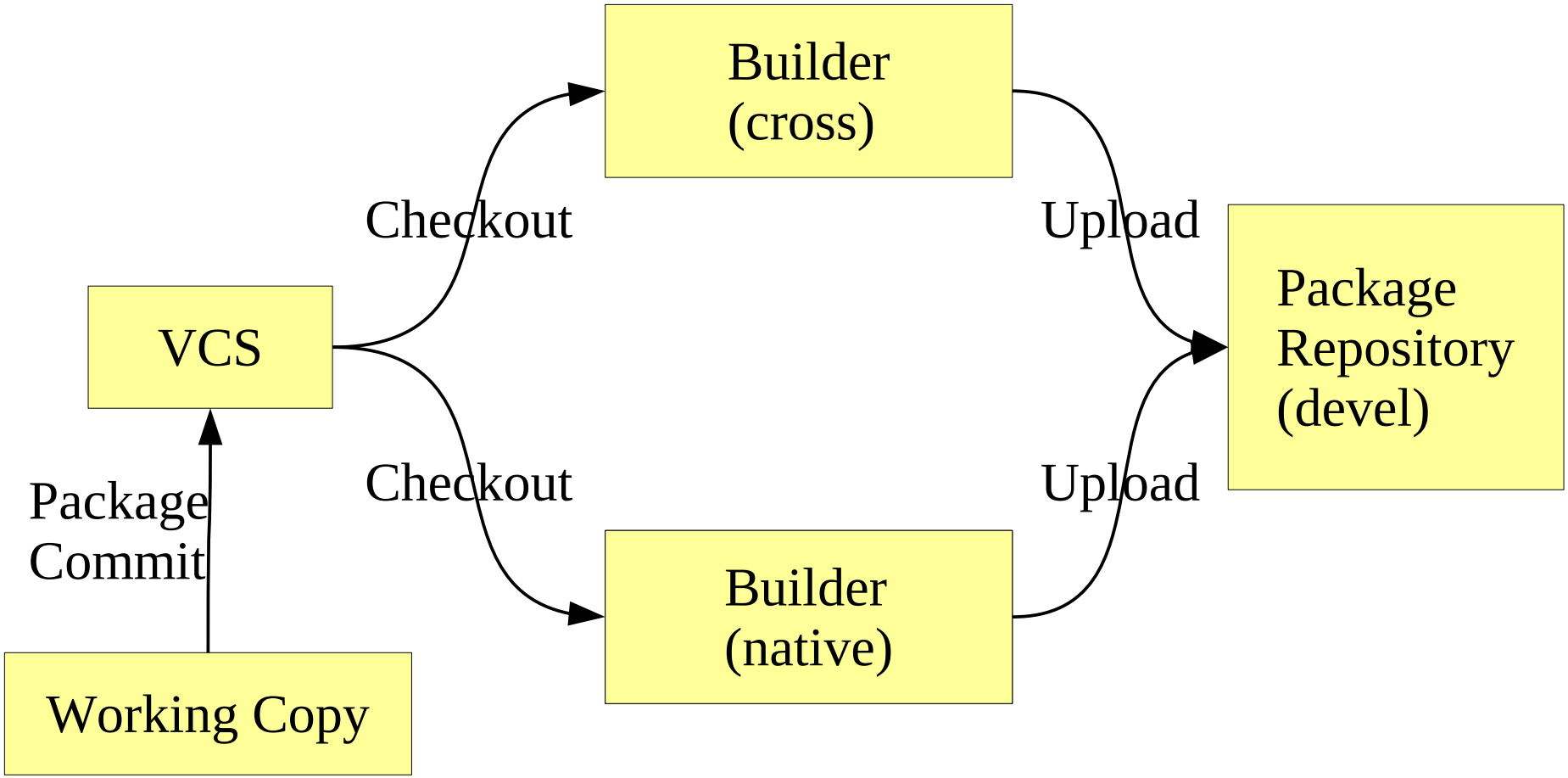
Some Design Decisions

- Opt for native compilation:
  - Was every package designed for cross compilation?
  - Some packages are not trivial (*perl*...)
  - This is how Debian maintain their architectures
- However, there are exceptions:
  - Cross compilation is not always a problem:
    - The (legacy, proprietary) application itself.
    - The kernel
  - Usually a PC is faster than a small embedded device:
    - Did I mention a kernel compile?

Wanted Builds

- Local, interactive compile of a single binary:
  - Usually the application from some IDE (e.g: `eclipse`)
  - Manually copy the result to device
- Local, interactive package build:
  - Copy results (`*.deb`) to device
  - Manual install: `dpkg -install *.deb`
- Automated build:
  - No manual intervention
  - Results to package repository
  - Can install to devices via APT

Automatic Build – from 20,000 feet



Repository – uploading

- Can use either `dupload(1)` or `dput(1)`
- Created a “debrepo” user:
  - Owner of development repositories
  - Upload via `scp` from the builders
  - Builders run as separate user:
    - Their public key is in

      `~debrepo/.ssh/authorized_keys`
    - But without a pass-phrase
- Manual builds **NEVER** reach repositories

Repository – management

- Used `mini-dinstall(1)`:
  - We run it from “debrepo” crontab
  - Can alternatively be run as a daemon
  - Can run a post-install script:
    - I use this to post notification on a local IRC server
- Other goodies:
  - A trivial `~debrepo/public_html/cgi-bin/report`
  - Meta-info from `*.changes` files:

    name, version, who, when, why

What is a package commit ?

- What should trigger a build?
  - Not every source file commit
  - Only when someone bump the version
  - In Debian, it is stored in `debian/changelog`
- Enters `svn-autoreleasedeb(1)`

svn-autoreleasedeb(1)

- Runs from `cron` -- reads an XML config file (Ouch...)
- For each listed package URL:
  - Fetch `<url>/debian/changelog`
  - Compare version string to recorded state
  - If different:
    - Checkout package to temporary directory
    - Build it using `svn-buildpackage`
    - Use `dupload` to send results to repository
- Leaves a lot to be desired... but it's a perl script...

svn-autoreleasedeb – first fixes

- Better reporting:
  - Added state info: timestamp, build status
  - Added cgi script to show status
- Modified so it won't retry repeated failures
- Improved logging: a summary + per-package
- Optimize cron runs via a tiny shell script:
  - Svn commit hook logs commits of `debian/changelog`
  - Skip runs if this log is older than last build state
- Allow build options – e.g: `-a<architecture>`

And now to something completely different

- What is `chroot(2)` ?
  - Allow a process to change its current root directory.
  - Inherited by child processes (just like cwd)
  - Only root user may call `chroot(2)`
  - But a root user may affect things outside of it

    (create device files, mount, access `/sys`, `/proc`, etc.)
- Use cases:
  - Secure services (droping privileges after chroot)
  - Clean build environments – Yay!

debootstrap(8)

- Bootstrap a basic debian system

  `debootstrap squeeze /tmp/my-new-debian`
- A set of shell scripts (can run on non-debian systems)
- Can only be done by the root user
- With "`--variant=minbase`" also include `apt(8)`
- So after a debootstrap, we can (as root):

  `chroot /tmp/my-new-debian`
- And then install other package with `apt(8)`

pbuilder(8)

- Debian “personal builder”
- Create a clean debootstrap'ed environment:

```
pbuilder –create:
```

  - Other parameters taken from configuration file
  - It is saved in a tarball for re-use
- A more efficient variant is `cowbuilder(8)`
  - Same behavior/options as `pbuilder(8)`
  - Use Copy-On-Write for better performance

pbuilder(8) – usage

- Running `pbuilder –build <source package>`:
  - Extract the saved environment
  - Copy the source package into the chroot
  - Install package Build-Depends inside the chroot
  - Runs a package build inside the chroot
  - Copy the results out
- But:
  - The input is a source package
  - These commands are privileged... (chroot)

pdebuild(8)

- A wrapper around `pbuilder/cowbuilder`:
  - The `--builder` command line option select which one
- Running it from within source tree:
  - Build a source package
  - Uses `sudo(8)` to run the builder on this package
  - Can choose the results location
- Patched `svn-autoreleasedeb(8)` to use `pdebuild(8)`

BINGO!

What about interactive builds?

- Host requirements:
  - Install/maintain toolchains
  - Build dependencies
  - What if two branches have conflicting requirements?
  - How about two projects on the same host?

    (composed of different Linux versions)
  - It's getting complex and risky
- Enter `schroot(1)`...

schroot(1)

- A set-uid wrapper for `chroot(8)`:
  - So mortal users can use it
- Controlled access per environment:
  - Drops privileges inside the chroot
  - Which users/groups can use it
  - Which users/groups can use it as root
- Can support COW technologies (LVM, unionfs):
  - A clean environment
- Bind-mount selected directories (e.g: `/home`)

Using schroot(1) in a project

- Created a project wrapper:
  - Runs via sudo
  - Create debootstrapped environment

    (in pre-selected directory: e.g: /srv/schroot/...)
  - Create matching scrhoot configuration
  - Use schroot to install further project dependencies
- Now users can create personal build environments
  - Even Fedora users can build Debian packages...

Handling cross-compilation libraries

- The cross compiler need to find target libraries:
  - Basic libraries (e.g: glibc) are part of the toolchain
  - But what about project built libraries?
- Embedian created `dpkg-cross(1)`:
  - Input `libfoo-dev_1.2.3-4_<arch>.deb`
  - Output: `libfoo-dev-<arch>-cross_1.2.3-4_all.deb`
    - Throws everything but libraries, headers, etc.
    - Relocates files into toolchain's sys-root

Integrating cross libraries

- Prepare a separate “cross” repository
- In the development repository post-install script:
  - Check if the package is included in a defined list
  - If yes, run dpkg-cross on the package *.deb file
  - Move the result into the “cross” repository

Questions?

Thank You!