roar-ng User Manual

Table of Contents

Overview

roar-ng is a generic distribution building system originally forked (but now independent) from Woof, the Puppy Linux build system.

It provides an architecture-independent, flexible and portable infrastructure for the creation of fast and portable "live" GNU/Linux distributions. It provides support for the binary package format and repositories of various GNU/Linux distributions.

The development of roar-ng started as a collection of source hacks of Woof and evolved into a complete, independent re-implementation. It provides advanced features not found in Woof, such as parallel downloads, automatic package splitting, simple branding and easy porting to different processor architectures.

Dependencies

roar-ng has a small number of dependencies and has a small list of packages that must be present in every distribution built by it.

roar-ng Itself

- A POSIX-compliant shell (either Bash or DASH). /bin/dash is required and may be a symlink to Bash.
- For support for some distributions: Python 2.x.
- Squashfs tools.
- cdrkit or cdrtools (for either mkisofs or genisoimage, respectively).
- GNU Binutils.
- cpio.
- gzip.
- Squashfs tools.
- For 4buildpackage: Aufs, in the host's kernel.
- For 1download: aria2, for optional parallel downloads.
- AdvanceCOMP.
- OptiPNG.
- XZ Util.

Result Distribution

- Squashfs, built into the kernel image.
- Aufs, built into the kernel image.
- Drivers for all devices and file systems the distribution can boot from, built into the kernel.
- DASH.
- BusyBox, with mdev.
- dialog.
- e2fsprogs.
- util-linux.

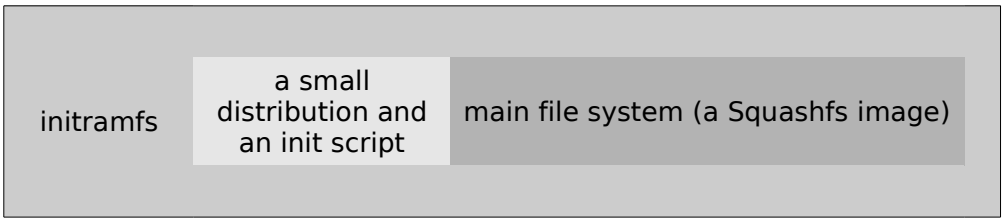
- udev.
- iptables.
- hsetroot.
- gtkdialog.
- rxvt-unicode.
- cwm.
- syslinux.
- Librsvg.
- XZ Util.

Distribution Structure

A typical distribution built using roar-ng consists of four parts:

1. An “initramfs”: the standard Linux boot procedure requires an initial file system, which contains a small operating system and a script (an “init” script) which is able to locate the “real” (or “main”) file system. This file system is loaded into memory by the boot loader and the kernel execute the init script. Once it locates this file system, it switches the file system root (/) to it, using /sbin/switch_root.
2. The main file system, contained in a compressed image placed inside the initramfs.
3. An ISO9660 image which contains a boot loader, the initramfs (which contains the entire distribution) and a kernel image.
4. A “devx” module; a special SFS extension (see later) which contains development packages, static libraries, headers, etc'.

It is quite hard to grasp the structure of this complex mess of file systems and images; here's what the initramfs looks like:



The kernel image and the initramfs are loaded by the boot loader and the kernel executes /init, which is the initial init script contained in the initramfs. This script mounts a layered file system which consists of two layers:

1. A bottom, read-only layer which consists of a Squashfs image containing the main file system (e.g the distribution itself). This is the file system the user sees; the initramfs gets destroyed.
2. A top, read-write layer which is either a temporary file system (a “ramdisk”, e.g tmpfs) on “live” sesions, or a writable image file contained on a disk partition, under persistent sessions. All changes to the file system are saved in this layer.

Once the init script is done setting up the layered file system, it executes /sbin/init, which in turn, executes the “real” init script. This script performs more “high-level” steps performed during the boot sequence, such as the execution of daemons.

The advantages of this structure are:

1. Speed: since the main file system is contained in the system memory, it is faster to read. However, since this is wasteful of expensive memory, it is stored in a compressed form, which is a great compromise between size and consumption of computer resources. This way, it consumes less memory, while preserving the speed advantage.
2. Portability and file system neutrality: since the initramfs' init script does not need to properly recognize and mount a disk partition in order to boot the operating system (since it is contained inside the initramfs already), any boot loader which supports Linux is enough to successfully boot the distribution.
3. Simplicity: the whole operating system consists of two files – the initramfs and a kernel image.
4. Small space footprint: only the changes to the distribution's main file system are actually stored to a partition. Moreover, most the distribution is compressed.
5. Easy administration: in order to revert the operating system to its pristine state, the save file needs to be deleted and that's it.
6. Easy rescue: if the operating system gets messed up, it is extremely easy to boot it into a “live” session which can be used to fix the problem.

However, this structure also has several disadvantages:

1. Slightly higher memory consumption.
2. Less transparency; technology-unaware users will find the operating system structure and implementation extremely hard to understand.
3. It makes it hard to update the operating system, since most of it is read-only; replacing a core package means rebuilding it.

The Skeleton

The roar-ng “skeleton” is a directory tree which contains core files which provide the unique features of its output distributions and therefore distinct between roar-ng and its competitors. It consists of four parts:

1. The main file system skeleton; this portion of the skeleton is the distribution “body”.
2. The initramfs skeleton, used to bridge between the boot loader and the main file system.
3. The “devx” module skeleton.
4. The ISO9660 image skeleton.

Configuration

roar-ng's configuration consists of three configuration files, contained under the “conf” directory:

1. “distrorc”: various distribution details, such as its name, version, etc'.
2. “bootrc”: boot settings, mostly related to the initramfs and save files.
3. “package_list”: a list of packages to be included in the result distribution.

Package List

The package list is a text file which contains the list of packages processed by roar-ng. Each line represents a meta-package (a package containing one or more binary packages of the same distribution) or a comment (which begins with the “#” character).

Each meta-package appears in the following format:

```
distribution|meta-package name|packages included in it|redirection rules
```

The redirection rules field contains a list of rules which tell roar-ng how to handle each component of the meta-package, once it is split by the 2createpackages; read later. Here is an example redirection rules field:

```
exe,dev>doc,doc>null,nls
```

Each module can be either redirected to another module (using the “>” syntax; in this example, the development files module is redirected to the documentation module), kept where it is (without any special syntax, as the “exe” and “nls” modules in the example) or deleted (using the “null” redirection).

Package Management

RXZ

“RXZ” (which stands for “roar-ng XZ”) is the native package format of distributions built using roar-ng. In contrast with other package formats, such as DEB or RPM packages, which contain only a subset of the compiled source package, RXZ packages contain whole, raw packages. They are compressed using LZMA2, through XZ-Util.

hpm

hpm (acronym of “Humble Package Format”) is a simple package manager that handles only two operations: the installation and removal of binary packages in the RXZ format.

SFS Extensions

“SFS extensions” are dynamically-loadable Squashfs images which contain either big packages (e.g the kernel sources) or whole suites of packages (e.g an entire desktop environment). They can be loaded at run-time using /usr/sbin/load_sfs (which pushes them into the layered file system, below the top, writable layer), but cannot be unloaded.

The advantage of SFS extensions over regular RXZ packages is their ability to be installed and removed transparently, without leaving any traces or wasting precious space in the save file.

Usage Instructions

0setup

0setup is the first script in the execution chain of roar-ng. Its purpose is to download the package lists of each supported distribution's package repositories, then convert them to a simple, common format that provides more efficient searching. Its result package lists are placed under the “repos” directory.

Its flow is linear and very simple:

```
for every distribution:
     for every repository of the distribution:
          download the package list
          convert the package list to the common format
```

Support for various distributions is implemented using a directory hierarchy, rather than conditions in each script's code. Each distribution has its own directory under “distro”, which

contains several files:

1. “repositories”: a list of package repositories, the URL of each repository's package list and a unique name. In addition, this file contains a list of package download mirrors, so binary packages can be downloaded from multiple sources concurrently, speeding up their download.
2. “convert_package_list”: a script which receives a distribution's original package list as a parameter and outputs a package list in roar-ng's simple format to its standard output.
3. “extract_package”: a script which receives a binary package and an output directory, then extracts the former into the latter.

Usage

0setup does not receive any parameters:

```
./0setup
```

1download

Once 0setup was executed, roar-ng is able to locate the result distribution's packages; that's where 1download steps in. It is a script which downloads all packages specified in the package list and puts them under “packages”.

1download was designed to be fast and efficient; download times are slow due to the large number of packages, so there is very little room for improvement in this area. However, the procedure used to locate each package and find appropriate download links has been streamlined and improved, in order to reduce its overhead.

Usage

1download does not receive any parameters:

```
./1download
```

It is possible to execute the script multiple times; it will download any extra packages added to the package list, but packages removed from it will still be present in the “packages” directory.

2createpackages

2createpackages is the package processing script of roar-ng: it processes the packages downloaded by 1download (and placed under “packages”) and puts their processed contents under “processed-packages”. Here's how it works:

```
for each package in the package list:
      run its distribution's extract_package script
      if there is a package template:
            apply the template
      optimize the package
      split the package
```

Package Templates

Each binary package downloaded by 1download gets extracted and a special template (located under “package-templates”) is copied into its directory. This template contains files and directories added to the script and an optional script, called “hacks.sh”, which runs from the extracted package's directory and performs extra modifications to it.

Package templates may contain “post_install.sh” (as in native roar-ng packages), which gets executed in the final stage of roar-ng.

Package Optimization

After the package extraction and the basic modifications performed by the “hacks.sh” script, each package is optimized, non-recursively. Testing shows that recursive optimization (I.e extraction of archives and optimization of their contents) is extremely slow and provides very little benefit.

This optimization consists of removal of debugging symbols from binaries, thorough optimization of images, re-compression of archives and more.

Package Splitting

Each optimized package is split into four parts using the splitpkg script: a main package, a development files package, a documentation package and language support files package.

Redirection

The redirection feature of roar-ng is very simple: each sub-package's contents can moved to another or removed. 2Createpackages performs this after the package splitting stage.

Usage

2createpackages does not receive any parameters:

```
./2createpackages
```

It is possible to execute the script multiple times; it will process any packages not present under “processed-packages”.

In order to re-process a package (for instance, because it was replaced), the processed package must be deleted first:

```
cd processed-packages
rm -rf $name ${name}_DEV ${name}_DOC ${name}_NLS
```

3builddistro

3builddistro is the final script of roar-ng. It builds a bootable ISO9660 image of the distribution, according to the supplied configuration. All its output is contained under the “sandbox3” working directory.

The script uses a skeleton for each component of its output; see the “skeleton” directory. The skeletons of the initramfs, root file system and the “devx” module consist of multiple parts (each in its own directory), while the ISO9660 image's doesn't.

It is recommended to edit the script before its execution, in order to tweak various parameters, such as the default desktop background.

Usage

3builddistro does not receive any parameters:

```
./3builddistro
```

4buildpackage

4buildpackage is a fourth, extra script which provides the ability to build RXZ packages, according to build scripts contained under the “devx” module's skeleton. It is the successor of a tool called “Builder”, which performs automatic package building, in a more complex way.

Each package built by the script is placed under “built-packages” and its source files are downloaded automatically by the script, into sandbox3/rw/tmp/build/$name. If a directory bearing the package name exists under “sources” prior to the script's execution, its contents

are copied to that directory, speeding up the building process by shortening or eliminating download times. However, source files are not copied back into the sources directory: this has to be done manually.

Usage

4buildpackage receives only one parameter, which is the package name:

```
./4buildpackage $package
```

The script must be executed after 3builddistro, as it performs the package building inside a chroot jail of its output.