CRUX Handbook

RELEASE 1.1

CRUX Handbook: RELEASE 1.1

Published 2003-05-03

This handbook covers the installation, configuration and administration of CRUX. Please note that this handbook only covers topics that are specific to CRUX [http://crux.nu/]. For further information about Linux see the Linux Documentation Project [http://www.tldp.org/].

Table of Contents

Preface

Per Lidén [mailto:per@fukt.bth.se] wrote this handbook. Robert McMeekin [mailto:viper@mcmeekin.info] converted it to DocBook. Numerous others have given feedback and improvement suggestions.

Chapter 1. Introduction

1.1. What is CRUX?

CRUX is a lightweight, i686-optimized Linux distribution targeted at experienced Linux users. The primary focus of this distribution is “keep it simple”, which it reflects in a simple tar.gz-based package system, BSD-style initscripts, and a relatively small collection of trimmed packages. The secondary focus is utilization of new Linux features and recent tools and libraries. CRUX also has a ports system which makes it easy to install and upgrade applications.

1.2. Why use CRUX?

There are many Linux distributions out there these days, so what makes this distribution any better than the others? Well, it's all about taste really. I can give you a hint about my taste, and perhaps we share the same taste, or we don't. First of all, I want a distribution made with simplicity in mind from beginning to end. Further, I want my packages up-to-date, not the latest bleeding-edge-alpha version, but the latest stable version. I want to easily create new and update old packages (updating a package in CRUX is often just a matter of typing **`pkgmk -d -u`**). I want packages optimized for my processor (think **`-march=i686`**). I don't want my filesystem cluttered with files I never use (think `/usr/doc/*`, etc). If I need more information about a specific program, other than information found on the man-page, I'll find it on the net. And finally, I want to use new features offered by recent Linux kernels (think devfs, reiserfs, ext3fs, etc).

If you are a somewhat experienced Linux user that wants a clean and solid Linux distribution as the foundation of your installation, prefers editing configuration files with an editor to using a GUI, and doesn't hesitate to download and compile programs yourself, then this distribution might suit you well.

1.3. License

1.3.1. Packages

Since CRUX is a Linux distribution, it contains software written by a lot of different people. Each software package comes with its own license, chosen by its author(s). To find out how a particular package is licensed, have a look at its source code.

1.3.2. Build Scripts

All package build scripts in CRUX (in package categories `base` and `opt`) are Copyright © 2000-2003 by Per Lidén and licensed through the GNU General Public License [http://www.gnu.org/copyleft/gpl.html].

1.3.3. NO WARANTY

CRUX is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. Use it at YOUR OWN RISK.

Chapter 2. Installing CRUX

2.1. Supported Hardware

Packages on the official CRUX ISO image are compiled with optimization for i686 (Pentium-Pro/Celeron/Pentium-II or better) processors. Do *not* try to install it on an i586 (Pentium, AMD K6/K6-II/K6-III) or lower processor, since it simply will not work. To install CRUX on an i586 system you need to download the i586 version of the CRUX ISO image.

The kernel used during installation, i.e. when booting from the CRUX ISO image (El Torito), is compiled with the following disk controllers and USB support:

Subsystem	**Driver(s) included in bootkernel**
IDE	Generic PCI IDE chipset
SCSI	7000FASST, ACARD, Adaptec AACRAID, Adaptec AIC7xxx, Adaptec I2O RAID, AdvanSys, AM53/79C974, AMI MegaRAID, BusLogic, Compaq Fibre Channel, NCR5380/53c400, IBM ServeRAID, SYM53C8XX, Tekram DC390(T) and Am53/79C974
USB	USB device filesystem, UHCI (Intel PIIX4, VIA, ...) support, USB Human Interface Device (full HID) support, HID input layer support

In order to install CRUX, your disk controller must be present in the list above. If your hardware is not supported or you have other problems installing CRUX you might find a solution in Section 2.4.

2.2. Installing From CD-ROM

1. Download the CRUX ISO image (`crux-1.1.iso`). To ensure that the download was successful you should examine its checksum using **`md5sum`**.

   ```
   $ md5sum crux-1.1.iso
   ```

   Compare the output with the file `crux-1.1.md5sum`, which can be found in the same directory as the ISO image on the download site. If the checksums match the download was successful and you can continue by burning the ISO image on a CD.

2. The ISO image is bootable, just insert the newly burned CD and reboot your computer. Press `Enter` at the boot prompt.

3. Login as **`root`** (no password required).

4. Create (if necessary) and format the partition(s) you want CRUX to be installed on.

   ```
   $ fdisk /dev/discs/disc?/disc
   $ mkreiserfs /dev/discs/disc?/part?
   $ mkswap /dev/discs/disc?/part?
   ```

   The amount of disk space you need depends on how many packages you choose to install. I recommend having at least a 1G root partition (CRUX will use about 200MB-500MB depending on your configuration).

   Using ReiserFS is recommended, but there is support for Ext2fs/Ext3fs and JFS as well. Further, I highly

recommend separating system data from user data, i.e. use a separate partition for `/home` (and possibly `/var`) since that will make your life a lot easier the day you want to upgrade/reinstall/remove your system.

Note

Make sure that any BIOS Virus Protection option is DISABLED as this option may prevent **`fdisk`** from writing new partitions correctly.

5. Mount the partition on which you want to install this distribution.

```
$ mount /dev/discs/disc?/part? /mnt
```

If you want the installation to span more than one partition, mount those partitions as well. For example, if you want to have a different partition for `/home` or `/var`, then do:

```
$ mkdir /mnt/var
$ mount /dev/discs/disc?/part? /mnt/var
```

6. Activate your swap partition(s).

```
$ swapon /dev/discs/disc?/part?
```

7. Type **`Setup`** to start the package installation script. The script will ask where you mounted your new root partition and which packages you want to install. Just select the packages you want and nothing else will be installed. However, I recommend at least installing all the packages marked `base`.

Once it has installed the selected packages, the **`Setup`** script will display an installation log. Make sure the last line in the log says "`0 error(s)`".

If you at a later stage find that you need some additional packages you can just mount the CRUX CD-ROM and use **`pkgadd`** to install them.

Note

There is no package dependency checking. This means that it is up to you to figure out that if you for example install the `sendmail` package you also need to install the `db` package.

Screenshots of Setup [http://crux.nu/doc/screenshots.html]

8. Now it's time to compile your kernel and do basic system configuration. The kernel compilation requires that you "chroot" into your new CRUX installation.

```
$ mount -t devfs devfs /mnt/dev
$ mount -t proc proc /mnt/proc
$ chroot /mnt /bin/bash
```

9. Set the **`root`** password.

```
$ passwd
```

10. Edit `/etc/fstab` to configure your filesystem(s). Editors **`vim`** and **`pico`** are available.

11. Edit `/etc/rc.conf` to configure keyboard, services and timezone. See Section 5.1.3 for details about `/etc/rc.conf`.

12. Edit `/etc/rc.d/net`, `/etc/hosts` and `/etc/resolv.conf` to configure your network (ip-address/gateway/hostname/domain/dns).

13. Go to `/usr/src/linux-2.4.20`, configure and compile a new kernel.

```
$ cd /usr/src/linux-2.4.20
$ make menuconfig
$ make dep
$ make clean
$ make bzImage
$ make modules
$ make modules_install
$ cp arch/i386/boot/bzImage /vmlinuz
$ cp System.map /
```

Remember to enable the following kernel options:

```
Code maturity level options  --->
      [*] Prompt for development and/or incomplete code/drivers

File systems  --->
      [*] /dev file system support
      [*]   Automatically mount at boot
```

14. Edit `/etc/lilo.conf` to boot the kernel you just compiled and run **`lilo`** to make the new system bootable.

15. Remove the CRUX CD-ROM from your drive and reboot from harddisk.

2.3. Upgrading From CD-ROM

1. Download the CRUX ISO image (`crux-1.1.iso`). To ensure that the download was successful you should examine its checksum using **`md5sum`**.

```
$ md5sum crux-1.1.iso
```

Compare the output with the file `crux-1.1.md5sum`, which can be found in the same directory as the ISO image on the download site. If the checksums match the download was successful and you can continue with burning the ISO image on a CD.

2. The ISO image is bootable, just insert the newly burned CD and reboot your computer. Press `Enter` at the boot prompt.

3. Login as **`root`** (no password required).

4. Mount your CRUX root partition.

```
$ mount /dev/discs/disc?/part? /mnt
```

If your installation spans over more than one partition, then mount these partitions as well. For example, if you have a different partition for `/var`, then do:

```
$ mount /dev/discs/disc?/part? /mnt/var
```

5. Activate your swap partition(s).

```
$ swapon /dev/discs/disc?/part?
```

6. Type **Setup** to start the package installation script. The script will ask you where you mounted your root partition and which packages you want to upgrade. It is a good idea to upgrade all packages, else you might get into trouble later, e.g. in case a new version of some library isn't 100% backwards compatible.

   When the **Setup** script has upgraded the selected packages an upgrade log will be displayed. Make sure the last line in the log says "`0 error(s)`".

   If you at a later stage find that you need some additional packages you can just mount the CRUX CD-ROM and use **pkgadd** to install them.

7. Now it's time to compile your kernel. The kernel compilation requires that you "chroot" into your CRUX installation.

```
$ mount -t devfs devfs /mnt/dev
$ mount -t proc proc /mnt/proc
$ chroot /mnt /bin/bash
```

8. Go to `/usr/src/linux-2.4.20`, configure and compile a new kernel. Remember to enable the following kernel options:

```
Code maturity level options  --->
      [*] Prompt for development and/or incomplete code/drivers

File systems  --->
      [*] /dev file system support
      [*]   Automatically mount at boot
```

9. Edit `/etc/lilo.conf` to boot the kernel you just compiled and run **lilo** to make the new system bootable.

10. Remove the CRUX CD-ROM from your drive and reboot from harddisk.

2.4. Alternative Installation Methods

2.4.1. Building Your Own Bootkernel

If you are unable to install CRUX from CD-ROM because your hardware is not supported by the bootkernel you can build your own bootkernel and add whatever hardware support you need. To do this you need a 1.44Mb floppy disk, access to another Linux box and the CRUX ISO image burned on a CD. Basic knowledge about how to configure and compile the Linux kernel is of course also required.

1. Build a new kernel with support for your hardware. Use the kernel configuration used by the existing bootkernel as a starting point (you can find it here [ftp://ftp.fukt.bth.se/pub/os/linux/crux/latest/misc/]) and add the hardware support you need. If the kernel gets too big it is safe to remove SCSI and USB drivers (unless you need them of course), but do not remove any filesystem related options.

2. Download the boot floppy creation kit

[ftp://ftp.fukt.bth.se/pub/os/linux/crux/latest/misc/mkbootfloppy.tar.gz] and unpack it.

3. Go into the `mkbootfloppy` directory and execute the **`mkbootfloppy`** script (as **`root`**). This script requires one argument, the kernel image you want to place on the floppy image. Before you do this, make sure you do not have anything mounted on `/mnt` since the **`mkbootfloppy`** script will use that as a mount point.

```
$ cd mkbootfloppy
$ ./mkbootfloppy /path/to/linux/kernel/arch/i386/boot/bzImage
1440+0 records in
1440+0 records out
mke2fs 1.27 (8-Mar-2002)
Added CRUX *
```

4. Write the resulting `boot.img` file to a floppy disk.

```
$ dd if=boot.img of=/dev/fd0
```

5. Insert both the floppy disk and the CRUX CD into the system you want to install CRUX on and reboot.

6. Install CRUX.

2.4.2. Network Installation

If you do not have a CD burner, are unable to boot your machine using the CRUX CD-ROM or for any other reason are unable to install CRUX the normal way (Section 2.2) you might want to check out Martin Opel's CRUX Network Setup Guide [http://rfhs8012.fh-regensburg.de/~opel/crux/network-setup.php] or HOWTO install CRUX via NFS [http://home.t-online.de/home/juergen.daubert/crux/doc/crux-nfs-install.html] by Jürgen Daubert.

Chapter 3. The Package System

3.1. Introduction

The package system (pkgutils [http://www.fukt.bth.se/~per/pkgutils/]) is made with simplicity in mind, where all packages are plain `tar.gz` files (i.e. without any kind of meta data). Packages follow the naming convention `<name>#<version>-<release>.pkg.tar.gz`, where `<name>` is the name of the program, `<version>` is the version number of the program, and `<release>` is the version number of the package. The `pkg.tar.gz` extension is used (instead of just `tar.gz`) to indicate that this is not just any `tar.gz` file, but a `tar.gz` that is meant to be installed using **pkgadd**. This way it is easy to tell packages apart from other `tar.gz` files.

pkgadd(8), **pkgrm(8)**, **pkginfo(8)**, and **pkgmk(8)** are the package management utilities. With these utilities you can install, uninstall, inspect, make packages and query the package database.

When a package is installed using **pkgadd** a new record is added to the package database (stored in `/var/lib/pkg/db`). The package system does not have any kind of dependency checking, thus it will not warn you if you install a package that requires other packages to be installed.

The following sections will in short describe how to use the package utilities. Additional information about these utilities can be found on their respective man page.

3.2. Using the Package System

3.2.1. Installing a Package

Installing a package is done by using **pkgadd**. This utility requires at least one argument, the package you want to install. Example:

```
$ pkgadd bash#2.05-1.pkg.tar.gz
```

When installing a package the package manager will ensure that no previously installed files are overwritten. If conflicts are found an error message will be printed and **pkgadd** will abort without installing the package. The error message will contain the names of the conflicting files. Example:

```
$ pkgadd bash#2.05-1.pkg.tar.gz
bin/sh
usr/man/man1/sh.1.gz
pkgadd error: listed file(s) already installed (use -f to ignore and overwrite)
```

To force the installation and overwrite the conflicting files you can use the option **-f** (or **--force**). Example:

```
$ pkgadd -f bash#2.05-1.pkg.tar.gz
```

The package system allows a file to be owned by exactly one package. When forcing an installation the ownership of the conflicting files will be transferred to the package that is currently being installed. Directories can however be owned by more then one package.

Warning

It is often not a good idea to force the installation unless you really know what you are doing. If a package conflicts with already installed files it could be a sign that the package is broken and installs unexpected files. Use this option with extreme care (preferably not at all).

As said earlier the package file itself does not contain any meta data. Instead the package manager uses the package filename to determine the package name and version. Thus, when installing a package file named `bash#2.05-1.pkg.tar.gz` the package manager will interpret this as a package named `bash` at version `2.05-1`. If **pkgadd** is unable to interpret the filename (e.g. # is missing or the filename does not end with `.pkg.tar.gz`) an error message will be printed and **pkgadd** will abort without installing the package.

3.2.2. Upgrading a Package

Upgrading a package is done using **pkgadd** with the **-u** option. Example:

```
$ pkgadd -u bash#2.05-1.pkg.tar.gz
```

This will replace the previously installed `bash` package with the new one. If you have not previously installed `bash`, **pkgadd** will print an error message. The package system does not care about the version number of the package in that you can "upgrade" version `2.05-1` with version `2.04-1` (or even with version `2.05-1` itself). The installed package will be replaced with the specified package.

Upgrading a package is equivalent to executing **pkgrm** followed by **pkgadd** with one (big) exception. When upgrading a package (with **pkgadd -u**) you have the option to prevent some of the already installed files from getting replaced. This is typically useful when you want to preserve configuration and log files.

When executing **pkgadd** the file `/etc/pkgadd.conf` will be read. This file can contain rules describing how **pkgadd** should behave when doing upgrades. A rule is built out of three fragments; *event*, *pattern* and *action*. The event describes in what kind of situation this rule applies. Currently only one type of event is supported, that is `UPGRADE`. The *pattern* is a filename pattern expressed as a regular expression and the action applicable to the `UPGRADE` event is `YES` or `NO`. More than one rule of the same event type is allowed, in which case the first rule will have the lowest priority and the last rule will have the highest priority. Example:

```
#
# /etc/pkgadd.conf: pkgadd(8) configuration
#

UPGRADE         ^etc/.*$                NO
UPGRADE         ^var/log/.*$            NO
UPGRADE         ^etc/X11/.*$            YES
UPGRADE         ^etc/X11/XF86Config$    NO

# End of file
```

The above example will cause **pkgadd** to never upgrade anything in `/etc/` or `/var/log/` (subdirectories included), except files in `/etc/X11/` (subdirectories included), unless it is the file `/etc/X11/XF86Config`. The default rule is to upgrade everything, rules in this file are exceptions to that rule.

Note

A *pattern* should never contain an initial "/" since you are referring to the files in the package, not the files on the disk.

If **pkgadd** finds that a specific file should not be upgraded it will install it under `/var/lib/pkg/rejected/`. The user is then free to examine, use and/or remove that file manually. Files in this directory are never added to the package database. Example (using the above `/etc/pkgadd.conf`):

```
$ pkgadd -u bash#2.05-1.pkg.tar.gz
pkgadd: rejecting etc/profile, keeping existing version
$ ls /var/lib/pkg/rejected/
etc/
$ ls /var/lib/pkg/rejected/etc/
profile
```

3.2.3. Removing a Package

Removing a package is done by using **pkgrm**. This utility requires one argument, the name of the package you want to remove. Example:

```
$ pkgrm bash
```

Warning

This will remove all files owned by the package, no questions asked. Think twice before doing it and make sure that you did not misspell the package name since that could remove something completely different (e.g. think about what could happen if you misspelled `glib` as `glibc`).

3.2.4. Querying the Package Database

Querying the package database is done using **pkginfo**. This utility has a few options to answer different queries.

Option	Description
-i, **--installed**	List installed packages and their version.
-l, **--list** ***package*|*file***	List files owned by the specified ***package*** or contained in ***file***
-o, **--owner** ***file***	List owner of ***file***.

Examples:

```
$ pkginfo -i
audiofile 0.2.3-1
autoconf 2.52-1
automake 1.5-1
<...>
xmms 1.2.7-1
zip 2.3-1
zlib 1.1.4-1

$ pkginfo -l bash
bin/
bin/bash
bin/sh
etc/
etc/profile
usr/
usr/man/
usr/man/man1/
usr/man/man1/bash.1.gz
usr/man/man1/sh.1.gz

$ pkginfo -l grep#2.5-1.pkg.tar.gz
usr/
usr/bin/
usr/bin/egrep
usr/bin/fgrep
usr/bin/grep
usr/man/
usr/man/man1/
usr/man/man1/egrep.1.gz
usr/man/man1/fgrep.1.gz
usr/man/man1/grep.1.gz

$ pkginfo -o bin/ls
e2fsprogs  usr/bin/lsattr
fileutils  bin/ls
modutils   sbin/lsmod
```

3.3. Creating Packages

Creating a package is done using **pkgmk**. This utility uses a file called `Pkgfile`, which contains information about the package (such as name, version, etc) and the commands that should be executed in order to compile the package in question. To be more specific, the `Pkgfile` file is actually a **bash(1)** script, which defines a number of variables (`name`, `version`, `release` and `source`) and a function (`build`). Below is an example of what a `Pkgfile` file might look like. The example shows how to package the **grep(1)** utility. Some comments are inserted for explanation.

```
# Specify the name of the package.
name=grep

# Specify the version of the package.
version=2.4.2

# Specify the package release.
```

```
release=1

# The source(s) used to build this package.
source=(ftp://ftp.ibiblio.org/pub/gnu/$name/$name-$version.tar.gz)

# The build() function below will be called by pkgmk when
# the listed source files have been unpacked.
build() {
   # The first thing we do is to cd into the source directory.
   cd $name-$version

   # Run the configure script with desired arguments.
   # In this case we want to put grep under /usr/bin and
   # disable national language support.
   ./configure --prefix=/usr --disable-nls

   # Compile.
   make

   # Install the files, BUT do not install it under /usr, instead
   # we redirect all the files to $PKG/usr by setting the DESTDIR
   # variable. The $PKG variable points to a temporary directory
   # which will later be made into a tar.gz-file. Note that the
   # DESTDIR variable is not used by all Makefiles, some use prefix
   # and others use ROOT, etc. You have to inspect the Makefile in
   # question to find out. Some Makefiles do not support redirection
   # at all. In those cases you will have to create a patch for it.
   make DESTDIR=$PKG install

   # Remove unwanted files, in this case the info-pages.
   rm -rf $PKG/usr/info
}
```

In reality you do not include all those comments, thus the real `Pkgfile` for **`grep(1)`** looks like this:

```
# $Id: package.xml,v 1.1 2003/04/28 23:18:22 per Exp $
# Maintainer: Per Lidén <per@fukt.bth.se>

name=grep
version=2.4.2
release=1
source=(ftp://ftp.ibiblio.org/pub/gnu/$name/$name-$version.tar.gz)

build() {
  cd $name-$version
  ./configure --prefix=/usr --disable-nls
  make
  make DESTDIR=$PKG install
  rm -rf $PKG/usr/info
}
```

Note

The `build()` function in the example above is just an example of how **`grep`** is built. The contents of the function can differ significantly if the program is build in some other way, e.g. does not use **`auto-conf`**.

When the `build()` function has been executed, the `$PKG` directory will be made into a package named *<name>#<version>-<release>*`.pkg.tar.gz`. Before the package creation is completed, **`pkgmk`** will check the content of the package against the `.footprint` file. If this file does not exist, it will be created and the test will be skipped. The `.footprint` file will contain a list of all files that should be in the package if the build was successful or a list of all the files that were installed in `$PKG` (if the `.footprint` did not already exist). If there is a mismatch the test will fail and an error message will be printed. You should not write the `.footprint` file by hand. Instead, when a package has been upgraded and you need to update the contents of the `.footprint` file you simply do **`pkgmk -uf`**. This test ensures that a rebuild of the package turned out as expected.

If the package built without errors it's time to install it by using **`pkgadd`** and try it out. I highly recommend looking at the `Pkgfile` in another package(s), since looking at examples is a great way to learn.

3.4. Package Guidelines

3.4.1. General

- The name of a package should always be lowercase (e.g. `name=eterm` and not `name=Eterm`). In case the package is added to the CRUX ports system the excact same name should be use for the name of the directory in the ports structure, i.e. `/usr/ports/???/eterm`.
- Do not combine several separately distributed programs/libraries into one package. Make several packages instead.

3.4.2. Directories

- In general packages should install files in these directories. Exceptions are of course allowed if there is a good reason. But try to follow the following directory structure as close as possible.

Directory	Description
`/usr/bin/`	User command/application binaries
`/usr/sbin/`	System binaries (e.g. daemons)
`/usr/lib/`	Libraries
`/usr/include/`	Header files
`/usr/lib/<prog>/`	Plug-ins, addons, etc
`/usr/man/`	Man pages
`/usr/share/<prog>/`	Data files
`/usr/etc/<prog>/`	Configuration files
`/etc/`	Configuration files for system software (daemons, etc)

- `/usr/X11R6` and `/usr/???/X11` are reserved for XFree86 only. X related programs that are not shipped with XFree86 should be placed under `/usr` and NOT under `/usr/X11R6` or `/usr/???/X11`.
- `/opt` directory is reserved for manually compiled/installed applications. Packages should never place anything there.
- `/usr/libexec/` is *not* used in CRUX, thus packages should never install anything there. Use `/usr/lib/<prog>/` instead.

3.4.3. Remove Junk Files

- Packages should not contain “junk files”. This includes info pages and other online documentation, man pages excluded (e.g. `usr/doc/*`, `README`, `*.info`, `*.html`, etc).
- Files related to NLS (national language support), always use **`--disable-nls`** when available.
- Useless or obsolete binaries (e.g. `/usr/games/banner` and `/sbin/mkfs.minix`).

3.4.4. Pkgfile

- Do not add new variables to the `Pkgfile`. Only in very few cases does this actually improve the readability or the quality of the package. Further, the only variables that are guranteed to work with future versions of **`pkgmk`** are `name`, `version`, `release`, and `source`. Other names could be in conflict with internal variables in **`pkgmk`**.
- Use the `$name` and `$version` variables to make the package easier to update/maintain. For example,

`source=(http://xyz.org/$name-$version.tar.gz)` is better than `source=(http://xyz.org/myprog-1.0.3.tar.gz)` since the URL will automatically updated when you modify the `$version` variable.

- Remember that source is an array, i.e. always do `source=(...)` and not `source=...`

Chapter 4. The Ports System

4.1. Introduction

4.1.1. What is a Port?

A port is a directory containing the files needed for building a package using **pkgmk**. This means that this directory at least has the files `Pkgfile` (which is the package build description) and `.footprint` (which is used for regression testing and contains a list of files this package is expected to contain once it is built). Further, a port directory can contain patches and/or other files needed for building the package. It is important to understand that the actual source code for the package is not necessarily present in port directory. Instead the `Pkgfile` contains an URL which points to a location where the source can be downloaded.

The use of the word *port* in this context is borrowed from the BSD world, where a *port* refers to a program that has been ported to a system or platform. The word can sometimes be a bit misleading since most programs require no actual porting to run on CRUX (or on Linux in general).

4.1.2. What is the Ports System?

The term *Ports System* refers to a CVS repository containing *ports* and a client program capable of downloading *ports* from that CVS repository. CRUX users use the **ports(8)** utility to download ports from the CVS repositorty and place them in `/usr/ports/`. The **ports** utility uses **CVSup(1) [http://www.cvsup.org/]** to do the actual downloading/synchronization.

4.2. Using the Ports System

4.2.1. Synchronizing Your Local Ports Structure

When CRUX is installed for the first time the local ports structure (`/usr/ports/`) is empty. To bring your local ports structure up to date you use the **ports** utility with the **-u** option. Example:

```
$ ports -u
```

The **-u** option means *update*, and tells **ports** to contact the ports CVS repository and download new and updated ports. The output from this execution is something like this:

```
Connected to cvsup.fukt.bth.se
Updating collection base/cvs
...
Updating collection opt/cvs
...
Finished successfully
```

The output reveals which files are downladed, updated and deleted.

4.2.2. Listing Local Ports

When the local ports structure has been updated the directory `/usr/ports/` will contain two package categories, `base` and `opt`. Under each of these directories you will find ports. You can simply browse around in the directory structure to find out which ports are available.

```
$ cd /usr/ports/base/
$ ls
autoconf/   filesystem/  man/            sh-utils/
automake/   fileutils/   man-pages/      shadow/
bash/       findutils/   modutils/       sysklogd/
bin86/      flex/        nasm/           sysvinit/
binutils/   gawk/        ncurses/        tar/
bison/      gcc/         net-tools/      tcp_wrappers/
bsdinit/    glibc/       netkit-base/    tcsh/
bzip2/      grep/        patch/          textutils/
cpio/       groff/       perl/           time/
db/         gzip/        pkgutils/       traceroute/
```

```
dcron/      kbd/         procps/         util-linux/
devfsd/     less/        psmisc/         vim/
diffutils/  libtool/     readline/       wget/
e2fsprogs/  lilo/        reiserfsprogs/  which/
ed/         m4/          sed/            zlib/
file/       make/        sendmail/
```

You can also use **ports** with the **-l** option to list all local ports. Example:

```
$ ports -l
base/autoconf
base/automake
base/bash
base/bin86
base/binutils
base/bison
...
opt/xfree86
opt/xmms
```

If you are looking for a specific package it might be easier to use this approach (e.g. **ports -l | grep sendmail**) to find out if the package is available and if so in which category it is located.

4.2.3. Listing Version Differences

To find out if the ports structure carries ports that are different (likely newer) compared to the versions currently installed you can use the option **-d**. If version differences are found, the output from the above command could look something like this:

```
$ ports -d
Collection  Name     Port     Installed
base        glibc    2.2.5-1  2.2.4-2
opt         xfree86  4.2.0-1  4.1.0-2
```

If no version differences were found, i.e. the system is in sync with the ports structure. Then output will simply be:

```
$ ports -d
No differences found
```

4.2.4. Building and Installing Packages

Once you have found a port that you want to build and install you simply go into the desired port directory and use **pkgmk** to build it. Example:

```
$ cd /usr/ports/base/sendmail
$ pkgmk -d
```

The **-d** option means *download missing source files* and tells **pkgmk** to download the source(s) specified in the `Pkgfile` (in case the source is already downloaded this option is ignored). When the download is completed the package will be built. If the package was built successfully you can use **pkgadd** to install or upgrade it. Example:

```
$ pkgadd sendmail#8.11.6-2.pkg.tar.gz
```

To make life a bit easier these two steps can be made into one by using the options **-i** (for install) or **-u** (for upgrade). Example:

```
$ pkgmk -d -i
```

or

```
$ pkgmk -d -u
```

This will download, build and then install/upgrade the package. Note that the package will only be installed/up-

graded if the build is successful.

Chapter 5. Configuration

5.1. Initialization Scripts

5.1.1. Runlevels

The following runlevels are used in CRUX (defined in `/etc/inittab`).

Runlevel	Description
0	Halt
1 (S)	Single-user Mode
2	Multi-user Mode
3-5	(Not used)
6	Reboot

5.1.2. Layout

The initialization scripts used in CRUX follow the BSD-style (as opposed to the SysV-style) and have the following layout.

File	Description
`/etc/rc`	System boot script
`/etc/rc.single`	Single-user startup script
`/etc/rc.modules`	Module initialization script
`/etc/rc.multi`	Multi-user startup script
`/etc/rc.local`	Local multi-user startup script (empty by default)
`/etc/rc.shutdown`	System shutdown script
`/etc/rc.conf`	System configuration
`/etc/rc.d/`	Service start/stop script directory

Modify `/etc/rc.modules`, `/etc/rc.local` and `/etc/rc.conf` according to your needs.

5.1.3. Configuration Variables in `/etc/rc.conf`

The following configuration variables are found in `/etc/rc.conf`.

Variable	Description
`KEYMAP`	Specifies which keyboard map to load at system startup. The contents of this variable will be passed as argument to **`loadkeys(1)`**. The available keyboard maps are located in `/usr/share/kbd/keymaps/`. Example: `KEYMAP=sv-latin1`
`TIMEZONE`	Specifies the timezone used by the system. The available zone description files are located in `/usr/share/zoneinfo/`. Example: `TIMEZONE=Europe/Stockholm`
`HOSTNAME`	Specifies the hostname.

Variable	Description
	Example: `HOSTNAME=pluto`
`SERVICES`	Specifies which services to start at system startup. The services specified in this array must have a matching start/stop script in `/etc/rc.d/`. When entering multi-user mode the specified scripts will be called in the specified order with the argument **`start`**. At system shutdown or when entering single-user mode these scripts will be called in the reverse order with the argument **`stop`**. Example: `SERVICES=(crond identd sshd sendmail)`

5.1.4. Network Configuration

The network configuration is found in the service script `/etc/rc.d/net`. To enable this service you need to add `net` to the `SERVICES` array in `/etc/rc.conf`. By default this service script only configures the `lo` device, you have to add additional **`ifconfig(8)`** and **`route(8)`** commands if you want to setup other network devices (`eth0`, `eth1`, etc). Example:

```
#!/bin/sh
#
# /etc/rc.d/net: start/stop network
#

if [ "$1" = "start" ]; then
    /sbin/ifconfig lo 127.0.0.1
    /sbin/ifconfig eth0 195.38.1.140 netmask 255.255.255.224
    /sbin/ifconfig eth1 192.168.0.1 netmask 255.255.255.0
    /sbin/route add default gw 195.38.1.129
elif [ "$1" = "stop" ]; then
    /sbin/ifconfig eth1 down
    /sbin/ifconfig eth0 down
    /sbin/ifconfig lo down
else
    echo "usage: $0 start|stop"
fi

# End of file
```

If you want to configure your system to be a DHCP client you use the **`dhcpcd(8)`** command (istead of **ifconfig(8)**). Example:

```
#!/bin/sh
#
# /etc/rc.d/net: start/stop network
#

if [ "$1" = "start" ]; then
    /sbin/ifconfig lo 127.0.0.1
    /sbin/dhcpcd eth0 [add additional options if needed]
elif [ "$1" = "stop" ]; then
    killall -q /sbin/dhcpcd
    /sbin/ifconfig lo down
else
    echo "usage: $0 start|stop"
fi

# End of file
```

5.2. Passwords

CRUX uses MD5SUM passwords by default. This can be turned off if you instead want to use the traditional DES passwords. Note however that DES passwords are considered less secure. To disable MD5SUM passwords

change the `MD5_CRYPT_ENAB` variable in `/etc/login.defs` to `no`.

Further, when compiling programs that use the `crypt(3)` function to authenticate users you should make sure that these programs are linked against the `libcrypt` library (i.e. use **`-lcrypt`** when linking) which contains the MD5SUM version of the `crypt` function (this version is backwards compatible and understands DES passwords as well).

5.3. Upgrading the Kernel

The kernel source, which is found in `/usr/src/linux/` is not installed using **`pkgadd`**. If you decide to upgrade your kernel you can safely do so by manually replacing the kernel source with a newer version (or place it somewhere else). This will not make the package database inconsistent (since it's not installed with **`pkgadd`**) nor will it affect the kernel headers found in `/usr/include/linux` and `/usr/include/asm` since these are not symlinks to the kernel source, but instead contain copies of the headers.

Chapter 6. Frequently Asked Questions

6.1. General

1. **Why the name “CRUX”?**

   Well, if you look it up in a dictionary you'll find something like “the basic, central, or critical point or feature”, but the word CRUX also has a UNIX/Linux-ish sound to it.... and I guess that's why I chose it.

2. **When will the next version be released?**

   Well, the standard answer to this question is “when it's done”. New versions are however usually released every 3 or 4 months. Between releases, updated packages are made available through the ports system.

6.2. Installation

1. **Will CRUX work with AMD K6/K6-II/K6-III?**

   Yes and No. AMD K6, K6-II and K6-III have an i586 (Pentium) compatible instruction set. Packages on the official CRUX ISO are compiled with **`-march=i686`**, which means that CRUX requires a processor which has an i686 compatible instruction set (i.e. Intel PPro/Celeron/PII/PIII/P4 or AMD K7/Athlon). However, Jürgen Daubert maintains an i586 version of the CRUX ISO image which can be found here [ftp://ftp.fukt.bth.se/pub/os/linux/crux/latest/contrib/]. The i586 version of the CRUX ISO works on AMD K6/K6-II/K6-III.

2. **When booting from the CRUX CD-ROM I get a kernel panic saying “VFS: Unable to mount root fs”. What's wrong?**

   This can happen if you have more than one CD-ROM drive. Make sure you boot from then "first" CD-ROM drive, i.e. `/dev/cdroms/cdrom0`. If you must boot from a different drive (i.e. not the first one) you can still do that but you have to type **`CRUX root=/dev/cdroms/cdrom1`** at the boot prompt (**`cdrom1`** indicates that it is the second drive, **`cdrom2`** that it is the third, and so on).

3. **When booting CRUX for the first time I get the error “Unable to open initial console”. What's wrong?**

   You most likely forgot to enable devfs or didn't tell the kernel to mount devfs at boot. The installation instructions (Section 2.2) tell you how to enable it.

4. **When logging in to my newly installed CRUX for the first time it asks for a password, but the installation guide says “Login as root (no password required)”. What's wrong?**

   You most likely forgot to edit `/mnt/etc/fstab` before you rebooted or you entered the wrong name of your new root parition at the boot prompt.

6.3. Configuration

1. **Why are changes made to files under `/dev` lost the next time I boot into CRUX?**

CRUX uses `devfs`, which is a virtual filsystem kept in RAM. Changes made to files under `/dev` are always lost when you turn off the power. However, you can configure **`devfsd(8)`** to restore them when you boot. You need to modify `/etc/devfsd.conf` according to your needs. See the **`devfsd(8)`** man-page for details. Example:

```
#
# /etc/devfsd.conf: devfsd(8) configuration
#

REGISTER        .*              MKOLDCOMPAT
UNREGISTER      .*              RMOLDCOMPAT

LOOKUP          .*              MODLOAD

REGISTER        ^sound/.*       PERMISSIONS root.users 660
REGISTER        ^v4l/.*         PERMISSIONS root.users 660

# End of file
```

2. **How do I get sshd running?**

   You have to edit `/etc/hosts.deny` and/or `/ets/hosts.allow` to specify which hosts are allowed/denied access. To allow anyone to connect to your machine you can add `sshd: ALL` to `/etc/hosts.allow`. See the `hosts_access(5)` man-page for further information about the file format. When this is done you can start sshd by entering the command **`/etc/rc.d/sshd start`** and/or edit `/etc/rc.conf` and add `sshd` to the `SERVICES` array, i.e. `SERVICES=(... sshd ...)`, which means that sshd will be started when the system boots.

3. **Mozilla crashes or refuses to start, what's up?**

   Mozilla is extremely sensitive to missing `fonts.cache-1` files. If mozilla refuses to start (due to segmentation violation or just silent permature termination) it is most likely becuase the font cache files are missing. Run **`fc-cache`** (as **`root`**) to create/update the cache files. See the **`fc-cache`** man page for information about this program.