



Vrije Universiteit Brussel

FACULTY OF ENGINEERING

# Creating a Linux bootdisk with a customised kernel

---

Operating Systems and Security

Ward Van Heddeghem

---

January 2, 2008



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Report set-up	3
1.2	Required experience	3
1.3	System setup	3
<b>2</b>	<b>The Linux operating system</b>	<b>5</b>
2.1	Overview	5
2.2	Overview of the Linux boot process	5
2.2.1	Stage 1 boot loader	6
2.2.2	Stage 2 boot loader	7
2.2.3	Kernel	7
2.2.4	Init	8
2.3	Overview of a basic Linux root file system	8
<b>3</b>	<b>Compiling a kernel</b>	<b>9</b>
3.1	Overview	9
3.2	Preparation of your system	9
3.3	Configuration of the kernel	10
3.4	Creation of the kernel package	12
3.5	Installation of the kernel package	12
3.6	A word on GRUB	13
3.7	Removal of a kernel package	14
<b>4</b>	<b>Creating a customised kernel</b>	<b>15</b>
4.1	Overview	15
4.2	Loadable kernel modules	15
4.3	A batch script for creating multiple kernel packages	15
4.4	Making space on your file system	17
<b>5</b>	<b>Creating an initial RAM disk</b>	<b>18</b>
5.1	What's an initial RAM disk?	18
5.2	Anatomy of the initrd	18
5.2.1	Compressed cpio archive file	18
5.2.2	Loop device	19
5.3	Manually building a custom initial RAM disk	19
<b>6</b>	<b>Creating a floppy boot disk</b>	<b>22</b>

<b>7</b>	<b>Creating a CD-ROM boot disk</b>	<b>23</b>
7.1	GRUB . . . . .	23
7.2	Putting it all together . . . . .	23
7.3	Using Innotek Virtualbox . . . . .	24
7.4	Testing your bootable image . . . . .	25
7.5	Adding a boot menu to GRUB . . . . .	26
<b>8</b>	<b>Booting with an initial RAM disk</b>	<b>27</b>
<b>9</b>	<b>Conclusion and further work</b>	<b>28</b>

# 1 Introduction

This report describes how to create a minimal Linux operating system. This document has been created as part of a project assignment for the Operating Systems & Security course taught at the VUB (2007).

The goal of this project was to get hands-on experience with the act of compiling a Linux kernel and the creation of a bootable, trimmed-down Linux operating system which is small enough to fit on a floppy disk (1,44 MB), from here on referred to as **boot disk**.<sup>1</sup> This should give a better insight on

- the file structure of a Linux distribution,
- the core components of the Linux operating system,
- the boot process,
- and an better overall understanding of Linux.

Since so-called HOW-TO documentation on both these subjects (kernel compiling and bootdisk creation) is available on the internet, this project is by no means a pioneering effort. However, since the available documentation is often written for a specific Linux distribution and is rather outdated (the most recent HOW-TOs date from 2003), and given my non-existing experience with kernel compilation, this task still provided to be interesting and challenging enough to make it an educative experience.

## 1.1 Report set-up

The result of this project is this document. It is a syntheses of information available on the Internet, and a lot of experimentation to get this information ported to the Ubuntu 7.10 platform; the Linux distribution chosen for this project.

Chapter 2 starts with a brief overview of Linux, the Linux boot process, and the Linux file system. All subsequent chapters have been set up as a walkthrough, providing the reader with detailed instructions on how to create a minimal boot disk himself. Chapter 3 details how to compile a kernel. Chapter 4 shows how to decrease the kernel size. Chapter 5 instructs on how to create an initial RAM disk. Chapter 7 discusses how to put everything together to have a bootable CD-ROM. And chapter 8 finally goes into the details of the boot sequence of the initial RAM disk.

## 1.2 Required experience

This report assumes that you know what Linux is, what a kernel is and that you are familiar with the terminal and basic Linux commands such as `ls`, `mount`, etc.

## 1.3 System setup

All of the instructions described on in this report have been performed on **Ubuntu 7.10**, a relatively new Linux distribution which is becoming increasingly popular<sup>2</sup>.

I have used the template directory structure shown in figure 1 for this project, and it will be used throughout this report. It is advised to the reader who wants to follow along, to set up the same directories.

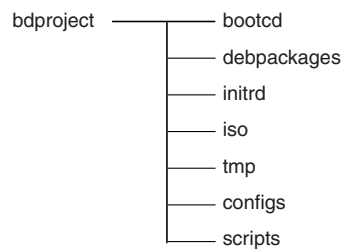
---

<sup>1</sup>Because of time constraints, the goal of creating a boot floppy disk had to be abandoned, and has been replaced by creating a bootable CD-ROM. See section 6.

<sup>2</sup>According to [www.distrowatch.com](http://www.distrowatch.com), it currently (December 2007) ranks in the top 2 of Linux distribution.

Description of the folders:

- `bdproject` - root folder of the project
- `bootcd` - will contain the contents of the boot disk
- `configs` - will contain the `.config` kernel configuration files
- `debpackages` - will contain the Debian kernel package(s)
- `initrd` - will contain the contents of the initial RAM disk
- `iso` - will contain the `.iso` image file of the boot disk
- `scripts` - will contain a few scripts
- `tmp` - for general temporary files



**Figure 1:** Template directory structure

## 2 The Linux operating system

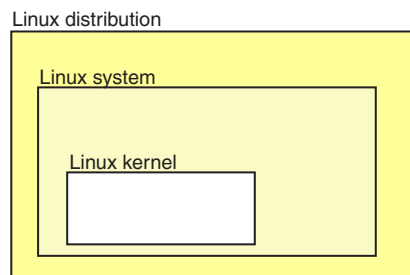
### 2.1 Overview

This section will very briefly look at the main components of Linux.

**Credit:** *The content of this section is an extract from “Operating System Concepts”[11].*

Linux is a version of UNIX that has gained popularity in recent years. In its early days, Linux development revolved largely around the central operating-system kernel - the core, privileged executive that manages all system resources and that interacts directly with the computer hardware. We need much more than this kernel to produce a full operating system of course. It is useful to make a distinction between the Linux kernel and a Linux system. The **Linux kernel** is an entirely original piece of software developed from scratch by the Linux community. The **Linux system**, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, a need has arisen for another layer of functionality on top of the linux system. This need has been met by various Linux distributions. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages in the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, web browsers, word processors and so on.



**Figure 2:** Core components of a Linux distribution

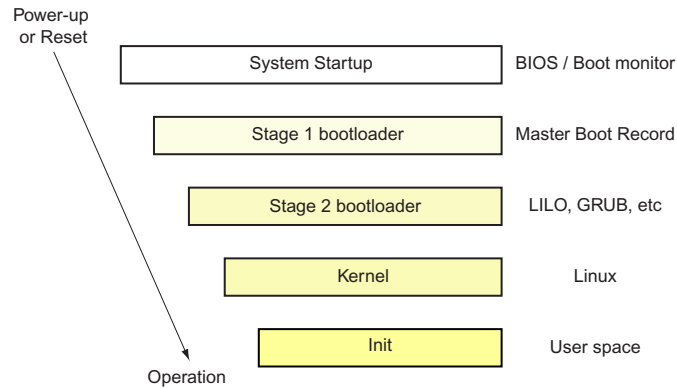
### 2.2 Overview of the Linux boot process

**Credit:** *The following text is a summary of the article “Inside the Linux boot process”[6].*

When a system is first booted, or is reset, the processor executes code at a well-known location. In a personal computer (PC), this location is in the basic input/output system (BIOS), which is stored in flash memory on the motherboard. Because PCs offer so much flexibility, the BIOS must determine which devices are candidates for boot.

When a boot device is found, the **first-stage boot loader** is loaded into RAM and executed. This boot loader is less than 512 bytes in length (a single sector), and its job is to load the second-stage boot loader.

When the **second-stage boot loader** is in RAM and executing, a splash screen is commonly displayed, and the Linux **kernel** and an optional **initial RAM disk** (temporary root file system) are loaded into memory. When the images are loaded, the second-stage boot loader passes control to the kernel and the kernel is decompressed and initialised. At this stage, the kernel checks the system hardware, enumerates the attached hardware devices, mounts the root device, and then loads the necessary kernel

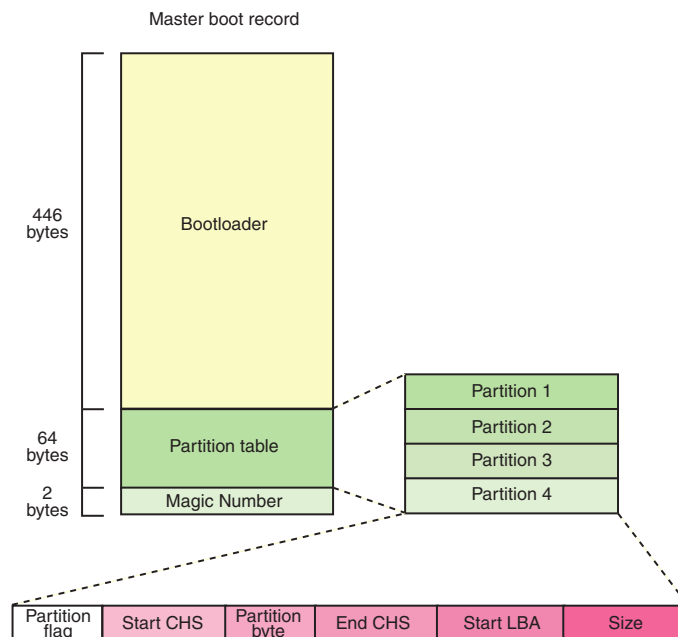


**Figure 3:** The 20,000-foot view of the Linux boot process

modules. When complete, the first user-space program (**init**) starts, and high-level system initialisation is performed.

That's Linux boot in a nutshell. Now let's dig in a little further and explore some of the details of the Linux boot process.

### 2.2.1 Stage 1 boot loader



**Figure 4:** Anatomy of the MBR

The primary boot loader that resides in the MBR is a 512-byte image containing both program code and a small partition table (see figure 4). The first 446 bytes are the primary boot loader, which contains both executable code and error message text. The next sixty-four bytes are the partition table, which contains a record for each of four partitions (sixteen bytes each). The MBR ends with two bytes that are defined as the magic number (0xAA55). The magic number serves as a validation check of the MBR.

The job of the primary boot loader is to find and load the secondary boot loader (stage 2). It does this by looking through the partition table for an active partition. When it finds an active partition, it scans the remaining partitions in the table to ensure that they're all inactive. When this is verified, the active partition's boot record is read from the device into RAM and executed.

### 2.2.2 Stage 2 boot loader

The secondary, or second-stage, boot loader could be more aptly called the kernel loader. The task at this stage is to load the Linux kernel and optional initial RAM disk.

The first- and second-stage boot loaders combined are called Linux Loader (LILO) or GRand Unified Bootloader (**GRUB**) in the x86 PC environment. Because LILO has some disadvantages that were corrected in GRUB, let's look into GRUB.

The great thing about GRUB is that it includes knowledge of Linux file systems. Instead of using raw sectors on the disk, as LILO does, GRUB can load a Linux kernel from an ext2 or ext3 file system. It does this by making the two-stage boot loader into a three-stage boot loader. Stage 1 (MBR) boots a stage 1.5 boot loader that understands the particular file system containing the Linux kernel image. Examples include `reiserfs_stage1_5` (to load from a Reiser journaling file system) or `e2fs_stage1_5` (to load from an ext2 or ext3 file system). When the stage 1.5 boot loader is loaded and running, the stage 2 boot loader can be loaded.

The `/boot/grub` directory contains the stage1, stage1.5, and stage2 boot loaders, as well as a number of alternate loaders (for example, CD-ROMs may use the `iso9660_stage_1_5`).

With stage 2 loaded, GRUB can, upon request, display a list of available kernels (this list is defined in `/boot/grub/menu.lst`). You can select a kernel and even amend it with additional kernel parameters. Optionally, you can use a command-line shell for greater manual control over the boot process.

With the second-stage boot loader in memory, the file system is consulted, and the default kernel image and `initrd` image are loaded into memory. With the images ready, the stage 2 boot loader invokes the kernel image.

### 2.2.3 Kernel

With the kernel image in memory and control given from the stage 2 boot loader, the kernel stage begins. The kernel image isn't so much an executable kernel, but a compressed kernel image. Typically this is a `zImage` (compressed image, less than 512KB) or a `bzImage` (big compressed image, greater than 512KB), that has been previously compressed with `zlib`. At the head of this kernel image is a routine that does some minimal amount of hardware setup and then decompresses the kernel contained within the kernel image and places it into high memory. If an initial RAM disk image is present, this routine moves it into memory and notes it for later use. The routine then calls the kernel and the kernel boot begins.

During the boot of the kernel, the initial-RAM disk (`initrd`) that was loaded into memory by the stage 2 boot loader is copied into RAM and mounted. This `initrd` serves as a temporary root file system in RAM and allows the kernel to fully boot without having to mount any physical disks. Since the necessary modules needed to interface with peripherals can be part of the `initrd`, the kernel can be very small, but still support a large number of possible hardware configurations. After the kernel is booted, the root file system is pivoted where the `initrd` root file system is unmounted and the real root file system is mounted.

The `initrd` function allows you to create a small Linux kernel with drivers compiled as loadable modules. These loadable modules give the kernel the means to access disks and the file systems on those disks, as well as drivers for other hardware assets. Because the root file system is a file system on a disk, the `initrd` function provides a means of bootstrapping to gain access to the disk and mount the real root file system. In an embedded target without a hard disk, the `initrd` can be the final root file system, or the final root file system can be mounted via the Network File System (NFS).



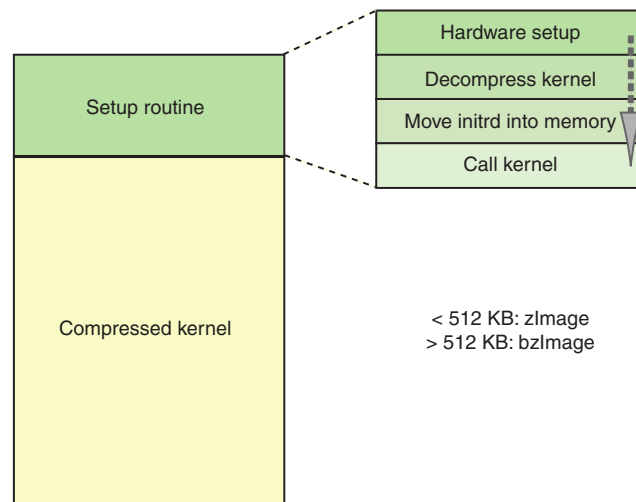


Figure 5: Kernel structure

### 2.2.4 Init

After the kernel is booted and initialised, the kernel starts the first user-space application. This is the first program invoked that is compiled with the standard C library. Prior to this point in the process, no standard C applications have been executed.

In a desktop Linux system, the first application started is commonly `/sbin/init`. But it need not be. Rarely do embedded systems require the extensive initialisation provided by `init` (as configured through `/etc/inittab`). In many cases, you can invoke a simple shell script that starts the necessary embedded applications.<sup>3</sup>

## 2.3 Overview of a basic Linux root file system

**Credit:** *This chapter is based on the "The Linux Bootdisk HOW-TO" [7].*

Since the standard Linux directory tree is extensive and not all parts are completely relevant for this project, I will limit this section to an overview of a minimum set of directories needed to support a bare Linux system. For a more detailed discussion on the standard Linux directory tree, see [9].

The following is a reasonable minimum set of directories for the root file system:

- `/dev` - Device files, required to perform I/O
- `/proc` - Directory stub required by the `proc` file system (information about processes)
- `/sys` - Pseudo file system, providing devices and driver info from kernel to userspace
- `/etc` - System configuration files
- `/sbin` - Critical system binaries
- `/bin` - Essential binaries considered part of the system
- `/lib` - Shared libraries to provide run-time support
- `/usr` - Additional utilities and applications

<sup>3</sup>Ubuntu, from version 6.10 and onward, no longer uses `init` to manage its services during start-up or shutdown of the system. Instead, it uses 'upstart' as an event-based replacement for the traditional `sysvinit` utility that is common to Linux-based operating system. See <http://upstart.ubuntu.com/>

## 3 Compiling a kernel

### 3.1 Overview

The default approach to compiling a kernel is using the commands shown in listing 1.

**Listing 1:** Default approach to kernel compiling

```
# make dep
# make clean
# make bzImage
# make modules
# make modules_install
# make install
```

We will *not* go down this route, since Debian-based distributions, such as Ubuntu 7.10, provide the `make-kpkg` command (make kernel package), which simplifies kernel compilation and installation. The result of this will be a Debian installation package (extension `.deb`) which contains our new kernel and all necessary support files (modules, `initrd`, etc.). This installation package can then easily be installed through the command `dpkg -i` (Debian package, installation).

The general steps to be taken for creating and installing a customised kernel are as follows:

1. preparation of your system (installation of required software, and kernel source)
2. configuration of the kernel
3. creation of the kernel package, this comprises kernel compilation and (optionally) module compilation
4. installation of the kernel package

In the following sections, we will look into the details of each step.

### 3.2 Preparation of your system

**Credit:** *The following text has been largely taken (and slightly adapted) from the article "How to Customise your Ubuntu kernel" [5].*

A default Ubuntu 7.10 installation does not come installed with all software that is required for kernel compilation. Missing parts include the Linux kernel source code.

However, before you can start, you need to figure out what kernel version you are currently running. To do so, you can use the command `uname -r`, which prints system information (the `-r` option prints only the kernel release):

**Listing 2:** Looking up the current kernel version

```
\$ uname -r
2.6.22-14-generic
\$ uname -a
Linux mango 2.6.22-14-generic #1 SMP Sun Oct 14 23:05:12 GMT 2007 i686 GNU/Linux
```

The listing 2 shows that I am running the 2.6.22-14 kernel. In the next commands, you have to substitute your kernel number for whatever kernel number you are running.

Now you need to install the linux source for your kernel. You also need to install the `ncurses` library and some other tools to help you compile. The `ncurses` library provides a set of subroutines for handling navigation on a terminal screen using the cursors.

Before doing so, make sure that you have a working connection to the internet and that downloading source code has been checked (System > Administration > Software Sources \ Ubuntu Software).

### Listing 3: Installing the required packages

```
$ sudo apt-get install linux-source-2.6.22
$ sudo apt-get install kernel-package
$ sudo apt-get install libncurses5-dev
$ sudo apt-get install fakeroot
```

If you are curious where the Linux source gets installed to, you can use the `dpkg -L` command to tell you the files within a package:

### Listing 4: Finding out package installation folders

```
$ dpkg -L linux-source-2.6.22
/.
/usr
/usr/src
/usr/src/linux-source-2.6.22.tar.bz2
/usr/share
[trimmed]
```

You can see that the source has been installed to the `/usr/src` directory in a zipped file.

To make things easier, we'll put ourselves in root mode by using `sudo` to open a new shell.

### Listing 5: Opening a root shell

```
$ sudo /bin/bash
```

Now change directory into the source location so that you can install.

### Listing 6: Extracting the kernel source

```
# cd /usr/src
# bunzip2 linux-source-2.6.22.tar.bz2
# tar xvf linux-source-2.6.22.tar
# ln -s linux-source-2.6.22 linux
```

And finally, make a copy of your existing kernel configuration to use for the custom compile process.

### Listing 7: Copying the existing kernel configuration file

```
# cp /boot/config-2.6.22.14-generic /usr/src/linux/.config
```

You are now ready to customise your kernel configuration file.

## 3.3 Configuration of the kernel

**Credit:** *The following text has been largely taken (and slightly adapted) from the article "How to Customise your Ubuntu kernel" [5].*

The configuration of the kernel has to be done by altering the file `.config` which is located under `/usr/src/linux`. There are several ways to do this. We will use the command `make menuconfig`, which has to be executed while in `/usr/src/linux`. When we have run the mentioned command, and altered the kernel configuration, the result will be that the `.config` has been updated. This file will then later be used as an input for the compilation process.

Navigate to the linux source folder and launch the utility (`make menuconfig`) that will let you customise the kernel:

### Listing 8: Launching the configuration utility

```
# cd /usr/src/linux
# make menuconfig
```

```

Cryptographic options --->
Library routines --->
---
Load an Alternate Configuration File
Save Configuration to an Alternate File

```

**Figure 6:** Loading an alternate configuration file

First, go down to Load an Alternate Configuration File, and load the .config file. (just hit enter)

Now that you are inside the utility, you can set the options for your custom kernel. Navigation is pretty simple, there's a legend at the top if you get lost. For example, select Networking and hit the Enter key to go down into that category. See 7.

```

Code maturity level options --->
General setup --->
Loadable module support --->
Block layer --->
Processor type and features --->
Power management options (ACPI, APM) --->
Bus options (PCI, PCMCIA, EISA, MCA, ISA) --->
Executable file formats --->
Networking --->
Device Drivers --->
File systems --->
Instrumentation Support --->
Kernel hacking --->
Security options --->
Cryptographic options --->
Library routines --->
---
Load an Alternate Configuration File
Save Configuration to an Alternate File

```

**Figure 7:** Selecting Networking

You will now get a list of available support for your kernel. See figure 8. The symbol to the left of the line indicates whether the support will be built-in to the kernel (indicated by an [\*]), whether it will be available as a kernel module (indicated by an [M]), or whether it will not be available at all (indicated by [ ]). For example, in figure 8, Amateur Radio Support will be part of the kernel, and thus increase the kernel size. IrDA (infrared) subsystem support will be available as a kernel module, having no impact on the kernel size. An arrow (--->) to the right of the line indicates a submenu. If a checkbox is present on such a line, you can disable support for all entries in the subsystem.

```

--- Networking support
Networking options --->
[*] Amateur Radio support --->
<M> IrDA (infrared) subsystem support --->
<M> Bluetooth subsystem support --->
<M> Generic IEEE 802.11 Networking Stack
[ ] Enable full debugging output
--- IEEE 802.11 WEP encryption (802.1x)

```

**Figure 8:** Amateur radio support

By pressing the h or ? key, you can see the help for that particular item. See figure 9.

```
CONFIG_HAMRADIO:

If you want to connect your Linux box to an amateur radio, answer Y
here. You want to read <http://www.tapr.org/tapr/html/pkthome.html> and
the AX25-HOWTO, available from <http://www.tldp.org/docs.html#howto>.

Note that the answer to this question won't directly affect the
kernel: saying N will just cause the configurator to skip all
the questions about amateur radio.

Symbol: HAMRADIO [=y]
Prompt: Amateur Radio support
Defined at net/ax25/Kconfig:9
Depends on: NET
Location:
-> Networking
-> Networking support (NET [=y])
```

Figure 9: Help on “Amateur Radio support”

Hit Esc to exit the help screen, and then hit N to exclude amateur radio support from your kernel.

When you are finished making whatever choices you want, hit Exit and save the configuration when prompted.

Now you have a configuration ready for compile.

### 3.4 Creation of the kernel package

**Credit:** *The following text has been largely taken (and slightly adapted) from the article “How to Customise your Ubuntu kernel” [5].*

Before starting kernel compilation, you have to execute `make-kpkg clean` to make sure that your system is ready for the compile. This will remove all files in the kernel source directory created during a previous package creation/kernel build. Once this has been done, we can actually compile the kernel and create the kernel package using the command `make-kpkg` together with the appropriate options.

**NOTE:** *The kernel package creation may take a long time, depending on your system and the selected configuration. As an example, my initial kernel compilation took roughly 3 hours (on an Athlon XP 2000) and 1.5 hours under Parallels on a MacBook.*

Listing 9: Kernel package creation

```
# make-kpkg clean
# fakeroot make-kpkg -initrd -append-to-version=-customt1 kernel_image
kernel_headers
```

This process will create two .deb files in `/usr/src` that contain the kernel. The `linux-image****` file is the actual kernel image, and the other file (`linux-headers****`) contains the headers included in the kernel.

The `fakeroot` fakes root privileges for file manipulations. It is not strictly necessary to use it when you are already logged in as root, but it would be usefull if you had made yourself part of the `src` group. The `-initrd` option will make sure the kernel supports initial RAM disk loading (more on this later). The `-append-to-version` allows to specify a string to appended to the kernel name, in our case “customt1” (“t1” stands for “test 1”). The targets `kernel_image` and `kernel_headers` tell the `make-kpkg` command to produce a kernel package and a headers package.

### 3.5 Installation of the kernel package

**Credit:** *The following text has been largely taken (and slightly adapted) from the article “How to Customise your Ubuntu kernel” [5].*

You can install both created .deb files with the command `dpkg`. The filenames will probably be different on your system.

**NOTE:** Please note that when you run these next commands, this will set the new kernel as the new default kernel. This could break things! If your machine doesn't boot, you can hit `Esc` at the GRUB loading menu, and select your old kernel. You can then disable the kernel in `/boot/grub/menu.lst` or try and compile again.

**Listing 10:** Installation of the kernel packages

```
# dpkg -i linux-image-2.6.22.9-customt1_2.6.22.9-customt1-10.00.Custom_i386.deb
# dpkg -i linux-headers-2.6.22.9-customt1_2.6.22.9-customt1-10.00.Custom_i386.deb
```

You can now reboot your machine. If everything works, you should be running your new custom kernel. You can check this by using `uname`. The exact number might be different on your machine.

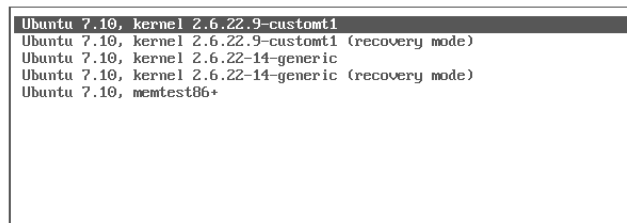
**Listing 11:** Checking the kernel version packages

```
$ uname -r
$ 2.6.17.14-customt1
```

The kernel image can now be found in the `/boot` directory. It will be named `vmlinuz-2.6.22.9-customt1`. As you can see, the custom string that you have specified with the `make-kpkg` command is appended to the kernel file name. In the same directory you will also find the initial RAM disk (`initrd.img-2.6.22.9-customt1`) and your kernel configuration file (`config-2.6.22.9-customt1`).

### 3.6 A word on GRUB

If your Ubuntu 7.10 OS is installed as part of a dual-boot configuration, by installing the kernel package an entry will have been added to the GRUB boot menu. This allows you to choose whether you want to boot either your original kernel, or the kernel you have just installed. See figure 10.



Use the `↑` and `↓` keys to select which entry is highlighted. Press enter to boot the selected OS, 'e' to edit the commands before booting, or 'c' for a command-line.

**Figure 10:** Choosing which kernel to boot in the GRUB menu

However, if Ubuntu 7.10 is installed as a standalone OS, or if you are experimenting inside a virtual machine, then the GRUB menu is not visible by default. To enable the GRUB boot menu open GRUB's `menu.lst` file for editing.

**Listing 12:** Opening the GRUB menu.lst file

```
$ cd /boot/grub
$ sudo gedit menu.lst
```

With `menu.lst` open, comment out the line `hiddenmenu` by preceding it with a `#`. Also, increase the timeout to a more reasonable value, like 10 seconds. The result is visible in listing 13.

**Listing 13:** Editing the GRUB menu.lst file to show the menu

```
...
## timeout sec
# Set a timeout, in SEC seconds, before automatically booting the default entry
# (normally the first entry defined).
timeout          10

## hiddenmenu
# Hides the menu by default (press ESC to see the menu)
# hiddenmenu
...
```

### 3.7 Removal of a kernel package

The advantage of creating kernel packages with `make-kpkg`, as opposed to compiling the kernel using the `make` commands mentioned in chapter 3.1, is that it is very easy to install and uninstall kernels and all related files (modules etc.).

To uninstall a kernel package, you have two options:

- The command `apt-get remove`, followed by the package name. `apt-get remove` will leave configuration files for the package on your system. A configuration file is defined as any file you might have edited in order to customise the program for your system or your preferences. This way, if you later reinstall the package, you won't have to set everything up a second time.
- However, you might want to erase the configuration files too, so `apt-get` also provides a `purge` option. `apt-get purge` will permanently delete every last file associated with the specified package.

**Listing 14:** Removal of a kernel package

```
$ sudo apt-get remove linux-image-2.6.22.9-customt1
$ sudo apt-get purge linux-image-2.6.22.9-customt1
```

**NOTE:** *Debian based Linux distributions provide a couple of tools to manage packages. We have used `dpkg` to install or kernel package in chapter 3.5. We have used `apt-get` to remove a package. We could also have used the command `dpkg` to remove packages. For more information on the Debian package management system and the different tools, see [2] and [12].*

## 4 Creating a customised kernel

### 4.1 Overview

The previous chapter showed the procedure of creating a new kernel with - by way of an example - amateur radio support removed. For the purpose of this project, the goal is of course to remove as much as possible from the kernel in order to make the kernel size small enough to fit on a floppy disk. Since the floppy disk will not only contain the kernel image, but also some additional files, the kernel size must of course be smaller than 1.44 MB. Just how small can only be calculated once we have created all the additional files (i.e., the initial RAM disk image and the GRUB files).

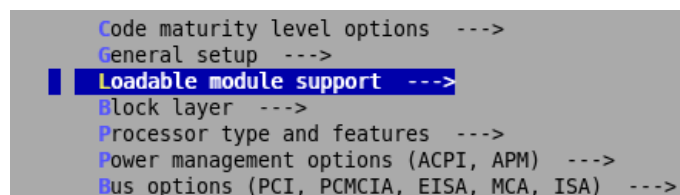
This chapter will provide some tricks and tools to aid in experimenting with kernel creation. To this end, since it can be tedious to experiment with kernel package creation because of the long waiting times, I will provide a script that you can run overnight. This script allows you to create multiple kernel packages in one go, based on several configuration files that you should prepare before starting the script.

You might also notice that creating the kernel packages can consume a lot of disk space. If you are on a small partition, at the end of this chapter I will give a few tips on cleaning up your disk.

### 4.2 Loadable kernel modules

Since Linux kernel version 1.2, **loadable kernel modules** are supported. Loadable kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Loadable kernel modules can be loaded automatically by the kernel when needed.

Support for kernel modules however, can be enabled or disabled in the kernel through the configuration file (option `Loadable module support --->`).



```
Code maturity level options --->
General setup --->
Loadable module support --->
Block layer --->
Processor type and features --->
Power management options (ACPI, APM) --->
Bus options (PCI, PCMCIA, EISA, MCA, ISA) --->
```

Figure 11: Loadable kernel modules support options

When support for loadable kernel modules is enabled, all items marked with [M] will be compiled as a module. When the Debian kernel package is installed, these modules are installed in the folder `/lib/modules/`.

When support for loadable kernel modules is disabled, all items marked with [M] will automatically toggle to [\*], resulting in an inclusion of these items in the kernel. Take note that this will increase the size of your kernel.

### 4.3 A batch script for creating multiple kernel packages

Depending on the speed of your system and the kernel configuration options selected, kernel package creation may take a lot of time. On my system, the first kernel package creation took me 3 hours! This can make it inconvenient to experiment with different configuration options. Therefore, I have created a simple bash script<sup>4</sup> named `mkmultikernel.sh` which allows to compile and create multiple kernel

<sup>4</sup>For a good introduction to bash scripting, see the article "Bash Shell Programming in Linux" [1].



packages in one go. You could start the script in the evening and have all your .deb packages ready in the morning.

**Listing 15:** Utility (mkmultikernel.sh) to create multiple kernel packages

```
#!/bin/bash
# Do a multiple kernel compile

# * Configure $CONFIGS_TO_PROCESS to contain the required config file suffixes
# * Config files should be located in $CONFIG_DIR
#   and files should be named configt1, configt2, configt3, ...
#   with t1, t2, ... the suffixes as specified in $CONFIGS_TO_PROCESS
# * The resulting debian packages (*.deb) will be moved to $STORE_DIR

#Check if root
if [ "$(id -u)" != "0" ]; then
    echo "This script must be run as root"
    exit 1
fi

#Init
CONFIGS_TO_PROCESS="t15 t16"           #config files suffixes, modify this !
CONFIG_DIR=/home/ward/bdproject/configs #config files location, modify this !
STORE_DIR=/home/ward/bdproject/debpackages #where to move the packages to, modify
this !
LOGFILE=/home/ward/bdproject/build.log  #location of log file, modify this !
DATEOUTPUT="date +%F%t%R:%S%t"

#Do all different compiles
for fn in $CONFIGS_TO_PROCESS; do

    echo " " >> ${LOGFILE}
    echo " " >> ${LOGFILE}
    echo "${DATEOUTPUT} Starting for $fn ..." >> ${LOGFILE}
    echo "${DATEOUTPUT} Starting for $fn ..."

    #Prepare and clean up
    echo "${DATEOUTPUT} Cleaning ..." >> ${LOGFILE}
    echo "${DATEOUTPUT} Cleaning ..."
    cd /usr/src/linux
    make-kpkg clean

    #Fetch correct config file
    echo "${DATEOUTPUT} Copy config file config$fn ..." >> ${LOGFILE}
    echo "${DATEOUTPUT} Copy config file config$fn ..."
    rm /usr/src/linux/.config
    cp $CONFIG_DIR/config$fn /usr/src/linux/.config

    #compile
    echo "${DATEOUTPUT} Compile ..." >> ${LOGFILE}
    echo "${DATEOUTPUT} Compile ..."
    fakeroot make-kpkg -initrd -append-to-version=-custom$fn kernel_image
    kernel_headers

    #show folder content after compile
    echo "${DATEOUTPUT} Folder content:" >> ${LOGFILE}
    echo "${DATEOUTPUT} Folder content:"
    cd /usr/src
    echo "$(ls -lh)" >> ${LOGFILE}

    #move to other location
    echo "${DATEOUTPUT} Move deb packages ..." >> ${LOGFILE}
    echo "${DATEOUTPUT} Move deb packages ..."
    mkdir $STORE_DIR/$fn
    mv /usr/src/*$fn*.deb $STORE_DIR/$fn

done

exit
```

Before you can run the script, you will have to create different `.config` files with your desired kernel configuration, as described in chapter 3. Each time you have created a `.config` file, copy it to the `configs` folder and rename it to `.configtx` where `x` can be any number, e.g. `.configt9`, `configt10`, `configt11`, etc.

You need to specify the location of the config files by modifying the variable `CONFIG_DIR` in the script. The config files to process have to be specified in the `CONFIGS_TO_PROCESS` variable. The created debian packages will be moved to the folder specified in `STORE_DIR`. And finally, a log file will be created as specified in `LOGFILE`. Make sure to adjust all four mentioned variables as required.

**NOTE:** *Since the size of the created kernel packages can be large, make sure you have enough room on the disk where the packages will be moved to. For some tips on freeing up space on your file system, see chapter 4.4.*

Save the content of the adjusted script in a file called `mkmultikernel.sh` and place it in a `scripts` folder. Make the file executable using the command `chmod` in the folder where you have saved the script. To start the script, enter `sudo ./mkmultikernel.sh`.

**Listing 16:** Starting the `mkmultikernel` script

```
$ cd ~/bdproject/scripts
$ chmod +x mkmultikernel.sh
$ sudo ./mkmultikernel.sh
```

The script will, for each of your config files, first clean up any garbage left over from a previous compilation, fetch one of your config files, use it to create a kernel package and move the package to the location you specified.

You can then install the desired kernel package as described in chapter 3.5.

## 4.4 Making space on your file system

Creating multiple kernels might consume quite some space on your file system. The following tips might help you to regain some of that space:

- **Remove the tar source file.** When installing the source files in section 3.2, the `linux-source-2.6.22.9.tar` file is not removed automatically. Since it can be quite large, you might want to remove it.
- **Uninstall kernel packages that you no longer require.** See chapter 3.7. Make sure to use the `purge` option.
- **Clear the local cache of your debian packages.** When you install software on your system, generally the appropriate installation packages are fetched from the Internet. Your system keeps a local copy of these installation packages, which might grow to a substantial amount after a certain amount of time. To clear this local cache, type `apt-get clean` at the command prompt. This clears out the local repository of retrieved package files. It removes everything but the lock file from `/var/cache/apt/archives/` and `/var/cache/apt/archives/partial/`.
- **Empty your 'deleted items'.** Of course, do not forget to empty your 'deleted items' folder on your desktop.

## 5 Creating an initial RAM disk

**Credit:** *The following text is largely taken from the excellent article “Linux initial RAM disk (initrd) overview” [8]. I have made some slight changes to make it fit this project.*

### 5.1 What’s an initial RAM disk?

The **initial RAM disk (initrd)** is an initial (temporary) root file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure. The kernel then mounts this initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.

The initrd contains a minimal set of directories and executables to achieve this, such as the `insmod` tool to install kernel modules into the kernel.

In the case of desktop or server Linux systems, the initrd is a transient file system. Its lifetime is short, only serving as a bridge to the real root file system. In embedded systems with no mutable storage, the initrd is the permanent root file system. The latter is the approach that we will use in this report.

### 5.2 Anatomy of the initrd

The initrd image contains the necessary executables and system files to support the second-stage boot of a Linux system.

The method for creating the initial RAM disk can vary. It can be constructed using the **loop device** or as a **compressed cpio archive file**. Since the initial RAM disk that is created by the `make-kpkg` used in chapter 3.4 is a compressed cpio archive file, we will first have a look at this. The initial RAM disk that we will create however, will use the loop device technique.

#### 5.2.1 Compressed cpio archive file

To inspect the contents of a cpio archive, use the commands listed in listing 17. Note that even if your initrd image file does not end with the `.gz` suffix, it’s a compressed file, and you can add the `.gz` suffix to gunzip it.

**Listing 17:** Extracting the initrd (compressed cpio archive file)

```
# cd ~/bdproject
# mkdir tmp
# cd tmp
# cp /boot/initrd.img-2.6.22.9-customt1 initrd.img.gz
# gunzip initrd.img.gz
# cpio -i --make-directories < initrd.img
```

The result is a small root file system, as shown in listing 18. The small, but necessary, set of applications are present in the various directories.

**Listing 18:** Default Linux initrd directory structure

```
# ls -al
total 13612
drwxr-xr-x 11 ward ward    4096 2007-12-15 21:42 .
drwxr-xr-x  4 ward ward    4096 2007-12-15 21:41 ..
drwxr-xr-x  2 ward ward    4096 2007-12-15 21:42 bin
drwxr-xr-x  3 ward ward    4096 2007-12-15 21:42 conf
drwxr-xr-x  6 ward ward    4096 2007-12-15 21:42 etc
-rwxr-xr-x  1 ward ward    3295 2007-12-15 21:42 init
-rw-r--r--  1 ward ward 13867008 2007-12-15 21:41 initrd.img
drwxr-xr-x  5 ward ward    4096 2007-12-15 21:42 lib
```

```
drwxr-xr-x  2 ward ward    4096 2007-12-15 21:42 modules
drwxr-xr-x  2 ward ward    4096 2007-12-15 21:42 sbin
drwxr-xr-x 12 ward ward    4096 2007-12-15 21:42 scripts
drwxr-xr-x  3 ward ward    4096 2007-12-15 21:42 usr
drwxr-xr-x  3 ward ward    4096 2007-12-15 21:42 var
#
```

Of interest in listing 18 is the `init` file at the root. This file, like the traditional Linux boot process, is invoked when the `initrd` image is decompressed into the RAM disk.

### 5.2.2 Loop device

The loop device is a device driver that allows you to mount a file as a block device and then interpret the file system it represents. Support for the loop device may or may not be present in your kernel, but you can enable it through the kernel's configuration tool (`make menuconfig`) by selecting `Device Drivers > Block Devices > Loopback Device Support`. In the Ubuntu 7.10 kernel, support is by default build-in.

Had the `initrd` image been created using a loop device, you can inspect the loop device as shown in listing 19 (your `initrd` file name will vary).

**Listing 19:** Extracting the `initrd` (loop device)

```
# cd ~/bdproject
# mkdir tmp
# cd tmp
# cp /boot/initrd.img.gz .
# gunzip initrd.img.gz
# mount -t ext -o loop initrd.img /mnt/initrd
# ls -la /mnt/initrd
```

You can now inspect the `/mnt/initrd` subdirectory for the contents of the `initrd`. Again, note that even if your `initrd` image file does not end with the `.gz` suffix, it's a compressed file, and you can add the `.gz` suffix to `gunzip` it.

## 5.3 Manually building a custom initial RAM disk

An `initrd` is automatically created when you created your kernel package. Upon installation of the kernel package, the `initrd` file is installed in the `/boot` folder. We have seen above how you can inspect the contents of the created `initrd` file.

In this report we will create our `initrd` from the ground up however; and it will serve as the permanent root file system. Listing 20 shows how to create an `initrd` image.

**Listing 20:** Utility (`mkird.sh`) to create a custom `initrd`

```
#!/bin/bash

# Housekeeping...
rm -f /tmp/ramdisk.img
rm -f /tmp/ramdisk.img.gz
rm -fr /mnt/initrd

# Ramdisk Constants
RDSIZE=4000
BLKSIZE=1024

# Create an empty ramdisk image
dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE

# Make it an ext2 mountable file system
/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE
```

```

# Mount it so that we can populate
mkdir /mnt/initrd
mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0

# Populate the filesystem (subdirectories)
mkdir /mnt/initrd/bin
mkdir /mnt/initrd/sys
mkdir /mnt/initrd/dev
mkdir /mnt/initrd/proc

# Grab busybox and create the symbolic links
pushd /mnt/initrd/bin
cp /usr/lib/initramfs-tools/bin/busybox .
ln -s busybox ash
ln -s busybox mount
ln -s busybox echo
ln -s busybox ls
ln -s busybox cat
ln -s busybox ps
ln -s busybox dmesg
ln -s busybox sysctl
popd

# Grab the necessary dev files
pushd /mnt/initrd/dev
cp -a /dev/console /mnt/initrd/dev
cp -a /dev/ram0 /mnt/initrd/dev
#ln -s ram0 ramdisk                                #optional
cp -a /dev/null /mnt/initrd/dev
#cp -a /dev/tty1 /mnt/initrd/dev                    #optional
#cp -a /dev/tty2 /mnt/initrd/dev                    #optional
popd

# Equate sbin with bin
pushd /mnt/initrd
ln -s bin sbin
popd

# Create the init file
cat >> /mnt/initrd/linuxrc << EOF
#!/bin/ash
echo
echo "Simple initrd is active"
echo
mount -t proc /proc /proc
mount -t sysfs none /sys
/bin/ash --login
EOF

chmod +x /mnt/initrd/linuxrc

# Finish up...
umount /mnt/initrd
rm -r /mnt/initrd
gzip -9 /tmp/ramdisk.img
cp /tmp/ramdisk.img.gz /home/ward/bdproject/initrd/ramdisk.img.gz

```

To create an initrd, begin by creating an empty file, using /dev/zero (a stream of zeroes) as input writing to the ramdisk.img file. The resulting file is 4MB in size (4000 1K blocks). Then use the mke2fs command to create an ext2 (second extended) file system using the empty file. Now that this file is an ext2 file system, mount the file to /mnt/initrd using the loop device. At the mount point, you now have a directory that represents an ext2 file system that you can populate for your initrd. Much of the rest of the script provides this functionality.

The next step is creating the necessary subdirectories that make up your root file system: /bin, /sys, /dev, and /proc. Only a handful are needed (for example, no libraries are present), but they contain quite a bit of functionality.

To make your root file system useful, use **BusyBox**. This utility is a single image that contains many individual utilities commonly found in Linux systems (such as `ash`, `awk`, `sed`, `insmod`, and so on). The advantage of BusyBox is that it packs many utilities into one while sharing their common elements, resulting in a much smaller image. This is ideal for embedded systems. Copy the BusyBox image from its source directory into your root in the `/bin` directory. A number of symbolic links are then created that all point to the BusyBox utility. BusyBox figures out which utility was invoked and performs that functionality. A small set of links are created in this directory to support your `init` script (with each command link pointing to BusyBox).

The next step is the creation of a small number of special device files. I copy these directly from my current `/dev` subdirectory, using the `-a` option (archive) to preserve their attributes.

The penultimate step is to generate the `linuxrc` file. After the kernel mounts the RAM disk, it searches for an `init` file to execute. If an `init` file is not found, the kernel invokes the `linuxrc` file as its start-up script. You do the basic setup of the environment in this file, such as mounting the `/proc` file system. In addition to `/proc`, I also mount the `/sys` file system and emit a message to the console. Finally, I invoke `ash` (a Bourne Shell clone) so I can interact with the root file system. The `linuxrc` file is then made executable using `chmod`.

Finally, your root file system is complete. It's unmounted and then compressed using `gzip`. The resulting file (`ramdisk.img.gz`) is copied to the your `~/bdproject/initrd` subdirectory so it can be loaded via GRUB.

To build the initial RAM disk, you simply invoke `mkird`, and the image is automatically created and copied to `~/bdproject/initrd`.

## 6 Creating a floppy boot disk

Lack of time for experimenting prevented me from creating a kernel small enough to fit on a floppy. I did manage to put a small kernel image (downloaded from the Internet) on a floppy. However, since I did not extensively experiment with this, I will not describe here how to create a floppy boot disk.

Instead, I will continue with a very similar approach: creating a CD-ROM bootdisk. The advantage of a CD-ROM boot disk is obviously that the kernel size doesn't need to be as small as it has to be for fitting on a floppy. Therefore, the next chapter will explore how this is done.

## 7 Creating a CD-ROM boot disk

If you have followed the steps in chapter 3 and 5, you now have two files ready: your kernel and your initrd image.

This chapter will detail how to create a bootable CD-ROM from these two files. For this, we will need to introduce *GRUB, the Grand Unified Bootloader*. Also, to save on CD-ROMs and ease testing, we will use *Innotek Virtualbox*, vitalisation software similar to VMWare and Microsoft VirtualPC.

### 7.1 GRUB

We have already discussed GRUB in section 2.2.2 and 3.6. We will briefly discuss it here again, since we will need to put grub on our bootdisk.

**Credit:** *The following text is an extract from "The GRUB manual" [4].*

GRUB (GRand Unified Bootloader) is a bootloader. Briefly, a boot loader is the first software program that runs when a computer starts. It is responsible for loading and transferring control to an operating system kernel software (such as Linux or GNU Mach). The kernel, in turn, initialises the rest of the operating system.

When booting with GRUB, you can use either a command-line interface, or a menu interface. Using the command-line interface, you type the drive specification and file name of the kernel manually. In the menu interface, you just select an OS using the arrow keys. The menu is based on a configuration file which you prepare beforehand. While in the menu, you can switch to the command-line mode, and vice-versa. You can even edit menu entries before using them.

The GRUB configuration files can be found, or are to be installed, in the directory `\boot\grub\`

### 7.2 Putting it all together

Now that we have a kernel and an initrd image, we are ready to create our bootable CD-ROM image, which is the final file to be created. Listing 21 shows how this can be done.

**Credit:** *This chapter is based on "Making a GRUB bootable CD-ROM" [10].*

**Listing 21:** Utility (mkbootcd.sh) for creating a boot CD-ROM

```
#!/bin/bash

# Housekeeping...
rm -fr /home/ward/bdproject/bootcd
rm -f /home/ward/bdproject/iso/bootcd.iso

#Set up required folders
cd /home/ward/bdproject
mkdir -p bootcd
mkdir -p bootcd/boot/grub

#Grab bootloader, kernel and initrd
cp /usr/lib/grub/i386-pc/stage2_eltorito bootcd/boot/grub
cp /boot/vmlinuz bootcd/
cp initrd/ramdisk.img.gz /bootcd/

#Create .iso file...
mkisofs -R -b bootcd/boot/grub/stage2_eltorito -no-emul-boot -boot-load-size 4
        -boot-info-table -o bootcd.iso bootcd

#Move to appropriate folder
mv bootcd.iso /iso/bootcd.iso
```



To create a bootable CD-ROM image, we begin by setting up a directory `bootcd` which will contain all the files on the bootcd. This folder should also contain a directory `/boot/grub` which will hold a `stage2_eltorito` boot image.

**El Torito** is a specification [3] for a bootable CD-ROM using BIOS functions, supported by our boot-loader GRUB. For booting from a CD-ROM, GRUB uses a special Stage 2 called 'stage2\_eltorito'. The only GRUB files you need to have in your bootable CD-ROM are this 'stage2\_eltorito' and optionally a config file 'menu.lst'. You don't need to use 'stage1' or 'stage2', because El Torito is quite different from the standard boot process.

Once our folder structure has been set up, the `eltorito` boot image, the kernel and the `initrd` image are copied to our `bootcd` folder. If required, you could copy additional files into the `bootcd` folder, which will then be available if you mount the CD-ROM after booting.

The next step is creating an ISO9660 file system based on our `bootcd` folder, which can easily be done with the command `mkisofs` (make iso file system).

The `-R` option is essentially for RockRidge extensions which allow us to have softlinks on the CD-ROM and mixed case filenames. The `-b` options specifies the path and filename of the boot image to be used when making an "El Torito" bootable CD-ROM. The `-no-emul-boot` option prevents the image from being treated as a floppy image. The `-boot-load-size 4` bit is required for compatibility with the BIOS on many older machines. When the `-boot-info-table` option is given, `mkisofs` will modify the boot file specified by the `-b` option by inserting a 56-byte "boot information table" at offset 8 in the file. The `-o bootcd.iso` option specifies the output filename. And finally the `bootcd` at the end specifies the folder to create your file system from. See the man pages of `mkisofs` for more information (type `man mkisofs` at the terminal prompt).

This command produces a file named `bootcd.iso`, which then can be burned into a CD-ROM (or a DVD), or loaded using a virtual machine (see below). `mkisofs` has already set up the disc to boot from the `boot/grub/stage2_eltorito` file, so there is no need to setup GRUB on the disc.

You can use the device '(cd)' to access a CD-ROM in your config file `menu.lst` (see below). This is not required; GRUB automatically sets the root device to '(cd)' when booted from a CD-ROM. It is only necessary to refer to '(cd)' if you want to access other drives as well.

### 7.3 Using Innotek Virtualbox

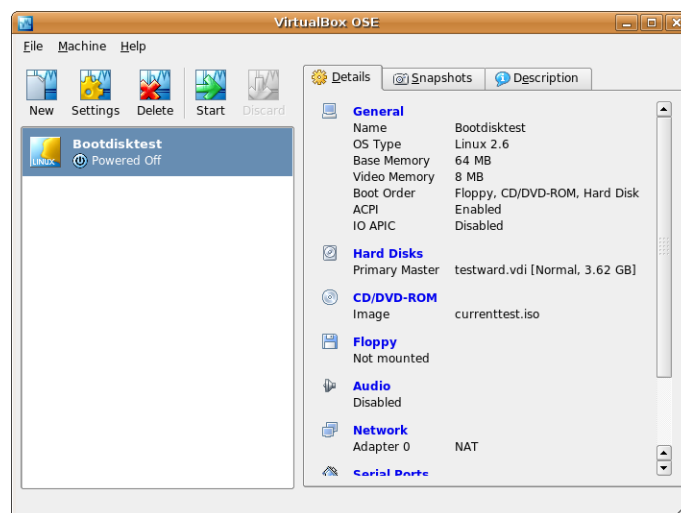


Figure 12: Innotek Virtualbox

Testing your boot disk (CD-ROM or floppy) becomes a lot more practical if you use virtualisation

software. Since VMware player is not (yet) compatible with Ubuntu 7.10, you can install **Innotek Virtualbox**, an excellent open source alternative. See figure 12.

Installing Virtualbox can be easily done through the application manager of Ubuntu 7.10 (Applications > Add/Remove).

Once installed, create a new virtual machine. Select “Kernel 2.6” for the OS type. For the amount of RAM, 64 Mb should suffice. A harddisk is not required.

Before starting up the virtual machine, you have to “insert” your CD-ROM (or floppy) by mounting it. This can be done by selecting the virtual machine, clicking “Settings”, “CD/DVD-ROM”. Click “Mount CD/DVD drive” and choose “ISO image file”. Then select the .iso file that you want to mount.

## 7.4 Testing your bootable image

With our bootcd.iso image ready, we are now finally able to test it.

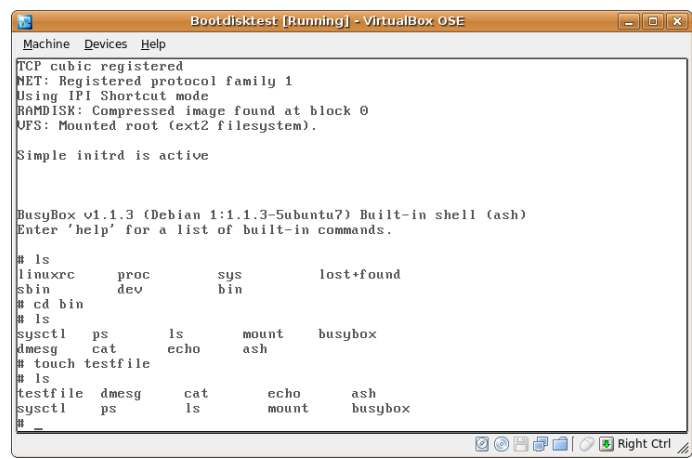
**Credit:** *This chapter is based upon “Linux initial RAM disk (initrd) overview” [8].*

When you boot from the image, the GRUB command line will appear. You can now interact with GRUB to define the specific kernel and initrd image to load. The `kernel` command allows you to define the kernel file, and the `initrd` command allows you to specify the particular initrd image. When these are defined, use `boot` to boot the kernel, as shown in listing 22.

**Listing 22:** GRUB command line

```
grub> root (cd)
grub> kernel /vmlinuz
grub> initrd /ramdisk.img.gz
grub> boot
```

**NOTE:** *Should you not specify the initrd image, then your kernel will boot but terminate with a kernel panic because it can not find the initrd image.*



**Figure 13:** Booting your Linux kernel with your simple initrd

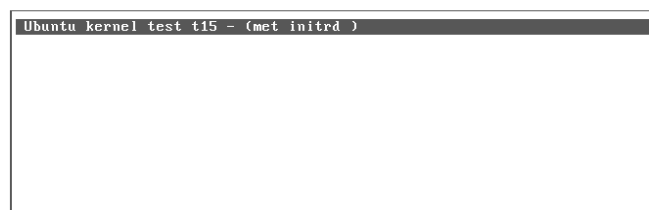
After the kernel starts, it checks to see if an initrd image is available (more on this in chapter 8), and then loads and mounts it as the root file system. You can see the end of this particular Linux start-up in figure 13. When started, the ash shell is available to enter commands. In this example, I explore the root file system and demonstrate that you can write to the file system by touching a file (thus creating it). Note here that the first process created is `linuxrc` (commonly `init`).

## 7.5 Adding a boot menu to GRUB

To avoid typing the same commands at the GRUB prompt each time you boot your customised system, you can put these in a config file named `menu.lst`. This file provides you with a simple menu after booting (see listing 23), and should be placed in the `boot/grub` folder in your bootdisk. The easiest way to create this `menu.lst` file is by copying the existing file from the `/boot/grub` folder on your root system. You can then modify it so that it contains the following:

**Listing 23:** Example configuration of `menu.lst`

```
title      Standalone test kernel
root      (cd)
kernel    /vmlinuz-t15
initrd    /ramdisk.img.gz
```



Use the ↑ and ↓ keys to select which entry is highlighted.  
Press enter to boot the selected OS, 'e' to edit the  
commands before booting, or 'c' for a command-line.

**Figure 14:** The grub boot menu (with only one entry)

Once you have a customised `menu.lst` file, put it in the `boot/grub/` folder on your bootdisk *before* running the `mkisofs` command (you might need to modify listing 21 if you want to use the `mkbootcd.sh` script).

## 8 Booting with an initial RAM disk

**Credit:** *The text below is an extract from “Linux initial RAM disk (initrd) overview” [8].*

Now that you’ve seen how to build and use a custom initial RAM disk, this chapter explores how the kernel identifies and mounts the initrd as its root file system. I walk through some of the major functions in the boot chain and explain what’s happening.

The boot loader, such as GRUB, identifies the kernel that is to be loaded and copies this kernel image and any associated initrd into memory. You can find much of this functionality in the `./init` subdirectory under your Linux kernel source directory.

After the kernel and initrd images are decompressed and copied into memory, the kernel is invoked. Various initialisation is performed and, eventually, you find yourself in `init/main.c:init()` (subdir/file:function). This function performs a large amount of subsystem initialisation. A call is made here to `init/do_mounts.c:prepare_namespace()`, which is used to prepare the namespace (mount the dev file system, RAID, or md, devices, and, finally, the initrd). Loading the initrd is done through a call to `init/do_mounts_initrd.c:initrd_load()`.

The `initrd_load()` function calls `init/do_mounts_rd.c:rd_load_image()`, which determines the RAM disk image to load through a call to `init/do_mounts_rd.c:identify_ramdisk_image()`. This function checks the magic number of the image to determine if it’s a minux, etc2, romfs, cramfs, or gzip format. Upon return to `initrd_load_image`, a call is made to `init/do_mounts_rd:crd_load()`. This function allocates space for the RAM disk, calculates the cyclic redundancy check (CRC), and then uncompresses and loads the RAM disk image into memory. At this point, you have the initrd image in a block device suitable for mounting.

Mounting the block device now as root begins with a call to `init/do_mounts.c:mount_root()`. The root device is created, and then a call is made to `init/do_mounts.c:mount_block_root()`. From here, `init/do_mounts.c:do_mount_root()` is called, which calls `fs/namespace.c:sys_mount()` to actually mount the root file system and then `chdir` to it. This is where you see the familiar message shown in figure 13: VFS: Mounted root (ext2 file system).

Finally, you return to the `init` function and call `init/main.c:run_init_process`. This results in a call to `execve` to start the `init` process (in this case `/linuxrc`). The `linuxrc` can be an executable or a script (as long as a script interpreter is available for it).

The hierarchy of functions called is shown in listing 24. Not all functions that are involved in copying and mounting the initial RAM disk are shown here, but this gives you a rough overview of the overall flow.

**Listing 24:** Hierarchy of major functions in initrd loading and mounting

```
init/main.c:init
  init/do_mounts.c:prepare_namespace
    init/do_mounts_initrd.c:initrd_load
      init/do_mounts_rd.c:rd_load_image
        init/do_mounts_rd.c:identify_ramdisk_image
          init/do_mounts_rd.c:crd_load
            lib/inflate.c:gunzip
          init/do_mounts.c:mount_root
            init/do_mounts.c:mount_block_root
              init/do_mounts.c:do_mount_root
                fs/namespace.c:sys_mount
            init/main.c:run_init_process
          execve
```

## 9 Conclusion and further work

The aim of this report was to provide both a hands-on tutorial to creating a small, customised operating system, as well as a brief insight on the concept and technologies encountered to do so.

As I speak for myself, my expectations as stated in chapter 1 were certainly met. I have indeed learned a lot about the boot process, file systems, shell scripting, the Linux directory tree and a general better understanding of an operating system. The project was more challenging than expected: nonetheless the huge amount of documentation available in the Internet, the quality varies greatly, thereby actually decreasing the challenge to get *accurate* information. Many times, I was stuck in a dead end and had to retreat to try other approaches.

I did not have enough time, however, to create a floppy boot disk. I opted for a CD-ROM boot disk. Reducing the kernel size turned out to be a time-consuming task, involving a lot of experimentation. Creating a CD-ROM bootdisk, does not put any practical constraints on the kernel size.

If there would have been more time for this project, I would have liked to explore a couple of things in more detail, such as network booting, including a system call into the kernel and getting the kernel size further down so that it would have fit on a floppy disk.

## References

- [1] Bash Shell Programming in Linux, *www.arachnoid.com (on-line)*. 2006-03, printed 2007-12-29, from [http://www.arachnoid.com/linux/shell\\_programming.html](http://www.arachnoid.com/linux/shell_programming.html).
- [2] Chapter 14 - Removing and installing software, *Debian tutorial (on-line)*. 2006-06-17, printed 2007-12-29, from <http://www.debian.org/doc/manuals/debian-tutorial/ch-dpkg.html>.
- [3] El-torito, Bootable CD-ROM Format Specification, version 1.0, 1995-01-25.
- [4] GRUB manual, 2005-05-26, from <http://orgs.man.ac.uk/documentation/grub/grub.html>.
- [5] How to Customise Your Ubuntu Kernel, *the How-To Geek (on-line)*. Printed 2007-12-29, from <http://www.howtogeek.com/howto/ubuntu/how-to-customize-your-ubuntu-kernel/>.
- [6] Inside the Linux boot process, *IBM developerWorks library (on-line)*. 2006-05-31, printed 2007-12-29, from <http://www.ibm.com/developerworks/linux/library/l-initrd.html>.
- [7] The Linux Bootdisk HOW-TO, *The Linux Documentation Project (on-line)*. 2002-01, version 4.5, from <http://tldp.org/HOWTO/Bootdisk-HOWTO/>.
- [8] Linux initial RAM disk (initrd) overview, *IBM developerWorks library (on-line)*. 2006-07-31, printed 2007-12-29, from <http://www.ibm.com/developerworks/linux/library/l-linuxboot/>.
- [9] The Linux System Administrator's Guide, Chapter 3 Overview of the Directory Tree *The Linux Documentation Project (on-line)*. version 0.9, from <http://www.tldp.org/LDP/sag/html/index.html>.
- [10] Making a GRUB bootable CD-ROM, *GRUB manual (on-line)*. printed 2007-12-29, from [http://orgs.man.ac.uk/documentation/grub/grub\\_3.html#SEC11](http://orgs.man.ac.uk/documentation/grub/grub_3.html#SEC11).
- [11] Silberschatz, Galvin and Gagne. *Operating System Concepts*, Chapter 21.1 Linux History, pages 737-738. John Wiley & Sons, seventh edition, 2005

[12] Using Debian Linux Packages, *About debian Linux (on-line)*. printed 2007-12-30, from [!http://www.aboutdebian.com/packages.htm](http://www.aboutdebian.com/packages.htm).