Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

A Linux-based, Web-oriented operating system designed to boot quickly

by

Ulf Magnusson

LIU-IDA/LITH-EX-A--11/050--SE

2011-12-22

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

A Linux-based, Web-oriented operating system designed to boot quickly

by

Ulf Magnusson

LIU-IDA/LITH-EX-A--11/050--SE

2011-12-22

Supervisors: Prof. Simin Nadjm-Tehrani, Nicklas Larsson (Opera Software ASA)
Examiner: Prof. Simin Nadjm-Tehrani

Abstract

This thesis describes the design and implementation of a Linux-based, Web-oriented operating system called Awesom-O, designed with a focus on short boot time and small disk footprint. Among other techniques for lowering boot time, a semi-automatic method for generating a Linux kernel of minimal size for a given platform is developed, making use of an interpreter for the Linux kernel's configuration language, Kconfig. The boot process of the finished system is analyzed to identify limiting factors in lowering its boot time further, and techniques for overcoming these are suggested.

Excluding the initial BIOS stage of the boot process, the boot time of the finished system—up until it is idling inside the web browser interface waiting for user input—is 3.8 seconds (2.1 seconds to a shell prompt, 1.7 seconds in the kernel) on an Acer Travelmate 8200 laptop with an Intel Core Duo CPU at 2.0 GHz and a Momentus 5400.2 SATA (ST9120821AS) hard drive; 2.4 seconds (1.6 seconds to a shell prompt, 1.1 seconds in the kernel) on a Celsius M460 workstation with an Intel Core 2 Quad CPU at 2.5 GHz and a Barracuda 7200.11 SATA hard drive (ST3500320AS); 4.6 and 4.0 seconds respectively for the same systems when booting from a USB 2.0 device (a ChipsBank CBM2080 USB 2.0 stick); and 12.6 seconds on the BeagleBoard (8 seconds in the bootloader—an obvious area for future improvement).

The Web functionality in Awesom-O is implemented atop the Opera Linux Devices SDK: a software framework for integrating web browser functionality in small Linux-based systems.

Acknowledgments

I would like to thank my thesis supervisor, professor Simin Nadjm-Tehrani at the Department of Computer and Information Science at Linköping University, and Nicklas Larsson, who supervised the thesis at Opera Software.

Contents

Chapter 1

Introduction

1.1 Background

In recent years a new class of operating system has emerged, popularly termed the **Web Operating System** (**Web OS**). Characteristic of Web OSs is a focus on web browsing and **web applications**: applications built around technologies such as HTML and JavaScript that run inside a web browser, often making heavy use of external network resources. Web OSs commonly target relatively inexpensive hardware platforms, where their small disk footprint and relative simplicity enable acceptable performance. One such platform is the **netbook**: an umbrella term for a small, inexpensive laptop computer geared towards web browsing.

1.2 Purpose

The purpose of this thesis was to construct an experimental Web OS, codenamed **Awesom-O**, running atop the Linux kernel and the **Opera Linux Devices SDK**[3] (shortened to **Devices SDK** from here on), with a small disk footprint and a short boot time. This Web OS was to be used as a platform for exploring boot time and disk usage optimization techniques, and for identifying limiting factors with regard to these.

1.3 Goals

The goal of the thesis was to explore how short boot time can be attained for a Linux-based Web OS without modifying the Linux kernel source code. To this end, the system was to use a more-or-less minimal set of components, set up in a very basic way. In order to find profitable avenues for further optimization and identify limiting factors in lowering boot time, the boot process of the system was to be profiled to identify where time was being spent.

To have some concrete initial goals for boot time and disk usage, it was decided that the system should aim for

- A boot time of less than 10 seconds, excluding BIOS, on typical 2011 netbook hardware.
- A total size of less than 50 MiB, including the Linux kernel.

1.4 Methodology

To avoid unnecessary overhead and get a system containing more or less just the bare minimum of components required, the system was constructed "from scratch" rather than starting with an existing Linux distribution. More specifically, starting with just a Linux kernel image, software components found to be necessary were added one by one up until the point where the user interface was functional and the Devices SDK was able to fetch a web page from a remote server on the Internet and render it. Additionally, the Linux kernel itself was reconfigured to reduce boot time using a novel technique described in chapter 4 — *Boot time optimization through semi-automatic kernel minimization*.

1.5 Audience

Most of the material in this thesis should be useful to anyone wishing to gain a basic understanding of how Linux systems boot and run and of how their boot time can be improved, and not just to Web OS developers. The report also contains a lot of semi-obscure information about the Linux boot process, often presented in a friendlier way than in the sometimes very terse original documentation.

1.6 Scope

The intent of the thesis was never to create a very *usable* Web OS, but only to investigate ways to construct the foundation upon which a more complete system could be built. More specifically, little effort was put into creating a good user interface once the Devices SDK was able to fetch a page over the network and render it.

Three different platform configurations are investigated:

- IBM PC compatible (x86) systems booting from disk.
- IBM PC compatible (x86) systems booting from a USB mass storage device (a ChipsBank CBM2080 USB 2.0 stick).
- The Texas Instruments BeagleBoard[12], a small ARM-based single-board computer geared towards the open source development community.

1.7 Limitations

The following limitations apply to the scope of the thesis.

1.7.1 Modifying the Linux kernel

One approach to reducing boot time is to investigate where the Linux kernel spends time during boot and modifying its source code to run these stages in less time. This is only done in a simple sense in this thesis by disabling unneeded features (see section 4 — *Boot time optimization through semi-automatic kernel minimization*); no kernel code is modified.

1.7.2 BeagleBoard bootloader optimization

As we will see later, the bootloader accounts for the bulk of the boot time of the finished system on the BeagleBoard. A natural next step on that platform would have been to look into ways to load and initialize the Linux kernel faster. Unfortunately the automatic kernel minimization technique discussed later in this report took much longer to implement than I had anticipated, and so I ran out of time before I could look into BeagleBoard bootloader optimization.

1.7.3 Thumb-2 instructions

Disk footprint and most probably boot time could be improved on the BeagleBoard by compiling the kernel and applications to use the Thumb-2 instruction set, which yields smaller machine code compared to the standard ARM instruction set. Unfortunately Thumb-2 support was rather poor and buggy in the tools used in this thesis (particularly Buildroot) at the time of writing (*Dec. 2011*), so the standard ARM instructions set on the Cortex-A8 was used instead.

1.7.4 Filesystems

Awesom-O uses the second extended filesystem (ext2) throughout. For a production system this might be especially problematic for the BeagleBoard, where the root filesystem is stored in NAND Flash, as ext2 does not support wear leveling. Inodes, which store metadata such as disk location and size for files, might cause problems in particular as they never move on disk and are frequently updated. The emulation layer that makes the NAND Flash look like block device could also incur some overhead.

It might also be possible to speed up the boot by using read-only, compressed filesystems to store data that never (or seldom) changes, or by arranging data on the disk in a way that minimizes seeks (movement of drive heads); see section 6.1 — *Future investigations and improvements*.

1.7.5 Security and related concerns

Though important in systems that connect to the Internet, security concerns are outside the scope of this thesis. All processes in the early version of Awesom-O discussed run with superuser (root) privileges.

1.8 Nomenclature and abbreviations

- **PC** will stand for any IBM-compatible x86-based based system. Most of what is said will probably apply equally well to x86_64 systems, as well as to x86/x86_64 systems that aren't strictly IBM PC compatible, such as some EFI-based[6] systems.
- The Opera Linux Devices SDK will be referred to simply as the **Devices SDK**.
- When otherwise not qualified, **kernel** will refer to a 2.6.x Linux kernel with $x >= 27$.

- The notation `foo/{bar,baz,qux,...}`—a notation called **brace expansion**, supported in many shells—refers to the files `foo/bar`, `foo/baz`, `foo/qux`, ...
- The **wildcard** notation `foo*bar` refers to all files whose names can be obtained by substituting some (possibly empty) string for `*`. It will also be used in a more general sense throughout the report to refer to parts of a command or string that might vary.

1.9 Typographical conventions

The following typographical conventions will be used throughout the report:

- Important terms and names of products and software packages will be written in a **bold font** when they are first introduced.
- Shell and bootloader commands, names of shell utilities and functions, and filesystem paths will be written in a `fixed-width font` when they appear within paragraphs.
- Kernel parameters and configuration options will be written in a *thin italic font*.
- Keys on the keyboard and key combinations will be written in **BOLD CAPITAL LETTERS (CTRL-C)**.
- Code listings will use a

```
small fixed-width font on a light-gray background.
```

Chapter 2

Prerequisites

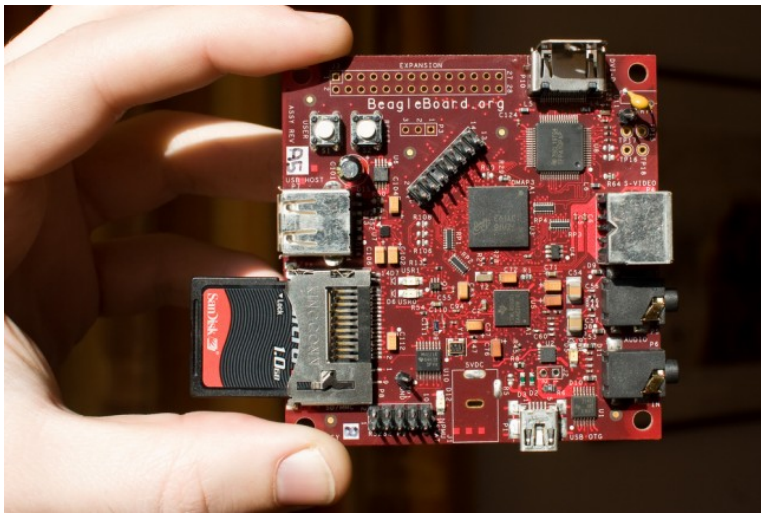
This chapter provides an overview of the BeagleBoard, the Linux kernel and the Linux boot process, all of which we will need basic familiarity with before discussing the implementation of Awesom-O.

2.1 The BeagleBoard

The BeagleBoard[12] is an ARM-based single-board computer developed by Texas Instruments that targets the open source development community. The C3 revision of the BeagleBoard was used for this thesis. System specifications include the following:

- An OMAP3530 processor with an ARM Cortex-A8 core running at 720 MHz.
- 256 MiB of DRAM.
- 256 MiB of on-board NAND Flash memory.
- DVI, USB and and SD/MMC peripheral connections.

The BeagleBoard can boot from either the on-board flash or an SD/MMC card. Detailed hardware specifications can be found at the official BeagleBoard website[12]. The board is pictured below:



2.2 The Linux kernel

The Linux kernel provides an environment for the concurrent execution of **processes**: tasks that perform the useful work within the system. At a high level, a process encapsulates the state of a running **program**, where the program is the set of instructions and data required to carry out the task. The Linux kernel manages resources such as CPU, RAM, disk storage, and peripheral hardware devices, and coordinates how processes make use of these resources.

To insulate programs from having to know the exact make-up of the system they are running on, and to prevent processes from being able to arbitrarily modify the state of the running machine, the kernel provides **hardware abstraction**. When a process wishes to use the services provided by a particular hardware device, it generally does so through an interface provided by the kernel. Often, one such interface will cover a class of similar devices, and the kernel will take care of translating the process's requests into a format the raw hardware on the device can understand. For the kernel to know how to talk to specific devices, corresponding **device drivers** must be provided. Device drivers translate the abstract request sent by processes into the device's native language, and also take care of initializing and maintaining the operation of the device.

In the Linux kernel, a device driver can either be **compiled-in** or loaded as a **module**. A compiled-in driver appears as part of the kernel binary (the compiled image of the kernel) and will typically probe for its device during boot. A module is a device driver (or other kernel component) that is loaded dynamically on an already running system.

The Linux kernel comes with source code for thousands of device drivers. The set of drivers to compile-in and to compile as modules can be configured before the kernel is compiled (using for example the `make menuconfig` interface). When striving to minimize the boot time of a system it makes sense to use as few drivers as possible, as any superfluous drivers (for example corresponding hardware devices that are not installed) will usually need time to probe for their corresponding devices[1].

2.3 The Linux boot process

We now give an overview of the Linux boot process. For early boot—up to the point where kernel code begins executing—the process differs significantly between PC and BeagleBoard, and separate discussions will be given. Once the kernel begins executing, the same high-level description will apply to both platforms.

2.3.1 Early boot on PC

BIOS

The first code that runs when a PC is powered on resides in the **BIOS** (**Basic Input/Output System**)[2]: boot firmware usually stored in (reprogrammable) ROM on a chip on the system's motherboard. The role of the BIOS is to perform basic hardware initialization, functional

[1] Another concern is that this probing could interfere with other devices.

[2] Some modern x86-based systems have replaced the BIOS with **EFI**: the Extensible Firmware Interface. EFI fills a role similar to that of the BIOS, but is designed to be more flexible and does away with some of the legacy limitations of BIOS.

testing, and identification of certain hardware peripherals (e.g. disk drives), and then initiate the next stage of the boot process by running code from a (usually user-specifiable) **boot device**[3].

Bootloader (Grub Legacy)

In the common case of booting from a disk or disk-like device, the instructions loaded by the BIOS from the boot device will reside in the first sector (512 bytes[4]) of the disk, called a **boot sector**. Usually, the disk will use disk partitioning, and the boot sector will be the **MBR**—**Master Boot Record**—which in addition to machine code also contains a table of disk partitions[5]. The code in the MBR comes from the **bootloader**, which performs the system initialization necessary before the operating system can run and then loads and executes the operating system kernel.

Awesom-O uses the **Grub 0.97** bootloader[9] (commonly called **Grub Legacy**), which up until the release of the Ubuntu 9.10 Linux distribution in 2009 was the overwhelmingly most popular Linux bootloader on PCs. Grub 0.97 splits the boot process up into several stages, where each stage loads the next successive stage. The successor to Grub Legacy, Grub2, also splits the boot process up into stages, but uses a slightly different scheme and nomenclature (in particular it supports loading "drivers" for different functionality from module files).

Grub Legacy was chosen over the more advanced Grub2 for simplicity, as Grub2 is notoriously more complex than its predecessor. Discussion with Grub developers in the *#grub* channel on the Freenode IRC network suggested that it was unlikely that performance would differ between the two versions for the simple use cases in Awesom-O.

The purpose of the Grub MBR code (*stage 1* in Grub terminology) is to load the main Grub binary (*stage 2* in Grub terminology). The *stage 2* binary optionally presents a menu of boot options (for example to let the user choose which OS to boot if several are installed), performs additional hardware initialization needed before the operating system kernel can run, and finally loads the kernel into memory and executes it (by jumping to the kernel image's *entry point*). The Grub *stage 2* binary is usually stored as a file in a filesystem (often in a small filesystem containing files needed during boot and not much more). Due to size limitations (512 bytes, some of which is used by the partition table), *stage 1* does not understand filesystem structure, and so loads *stage 2* directly from a fixed block number. As a more robust alternative that allows the *stage 2* binary to move around on the disk without breaking the boot process, Grub can optionally be set up so that *stage 1* loads a filesystem-specific *stage 1.5* binary, usually located directly after the MBR, whose sole purpose is to navigate the filesystem to locate *stage 2*.

[3]In addition to boot code the BIOS contains a simple system interface invoked via software interrupts, containing routines for interacting with certain hardware devices and performing primitive operations like printing characters on the screen in text mode; this is where BIOS gets its name. Modern operating systems access most hardware devices directly through device drivers and only use the BIOS for certain specialized operations. The BIOS interface is often used early in the boot process however, particularly by bootloaders.

[4]A shift is currently taking place towards 4096-byte sectors, which allow big files to be stored more efficiently by reducing the amount of metadata on disk.

[5]Disk partitions are a division of the disk into separate data areas, each typically containing its own filesystem or other data.

2.3.2 Early boot on BeagleBoard

Boot ROM and X-Loader

On the BeagleBoard, the first code executed upon turning on the system comes from the on-board boot ROM, which performs basic hardware initialization and then loads the **X-Loader** binary (from where depends on the boot order, which can be modified by holding down the USER button on the BeagleBoard while the system boots). The purpose of X-Loader is to set up an environment suitable for, and then loading and executing, the **Das U-Boot** bootloader. Das U-Boot in turn loads and executes the Linux kernel.

Bootloader (Das U-Boot)

Das U-Boot is a bootloader popular on ARM-based system, including the BeagleBoard. It has support for reading and writing SD/MMC cards and NAND flash, and is configured using an interface similar to that of the Bourne family of shells. Unlike Grub, Das U-Boot usually requires low-level operations such as loading the kernel into memory to be specified explicitly. In Awesom-O, we store the kernel and root filesystem on the BeagleBoard's on-board NAND Flash memory and use U-Boot commands to load the kernel into memory and execute it.

Once the bootloader has finished its job, the boot process continues within the kernel's initialization code. The remaining sections apply to both the PC and BeagleBoard boot process.

2.3.3 Hardware initialization (PC and BeagleBoard)

The Linux kernel begins its life by performing basic hardware initialization and setup[6]. This includes things like initializing video adapters, disk controllers and I/O devices; setting up interrupt tables; decompressing the kernel if it was stored in a compressed format on disk (the common case); switching from real to protected mode on x86; programming the MMU; setting the system time and date; and initializing various kernel data structures and subsystems. Near the end of the initialization process, any device drivers compiled into the kernel are made to probe for and initialize their corresponding hardware devices. Usually, each driver that finds a corresponding device will print out a short diagnostic message, which appears among the boot messages of the system on the system console[7].

Once all drivers compiled into the kernel have probed for and initialized their corresponding hardware devices, the boot proceeds with the kernel mounting the **root filesystem**. Before describing how the root filesystem is mounted, two concepts need to be introduced: **kernel parameters** and **device nodes**.

2.3.4 Kernel parameters

Kernel parameters are parameters to the Linux kernel for controlling aspects of how it boots and runs, passed to it by the bootloader in the form of the **kernel command line**. The kernel command line is a whitespace-separated list of parameters, where each parameter takes

[6]Even though the BIOS has already performed some hardware initialization, the Linux kernel reinitializes most hardware for portability purposes.

[7]These are written to the so-called **kernel ring buffer**, the contents of which can be inspected after boot with the `dmesg` command.

the form of either a single word (for example *quiet*, to instruct the kernel not to display boot messages on the system console), or two words separated by an equals sign with no intervening whitespace (for example *root=/dev/sda1* to specify the device holding the root filesystem, described later). 'Word' here is taken to mean any sequence of characters not containing any whitespace. An example of a complete kernel command line might be

```
root=/dev/sda1 init=/bin/start video=vesa:ywrap,mtrr:3 vga=0x360 rw quiet
```

Documentation for most kernel parameters that are recognized by the Linux kernel can be found in the Linux source tree in *Documentation/kernel-parameters.txt*. Any parameters in the kernel command line not recognized by the kernel are passed as arguments to the **init** process (see section 2.3.8 — *The `init` process*). This is how for example the *single* kernel parameter works, which instructs `init` to boot into single-user mode.

2.3.5 Device nodes

Device nodes (sometimes called **device files** or **special files**) are special files, commonly found in the `/dev` directory on Unix-like systems. Each device node corresponds to some device, where the "device" might be a hardware device or a pseudo-device implemented entirely in software. Each device node typically corresponds to a device driver in the kernel, though many device nodes might share the same device driver.

By reading from and writing to device nodes, and by using the `ioctl()` system call, processes can transfer data to and from the corresponding device and set parameters related to it. The rationale for device nodes is to allow for existing APIs and code for dealing with files to be used for device communication. Having the device interface be file-oriented also makes it easy to manipulate devices from a shell.

Device nodes are created using the `mknod` utility. The mapping from device nodes to devices (device drivers) is done via a pair of integers assigned to each device node: its **major number** and its **minor number**. The official list of major/minor number assignments can be found in the Linux source tree, in the file *Documentation/devices.txt*.

Device nodes come in two different varieties: **character devices** and **block devices**[8]. Character devices correspond to devices on which reads and writes are done in a "streaming" fashion, sending/receiving one character at a time; examples include keyboards, mice, terminals, and serial ports. Block devices correspond to devices that support random access and transfer data in blocks (generally of a fixed size); examples include storage devices such as hard disks and CD-ROMs.

Device nodes are completely characterized by their type and major and minor numbers, so creating device nodes on one system for use on another is perfectly safe (as long as the other system supports the device).

We are now ready to cover the next step of the boot process: mounting the root filesystem.

2.3.6 Mounting the root filesystem

Unix-like operating systems arrange files and directories in a tree structure with a single root denoted '`/`'. At any location in this tree corresponding to a directory the contents of a filesystem

[8]There are also FIFOs, commonly used for interprocess communication, as well as a newly introduced device type called a Memory Technology Device, tailored to the needs of Flash-based storage devices.

can be inserted, which in Unix parlance is called "mounting" the filesystem at that location. After hardware initialization, the next step in the Linux boot process is to locate a filesystem to serve as the root of the tree; that is, to mount a filesystem on '`/`'.

Linux provides multiple options for mounting the root. The traditional—and least flexible—option is the *root* kernel parameter, which takes as its argument the device on which the root filesystem resides. For example, to specify that the first partition of the first SATA hard drive plugged into the computer contains the root filesystem, the kernel command line might contain *root=/dev/sda1*[9], where "sd" is an abbreviation of "SCSI device" (Linux uses the SCSI subsystem for many things besides true SCSI disks), "a" is for the first drive, and "1" is for the first partition.

One problem with using the *root* kernel parameter is that we might not have enough information at boot time to determine which device holds our desired root filesystem. For example, we might want to fetch the filesystem image to use as the root from a remote server across the network, or we may need to search through devices connected to the computer in some intelligent fashion to find the one we need. Worse yet, the device might not even be initialized by the time the kernel attempts to mount it as root, or we might need to load device drivers or perform other initialization before the device can be used. The problem with devices getting initialized too late often occurs when booting from USB and MMC (SD/MMC cards) devices, as the corresponding kernel subsystems run asynchronously and usually finish device initialization at a later time than when the kernel first attempts to mount root. As putting dedicated code into the kernel to handle each and every way a user might conceivably want to locate and initialize the root device would not be feasible, the Linux kernel includes a different mechanism: the **initial RAM filesystem**.

2.3.7 The initial RAM filesystem (initramfs/initrd)

The initial RAM filesystem (abbr. initial RAM fs) is a root filesystem that resides in system memory, usually put there by the bootloader before the kernel begins executing[10]. The purpose of the initial RAM fs is to act as a temporary root filesystem containing the smarts necessary to locate, initialize and mount the "real" root. For some small embedded systems, the initial RAM fs might even be a candidate for use as the permanent root.

Linux provides two different initial RAM fs mechanisms. The older mechanism, **initrd**, expects an executable file `/linuxrc` (often a shell script) to exist in the initial RAM fs, which when executed should replace the root filesystem with the real root[11]. The newer mechanism, **initramfs**, largely meant to replace initrd, instead executes `/init` (or whatever was specified via the *rdinit* kernel parameter), which is never expected to exit. With initramfs, it is the responsibility of the `/init` process to replace itself with the real init process via the `exec()` system call once it has mounted (and usually `chroot()`'ed into) the real root[12].

[9]The notation *root=/dev/x* can be confusing given that `/dev/x` does not actually exist before the root has been mounted, and need not even exist at all in the root filesystem. This notation must be seen as purely symbolic, where `/dev/x` means "the device that conventionally gets assigned the device node `/dev/x`." Alternative forms do exist, including one where the device is specified using its major and minor device number.

[10]In Grub, the `initrd` (*initial RAM disk*) command is used to load an initial RAM fs image into memory in the format the kernel expects.

[11]This can be accomplished by using the `pivot_root(2)` system call.

[12]initramfs makes more efficient use of CPU and memory resources than initrd by not being a "true" filesystem with disk caching, which would be superfluous as the filesystem is already in memory.

2.3.8 The `init` process

The first process that runs on a Linux system after the root filesystem has been mounted is the **init** process. The executable for the `init` process has traditionally been `/sbin/init`, but can be set to anything via the *init* kernel parameter. `init` serves as the parent of all processes; all other processes are either directly or indirectly started by `init`. At boot, `init` will typically start processes that perform final system initialization, optionally prompt the user for a user ID and password, and then launch the process(es) that provide the user interface. This initialization might include, among other things, loading additional device drivers as modules, launching background processes (**daemons**), and starting a graphical user interface provider such as an X11 server. Despite its special role, the `init` process is just an ordinary (usually ELF, though it could also be for example a shell script) executable file[13].

The two most common flavors of `init` design are **BSD style** and **SysV style**, traditionally used on BSD- and SysV-derived (such as Linux) systems, respectively. Lately, many distributions are switching to newer `init` processes such as **Upstart**[13], originating with the Ubuntu distribution. Upstart allows for greater flexibility in how subtasks are run compared to the traditional init processes, in particular simplifying parallel execution of tasks during boot. We will use our own init process, `/bin/start`, that might serve as a model for the simplest init process possible.

2.4 Configuring and building a Linux kernel

Source code for recent mainline Linux kernel versions can be downloaded from `http://www.kernel.org` (*as of Dec. 15, 2011*). After fetching and unpacking the source, the next step is to configure the kernel to set parameters and select features and drivers to include. This is done by running a command of the form `make *config` in the top-level source directory, where the only two variants we will need to be aware of are `make menuconfig` and `make defconfig`. `make menuconfig` presents a menu-driven interface for configuring the kernel, while `make defconfig` installs a default configuration chosen by the maintainer of the architecture being built for; our purpose for using it is to avoid picking up a default kernel configuration from the development system on which we compile the kernel, which is likely to enable many things we do not need. `make menuconfig` can be run after `make defconfig` to modify the default configuration.

After the configuration step, the kernel is built by running `make` in the top-level source directory. The primary output of the build process is a binary kernel image plus a set of compiled modules.

2.5 Cross-compilation

We are almost ready to cover the design and implementation of Awesom-O, but will first need to know about some terminology and tools related to cross-compilation.

[13] A common strategy for troubleshooting systems that fail to boot (but are able to mount root) is to put *init=/bin/sh* on the kernel command line, which will launch an interactive root shell upon boot.

2.5.1 Cross-compilation terminology

A **(compiler) toolchain** is a set of development tools—compiler, linker, debugger, library creation tools, standard language libraries, etc.—used to create programs that run on some particular platform, called the toolchain's **target (platform)**. When we write for example "ARM toolchain", we mean a toolchain that targets an ARM-based platform. The **host (platform)** of the toolchain is the platform on which the toolchain itself runs.

A **cross-toolchain** is a toolchain whose host and target platforms differ, so that it produces programs that run on a different platform than the toolchain itself does. A compiler that forms part of a cross-toolchain is called a **cross-compiler**.

2.5.2 Motivation

Cross-compilers are most often used to compile for platforms that either lack a good native development environment or where compiling directly on the target platform would be prohibitively slow or cumbersome. The output of a cross-compiler might also be run in an emulator in cases where giving each developer an instance of the target hardware would be too expensive, or where an emulator gives some other advantage during development (e.g., easier inspection of system state).

2.5.3 Obtaining a cross-toolchain

There are many prebuilt (cross-)toolchains available[14], as well as software packages that can build toolchains for a wide variety of host and target platforms. In this thesis **Buildroot**[2] will be used to build cross-toolchains.

Many toolchains, including those produced by Buildroot, are based on the **GCC** (**GNU Compiler Collection**) suite of compilers and tools. The design of GCC is such that a new set of executables (a new toolchain) usually needs to be produced for each target. This thesis will exclusively use GCC-based toolchains.

[14] A popular GCC-based set of prebuilt toolchains is the CodeSourcery family: `http://www.codesourcery.com` (*retrieved Dec. 15, 2011*)

Chapter 3

Design and implementation

3.1 Overview

This chapter describes the design and implementation of Awesom-O. It begins with a high-level design overview, and then dives into the nitty-gritty with more detailed descriptions of the tools used as well as several annotated code listings.

3.2 High-level design

Awesom-O is primarily an experiment in disk footprint and boot-time optimization for Linux-based, special-purpose systems. It does not aim be usable as a general purpose Linux distribution, and only provides the infrastructure needed to run the Devices SDK.

In particular, Awesom-O does not strictly adhere to any filesystem layout standard such as Linux's **FHS (Filesystem Hierarchy Standard)**[17], for no other reasons than that it would complicate the system somewhat.

3.2.1 Kernel and drivers

In this thesis Awesom-O will only run on systems where we know precisely what hardware is installed. In such a scenario it usually makes most sense to only include precisely the drivers needed and to exclusively use compiled-in drivers[1]. In this way no time is wasted probing for non-existent hardware, and if all drivers are compiled-in we might even be able to omit module support from the kernel altogether[2].

Depending on system configuration (PC from disk, PC from USB, or BeagleBoard), different strategies will have to be used for mounting the root filesystem. This is covered in the implementation section.

The first Awesom-O-specific process that runs is its `init` process:

[1] General-purpose distributions that ship with a precompiled kernel instead listen for events sent out by the kernel when hardware devices are detected and load the corresponding drivers as modules in response.

[2] Unless we want to delay the loading of certain drivers or kernel subsystems. For an example of why this might be desirable, see section 6.1.7 — *Late-loading of drivers and subsystems*.

3.2.2 `/bin/start`—Awesom-O's `init` process

Awesom-O uses its own minimal `init` process: `/bin/start`. This process performs various initialization (see the implementation section for the nitty-gritty) before finally spawning a shell to run the shell script `/etc/initscript` and going to sleep (due to its special role the `init` process must never exit).

3.2.3 `/etc/initscript`—Awesom-O's initialization script

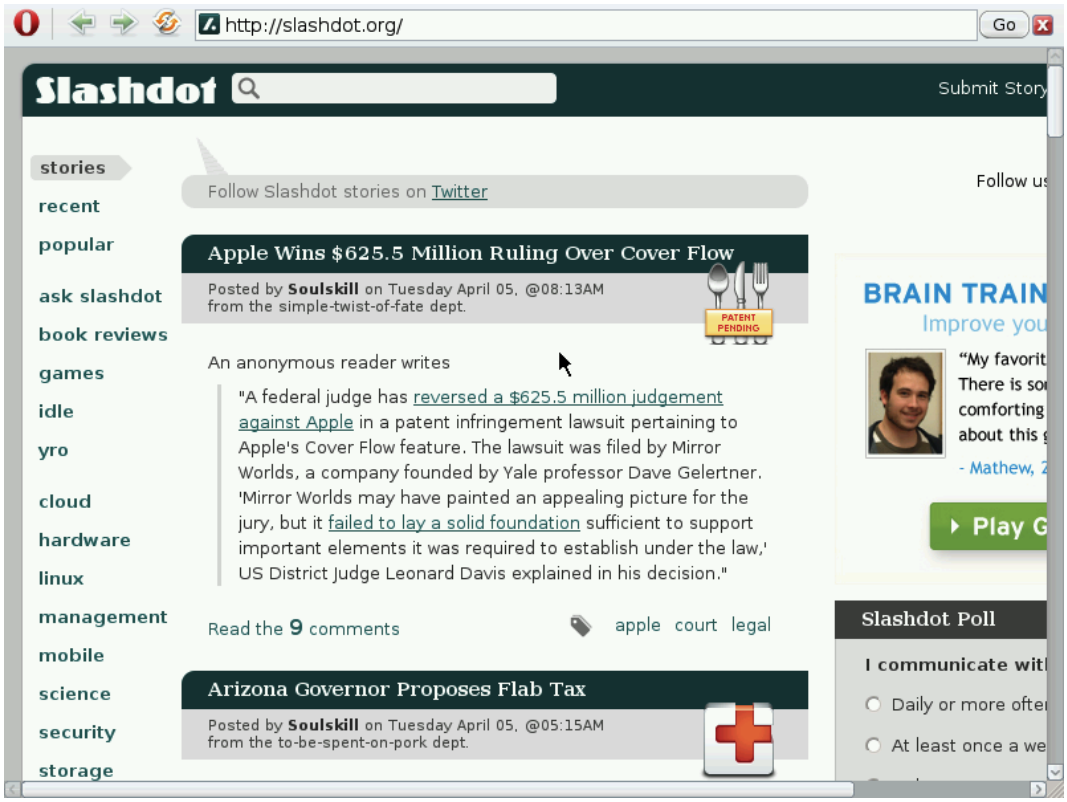
This script is run by `/bin/start`. It mounts the special **proc** and **sys** filesystems; launches a DHCP client for configuring network interfaces; sets values of environment variables used by the Devices SDK; and finally launches `/bin/opera`: a simple web browser implemented atop the Devices SDK.

3.2.4 `/bin/shutdown`—Shutting down Awesom-O

`/bin/shutdown` is used to shut down Awesom-O in an orderly fashion. This program sends *SIGTERM* (and *SIGKILL* if need be) to all non-`init` running processes to instruct them to quit, flushes disk caches, and then shuts down the system via the `reboot(2)` system call.

3.2.5 `/bin/opera`—Awesom-O's user interface and web browser functionality

The `/bin/opera` program is a simple web browser implemented on top of the Devices SDK and DirectFB (described in the next section). A screenshot appears below:



3.2.6 Software packages used

Besides the Linux kernel and the Devices SDK, Awesom-O makes use of the following third-party software packages:

Buildroot A collection of makefiles and patches for building a wide range of software packages for various platforms, primarily targeting embedded systems. It is used to build the remaining packages in this list, and also for building the (cross-)toolchain that in turn builds the Linux kernel and `/bin/{start,shutdown,opera}`.

uClibc An implementation of the standard C library optimized for small size, often used on embedded platforms. This is the default C library implementation used by Buildroot (that is, the C library that software built through Buildroot links against).

BusyBox A collection of tiny versions of common Unix utilities, bundled together into a single executable file. BusyBox provides the shell (`ash`), DHCP client (`udhcpc`), and various utilities such as `mount` used internally in Awesom-O.

DirectFB A library for doing advanced graphics on the **Linux framebuffer device** (documented in `Documentation/fb/framebuffer.txt` in the kernel source tree), supporting hardware acceleration given the right driver support. It is used by `/bin/opera` to draw graphics on the screen, and also for its input device handling capabilities.

3.3 Implementation

This section elaborates on the previous one and describes the implementation of Awesom-O in detail. It takes the form of instructions for putting the system together for our three target platforms (see section 1.6 — *Scope* for a reminder of what the different target platforms are).

3.3.1 Building toolchains and packages with Buildroot

Buildroot will be used to build the GCC toolchains and third-party software packages we need[3]. As the precise locations and names within the configuration interface of the configuration options mentioned below are bound to (and have been observed to) differ between Buildroot versions, we will only describe what needs to be enabled in general terms.

The Buildroot configuration interface is invoked by running `make menuconfig`[4] in the top-level Buildroot source directory. The configurations for our two target platforms will only differ in how the *Target Architecture* and *Target Architecture variant* options are set:

- For PC, we use *i386* with the variant *i686*.
- For BeagleBoard, we use *arm* with the variant *cortex-A8*[5].

[3] Buildroot can optionally produce a complete root filesystem image (hence the name), but we will not make use of this option as our root filesystem is non-traditional and extremely simple.

[4] Buildroot uses Kconfig, the same configuration system used by the Linux kernel.

[5] This requires GCC version 4.4.x or newer. As of Buildroot 2010.11, a GCC 4.3.x version is the default.

In the *Toolchain* subsection of the configuration interface we will need to include the following, as they are needed by the Devices SDK:

- A C++ compiler (`g++`).
- Support for wide (WCHAR) characters.
- Support for large (> 2 GiB) files in C standard library routines.

In the *Package Selection for the target* section we choose to build DirectFB and BusyBox (the former found under *Graphic libraries and applications* as of writing). We go over detailed configuration of these separately.

Configuring DirectFB

As the focus of the thesis is not on graphics performance we will not make use of the advanced hardware acceleration capabilities of DirectFB, and so need not include any GPU-specific drivers.

In addition to being used for drawing graphics on the screen, the input device handling capabilities of DirectFB will provide the mouse and keyboard support in `/bin/opera`. We use the **/dev/input/eventX** input driver, which reads input events from the Linux Input Subsystem (documented in `Documentation/input/input.txt` in the Linux Kernel source tree).

Configuring BusyBox

BusyBox by default includes hundreds of utilities, of which we only need a small subset. To save space we will use our own BusyBox configuration that includes just the things we need (plus a few utilities that are handy during development), as follows:

1. We perform an initial build of the toolchain and packages by issuing `make` in the top-level Buildroot directory.
2. We run `make busybox-menuconfig`[6], which invokes BusyBox's configuration interface.
3. Once we are done configuring BusyBox we save the configuration and exit the configuration interface.

The following utilities are included in Awesom-O's BusyBox configuration (the set could be narrowed down further as some of these are only included as development aids): `ash`, `cat`, `cp`, `date`, `dmesg`, `fbset`, `find`, `grep`, `ifconfig`, `ip`, `kill`, `ln`, `ls`, `mkdir`, `mknod`, `mount`, `mv`, `ping`, `rm`, `sed`, `sleep`, `stty`, `sync`, `top`, `touch`, `udhcpc`, `umount`, and `uname`.

In addition, the config script path for `udhcpc` should be set to '`/etc/dhcpevent`'.

[6]BusyBox as well as uClibc also use Kconfig.

Building and output

We are now ready to build the toolchain and packages using our custom configuration. This is done by issuing the `make` command in the top-level Buildroot directory. Buildroot places build output in the `output/` directory, where two subdirectories are of interest to us:

output/host/ This directory has a structure similar to a Unix root filesystem and contains utilities for the host system. It is of interest to us as it contains the toolchain in `output/host/usr/bin/`.

output/target/ This directory contains the root filesystem generated by Buildroot, from which a filesystem image can be generated. As our needs are very simple and our root filesystem layout non-traditional we will not make use of the option to generate a filesystem image, and will instead copy files out of this directory ourselves.

Per convention, the executables in the cross-toolchain have their names prefixed with a string of the form '*<CPU type>-<Manufacturer>-<Operating system>-*', where not all sections may be present. For example, the name the C++ compiler executable is `i686-linux-g++` when targeting x86 (i686) and `arm-linux-g++` when targeting ARM.

Having built a toolchain, we are now ready to build kernels for our target platforms. We will go over the process for PC and BeagleBoard separately.

3.3.2 Building a kernel for PC

We begin by downloading the kernel source (as of this report, Linux 2.6.37.1 was used) from `http://kernel.org` and decompressing it somewhere. To avoid picking up settings from a default configuration on the local system (which is likely to enable many things we do not need), we first run `make defconfig` (see section 2.4 — *Configuring and building a Linux kernel*). The kernel is then configured by running `make menuconfig` in the top-level source directory.

For an initial version of configuration of the kernel, the default kernel configuration will likely mostly suffice. We will need to enable support for VESA framebuffers (*Device Drivers→Graphics support→Support for frame buffer devices→VESA VGA Graphics support*), as well as support for whatever network interface hardware we are using. Once we are happy with the configuration, we exit the menu configuration interface and save our configuration.

If we were to use a native toolchain shipping with the system at this point—which would probably work in the common case where the development machine is also a PC, as the Linux kernel does not use the standard C library—we would simply have to run `make` to build the kernel. For consistency we will instead use the Buildroot toolchain, which turns the command into `make CROSS_COMPILE=`*<prefix>* `ARCH=`*<arch>*, where

1. *<prefix>* is the common prefix of the executables in the toolchain to be used. We need to make sure they are included in the shell's search path (*PATH*).

2. *<arch>* is a string identifying the platform for which to build a kernel. Often it should be the name of the corresponding `arch/` subdirectory in the kernel source tree, but variations exist[7].

[7] See the "Additional ARCH settings for..." comments in the top-level kernel `Makefile`.

If we wanted to build a 32-bit x86 kernel this would be something like[8]

```
$ make CROSS_COMPILE=i686-linux- ARCH=i386
```

The resulting kernel appears in `arch/x86/boot/bzImage`.

3.3.3 Building a kernel for BeagleBoard

Building a Linux kernel that will run on the BeagleBoard is somewhat more involved. We will use the **Linux-OMAP**[19] branch of the Linux kernel as it includes the latest OMAP support (the family to which the BeagleBoard's CPU belongs), and will add patches from the **OpenEmbedded**[20] project that, among other things, improve Linux framebuffer support on the BeagleBoard (as of writing (*Dec. 15, 2011*), these patches are required to even get a kernel that boots on the BeagleBoard). We will use the kernel configuration from the BeagleBoard version of the popular **Angstrom Linux distribution**[16]. As the details are long, technical, and fairly version-specific, they will not be duplicated here; they can be found at `http://elinux.org/BeagleBoardLinuxKernel` (*retrieved Dec. 15, 2011*). One common mistake to look out for is forgetting to apply the patches against the precise version of the OMAP kernel against which they were generated; pay attention to *SRCREV*.

Once the kernel has been patched and configured it can be built similarly to the x86 kernel. After making sure to run `make menuconfig` at least once, we issue the following command to build the kernel:

```
$ make CROSS_COMPILE=arm-linux- ARCH=arm
```

If everything goes well, the resulting kernel image will appear as `arch/arm/boot/uImage`.

Next we describe `/bin/start`, Awesom-O's init process.

3.4 `/bin/start` - the Awesom-O init process

Awesom-O uses its own `init` process, found in `/bin/start`. The source code appears in the following listing, followed by a walkthrough of some of the darker corners.

```
1  #include <errno.h>
2  #include <fcntl.h>
3  #include <signal.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8  #include <time.h>
9  #include <unistd.h>
10
11 // The TTY (terminal device) to which the standard streams of the
12 // spawned shell should be bound; this also becomes the controlling TTY
13 // of the shell. Using the startup default of /dev/console would break
14 // job control (Ctrl-C, etc.) in the shell as /dev/console can not be
15 // used as a controlling terminal.
16 //
```

[8] On CPUs that have multiple cores and/or support hyperthreading, the *-j* option for running builds in parallel can speed up the build process significantly.

```
17 // /dev/tty1 is the first Linux virtual console.
18 #define TTY "/dev/tty1"
19
20 // The initialization script to run
21 #define INITSCRIPT "/etc/initscript"
22
23 // execve() arguments for running INITSCRIPT
24 static const char *initscript_args[] = {"/bin/busybox", "ash", INITSCRIPT,
25                                         NULL};
26
27 // execve() arguments for starting an interactive shell
28 static const char *ishell_args[] = {"/bin/busybox", "ash", NULL};
29
30 // Empty environment for execve() calls
31 static const char *empty_env[] = {NULL};
32
33 static void fail(const char *msg) __attribute__ ((noreturn));
34 static void go_to_sleep() __attribute__ ((noreturn));
35
36 int main(int argc, char *argv[]) {
37     // SIGCHLD is a signal sent when a child process exits. As we do not
38     // care about the exit status of our children at this point we
39     // explicitly ignore this signal, which according to POSIX.1-2001,
40     // honored by Linux 2.6, will prevent child processes from becoming
41     // zombies.
42
43     // Use handy GCC-specific struct initialization syntax
44     struct sigaction act = { .sa_handler = SIG_IGN };
45     if( sigaction(SIGCHLD, &act, NULL) < 0 )
46         fail("Could not ignore SIGCHLD");
47
48     // Create a new session; we need this for job control in the shell.
49     if( setsid() < 0 )
50         fail("Could not create new session");
51
52     // Rebind standard streams to TTY
53
54     // First close the old stdin/out/err (probably bound to /dev/console)
55     close(0); close(1); close(2);
56
57     // Ignore any errors from above
58     errno = 0;
59
60     // This will use the lowest available file descriptor, i.e. 0 (=
61     // stdin), so the effect is to bind stdin to TTY. TTY will also become
62     // the controlling terminal.
63     if( open(TTY, O_RDWR | O_NONBLOCK, 0) != 0 )
64         fail("Could not open " TTY " as stdin");
65
66     // Make stdout and stderr point to the same place as stdin by
67     // duplicating the stdin file descriptor. dup() always uses the lowest
68     // available descriptor, so this will bind 1 (stdout) and 2 (stderr),
69     // in that order.
70     if( dup(0) != 1 ) // Assigns stdout
71         fail("Failed to reassign stdout to " TTY);
```

```
72      if( dup(0) != 2 ) // Assigns stderr
73          fail("Failed to reassign stderr to " TTY);
74
75      // Standard streams rebound. Now run INITSCRIPT or spawn an interactive
76      // shell.
77
78      if( argc == 2 && !strcmp(argv[1], "ishell") ) {
79          // Spawn an interactive shell if "ishell" was passed on the kernel
80          // command line; this is sometimes handy during development.
81          if( !fork() )
82              if( execve("/bin/busybox", ishell_args, empty_env) == -1 )
83                  fail("Failed to launch interactive shell");
84      }
85      else {
86          // Run the init script
87          if( !fork() )
88              if( execve(INITSCRIPT, initscript_args, empty_env) == -1 )
89                  fail("Failed to run initialization script " INITSCRIPT);
90      }
91
92      // The init process must not die, and we shouldn't busy-wait (init
93      // using ~100% CPU would be bad), so go to sleep.
94      go_to_sleep();
95  }
96
97  static void fail(const char *msg) {
98      char error_msg[256];
99      ssize_t write_res;
100     int msglen;
101     int fd;
102
103     fd = open("/dev/console", O_WRONLY | O_NOCTTY);
104     if( fd == -1 )
105         go_to_sleep();
106
107     msglen =
108         snprintf(error_msg, sizeof(error_msg),
109                  "%s\nerrno = %d (%s)\n", msg, errno, strerror(errno));
110     write_res = write(fd, error_msg, msglen > sizeof(error_msg) ?
111                                      sizeof(error_msg) : msglen);
112     if( write_res != -1 ) {
113         // Write an extra newline just to be sure the message becomes
114         // visible in case it was truncated
115
116         // Assign write_res to suppress GCC warning
117         write_res = write(fd, "\n", 1);
118     }
119
120     // Just to be safe
121     fsync(fd);
122
123     close(fd);
124     go_to_sleep();
125 }
126
```

```
127 static void go_to_sleep() {
128     const struct timespec ts = {60, 0};
129
130     for(;;)
131         nanosleep(&ts, NULL);
132 }
```

The **/bin/start** init process goes through the following steps (with corresponding line numbers):

37-46 Ensure child processes of `init` do not become zombies[9]. This is especially important for `init` as it adopts any process whose parent process dies.

48-50 Create a new **session**. Sessions are collections of **process groups**, where a process group is a grouping of processes that allows all processes in the group to be sent the same signal. For example, when executing a pipeline of commands such as

```
$ grep stuff file | sed s/wrong/right/g | tee output
```

we want **CTRL-C**—corresponding to the *SIGINT* signal—to interrupt the `grep`, `sed` and `tee` processes, and similarly want **CTRL-Z** (*SIGTSTP*) to suspend all three processes. To implement this behavior, the shell puts the three processes into the same process group. As another example, if we have processes running in the background (**CTRL-Z** followed by `bg`) as well as processes running in the foreground, we only want signals to go to the foreground processes. This would be implemented by making the foreground processes the *foreground process group*. If we do not call `setsid()` the shell will not be able to create process groups, and no job control will be available.

Sessions typically have a **controlling TTY**, which represents the terminal used for the session (this notion becomes a bit vague with graphical user interfaces such as X11), to which the standard stdin, stdout, and stderr streams will typically be bound by default. Key combinations such as **CTRL-C** are actually intercepted by the TTY driver[10], which sends the corresponding signal to all processes in the current foreground process group of the session[11].

54-73 Bind stdin/stdout/stderr to `/dev/tty1`, the first Linux virtual console. The Linux virtual consoles are fullscreen text terminals, usually visible before a graphical interface such as X11 has been launched[12]. Initially, the init process will likely have stdin/stdout/stderr bound to `/dev/console`, but this device can not be used as a controlling terminal (it is primarily intended as an output device for system messages).

86-89 Run the initialization script `/etc/initscript` in its own process. This script begins with the hashbang line `#!/bin/busybox ash`, instructing the kernel to run it in BusyBox's `ash` shell[13].

[9] A zombie is a process that has finished executing and has had its resources deallocated but that still holds an entry in the process table so that its parent process can collect its exit status.

[10] The mapping from key combinations to signals can be altered via the `stty` command.

[11] This is a simplified description; see [4] for the nitty-gritty, which also covers exactly how a process gets a controlling TTY.

[12] When backed by a driver capable of graphical output, virtual consoles often initially display the Linux penguin logo in the upper-left corner.

[13] It is in fact the *kernel* that interprets the hashbang line, inside the `execve()` system call.

127-132 Due to its special role, the init process must not exit (if it does we get a kernel panic), so instead we go to sleep. It is important that we do not simply busy-wait without sleeping at this point, as that would make our init process use up ~100% CPU while the system is running.

3.4.1 Building `/bin/start`

When building for PC, we compile `/bin/start` with the following command:

```
$ i686-linux-gcc -Os -s -march=i686 start.c -o start
```

The `-Os` flag instructs GCC to optimize for size.

When building for BeagleBoard, we compile `/bin/start` with

```
$ arm-linux-gcc -Os \
>               -s \
>               -march=armv7-a -mtune=cortex-a8 \
>               -mfpu=neon -mfloat-abi=softfp \
>               start.c -o start
```

The *-march* and *-mtune* flags tell GCC what ARM CPU to generate code and optimize for, respectively. The *-mfpu* option informs GCC what floating-point hardware is available. Though most of these optimization flags will not make any difference when compiling `/bin/start`, we use them here for consistency. Refer the GCC documentation for details.

3.5 `/bin/shutdown` - Shutting down Awesom-O

`/bin/shutdown` is responsible for shutting down Awesom-O in an orderly fashion. The source code appears in the following figure.

```
1  #include <errno.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdlib.h>
6  #include <sys/mount.h>
7  #include <sys/types.h>
8  #include <linux/reboot.h>
9  #include <unistd.h>
10
11 static void fail(const char *msg) __attribute__ ((noreturn));
12
13 int main() {
14     puts("Sending SIGTERM to all processes..");
15     if( kill(-1, SIGTERM) < 0 )
16         fail("Failed to send SIGTERM to all processes");
17
18     // Sleep for a second to give processes time to quit. We assume all
19     // processes terminate quickly in Awesom-O (except perhaps for the
20     // opera process, which shouldn't be running when we invoke shutdown
21     // anyway).
22     sleep(1);
23
24     puts("Killing off any remaining processes with SIGKILL..");
```

```
25      if( kill(-1, SIGKILL) < 0 )
26          fail("Failed to kill processes with SIGKILL");
27
28      // Remount root read-only and then flush filesystem buffers, to make
29      // sure everything gets out to disk
30
31      puts("Remounting root read-only...");
32      if( mount("", "/", NULL, MS_REMOUNT | MS_RDONLY, NULL) < 0 )
33          fail("Failed to remount root read-only");
34
35      puts("Flushing filesystem buffers...");
36      sync();
37
38      puts("Shutting down...");
39      // This will programmatically turn off the power if the system supports
40      // it (PC usually does provided ACPI support has been compiled in;
41      // BeagleBoard does not)
42      reboot(LINUX_REBOOT_CMD_POWER_OFF);
43
44      // reboot() will not return upon success, so us being here means
45      // something went wrong
46      fail("Shutdown failed");
47
48      return 1;
49  }
50
51  static void fail(const char *msg) {
52      fprintf(stderr, "%s: %s\n", msg, strerror(errno));
53      exit(1);
54  }
```

3.5.1 Building /bin/shutdown

We compile **/bin/shutdown** in the same way we did **/bin/start**. For PC, we do

```
$ i686-linux-gcc -Os -s -march=i686 shutdown.c -o shutdown
```

For BeagleBoard, we use

```
$ arm-linux-gcc -Os \
>               -s \
>               -march=armv7-a -mtune=cortex-a8 \
>               -mfpu=neon -mfloat-abi=softfp \
>               shutdown.c -o shutdown
```

3.6 The /etc/initscript script

Unless *ishell* is passed on the kernel command line, `/bin/start` will run the `/etc/initscript` script from the following figure:

```
1  #!/bin/busybox ash
2
3  # Mount /proc and /sys
4  /bin/busybox mount -t proc proc /proc
```

```
5  /bin/busybox mount -t sysfs sysfs /sys
6
7  # Mount a tmpfs on /tmp. This filesystem lives in system memory.
8  /bin/busybox mount -t tmpfs tmp /tmp
9
10 # /bin/links contains symbolic links with the names of commands, all
11 # linked to /bin/busybox. This makes running commands from the shell a bit
12 # nicer.
13 export PATH=/bin:/bin/links
14
15 # Launch the udhcpc daemon, which configures interfaces via DHCP
16 /bin/busybox udhcpc &
17
18 # Raise the loopback interface. The ethernet interface is raised in
19 # /etc/dhcpevent, called by udhcpc.
20 /bin/busybox ip link set dev lo up
21
22 # Resources needed by the Devices SDK
23 export OPERA_DIR=/opera_dir
24 export OPERA_HOME=/opera_home
25 export OPERA_FONTS=/fonts
26
27 # Launch the Awesom-O interface
28 /bin/opera -u file:///resources/start-page.html
29
30 # Launch an interactive shell after /bin/opera exits (handy during
31 # development)
32 exec /bin/busybox ash
```

/etc/initscript does the following:

3-5 Mount the **procfs** and **sysfs** filesystems on `/proc` and `/sys`. These are "pseudo filesystems" populated by the kernel and drivers, allowing system state to be queried and updated by reading and writing files. `procfs` primarily contains information about running processes, while `sysfs` contains information about devices and device drivers[14].

These are needed by parts of the C library as well as by several BusyBox utilities, including `udhcpc`.

7-8 Mount a **tmpfs** on `/tmp`. Many programs expect to be able to write temporary files into this directory.

10-13 Set up the shell search path. The `links` directory contains a set of symbolic links, all pointing to `/bin/busybox`. This lets us avoid having to prefix every command with "`busybox`" when we are working interactively (during development). BusyBox infers what utility to run from the name it was invoked under (this becomes `argv[0]` inside the `main()` function).

15-16 Launch the udhcpc DHCP client in order to configure network interfaces; see the next section.

[14] In practise the division is not always this clean, and `procfs` in particular contains many files that would more logically go in `sysfs`. This is for historical reasons, `procfs` having appeared before `sysfs`.

22-25 Set environment variables referring to the locations of resources needed by the Devices SDK; `/bin/opera` reads these.

3.7 The udhcpc configuration file

For simplicity and flexibility the `udhcpc` DHCP client does not know how to configure network interfaces, but instead relies on a user-supplied script to do this. This script is invoked whenever important DHCP-related events occur, with information about the event passed in the environment. In Awesom-O, this script is `/etc/dhcpevent`, which appears in the following figure, followed by a walkthrough.

```
1  #!/bin/busybox ash
2  dnsfile=/etc/resolv.conf
3
4  case $1 in
5    deconfig )
6      # Raise interface
7      /bin/busybox ip link set dev $interface up
8
9      # Deconfigure interface
10     /bin/busybox ip addr flush dev $interface
11     ;;
12   bound|renew )
13     # Configure interface
14     /bin/busybox ifconfig $interface $ip netmask $subnet
15
16     # Add default routes
17     if [ -n "$router" ]; then
18         /bin/busybox ip route del default
19         /bin/busybox ip route add default via $router
20     fi
21
22     # Clear the list of DNS servers
23     > $dnsfile
24
25     # Add domain
26     [ -n "$domain" ] && echo "search $domain" > $dnsfile
27
28     # Register DNS servers
29     for server in $dns; do
30         echo nameserver $server >> $dnsfile
31         echo Added DNS server $server
32     done
33     ;;
34 esac
```

The argument to the script ($1) describes the type of event. Information related to events is passed in environment variables (e.g. *interface*, *ip*, and *netmask*).

5-11 The *deconfig* event is sent when `udhcpc` is first started, as well as when a DHCP lease is lost[15]. Here we put the interface (`$interface`) in an *up* (activated, sometimes also

[15] A DHCP lease represents the right to use a particular IP address and configuration. As long as the DHCP lease is valid, the DHCP server will ensure the configuration does not conflict with that of any other client.

referred to as "raised") state and deconfigure it so that it will not keep using any old (expired) address it might have.

12-33 The *bound* event is sent when the interface receives a new IP address, while the *renew* event is sent when we successfully negotiated to keep our previous IP address after the DHCP lease expired. In both cases we configure the interface and register new DNS servers and gateways[16].

3.8 The `/bin/opera` executable

The `/bin/opera` executable provides user interface and web browser functionality in Awesom-O. To save space, it is statically linked against the Devices SDK and DirectFB libraries[17]. For simplicity, the DirectFB example implementation that comes with the Devices SDK will be used, with the precise implementation and compilation of `/bin/opera` being outside the scope of this thesis.

3.9 Assembling a working Awesom-O system

We are now ready to assemble the pieces from above into a working Awesom-O system. Depending on the platform (see section 1.6 — *Scope*), we will need to go about this in different ways.

3.9.1 Creating an empty root filesystem

We first need to create an empty filesystem to host the Awesom-O root filesystem files. The procedure differs between our target platforms. We will use the ext2 filesystem throughout[18].

- On PC booting from disk we simply create a new partition (using **fdisk** or another disk partitioning utility) and create an ext2 filesystem on it using **mke2fs**. If for example the device node of the new partition is `/dev/sda1`, we run

  ```
  $ mke2fs -L Awesom-O /dev/sda1
  ```

  We then mount the new filesystem in order to populate it[19]:

  ```
  $ mkdir rootfs
  $ mount /dev/sda1 rootfs
  $ cd rootfs
  ... (populate root filesystem)
  $ cd ..
  $ umount rootfs
  ```

[16] Gateways represents routes to addresses outside of the local network—commonly the Internet.

[17] This saves space as the linker can omit code for functions that are never invoked when linking statically.

[18] As mentioned in section 1.7.4 — *Filesystems*, a filesystem designed for Flash devices such as ubifs or jffs2 would be a better choice for a production system on the BeagleBoard.

[19] Some of these commands might require superuser privileges.

- The same procedure will be used to create the root filesystem for booting from USB on PC, only substituting device nodes corresponding to the USB device[20]. The root filesystem contents will differ slightly, as explained later.

- For BeagleBoard we will create an ext2 filesystem image on the development machine which we then write to NAND Flash. The following set of commands can be used to create a 70 MiB ext2 image:

```
$ dd if=/dev/zero of=disk.img bs=1M count=70
$ losetup /dev/loop0 disk.img
$ mke2fs -L Awesom-O /dev/loop0
$ mkdir rootfs
$ mount /dev/loop0 rootfs
$ cd rootfs
... (populate root filesystem)
$ cd ..
$ umount rootfs
$ losetup -d /dev/loop0
$ sync
```

  The commands above make use of a **loopback device**, which allows us to bind a block device (`/dev/loop0` above) to the contents of a file (`disk.img`). The final `sync` command ensures that all data has been written out to our filesystem image before we copy it onto the SD/MMC which we use to transfer the image to the BeagleBoard.

  How to write the ext2 image to NAND Flash is covered later, in section 3.11 — *Kernel and bootloader installation on BeagleBoard*.

3.9.2 Populating the root filesystem

We now proceed to populate the root filesystem. We will make use of the following shorthand:

- `$BR` refers to the Buildroot top-level directory.

- `$BIN` refers to the directory where the `start` and `shutdown` executables were placed on the development machine.

- `$SCRIPTS` refers to the directory containing the `initscript` and `dhcpevent` scripts on the development machine.

The values of `$BIN` and `$BR` will naturally differ depending on whether we are assembling a root filesystem for PC or the BeagleBoard.

The commands in the following sections should be run from one of the `rootfs` directories created above.

[20] An easy way to figure out which node corresponds to the USB mass storage device is to check `dmesg` output after connecting it.

Installing shared libraries

We first create an populate `/lib`, containing the shared libraries we need.

```
$ mkdir lib
$ for library in $BR/output/target/lib/\
> {libc.so.0,libm.so.0,ld-uClibc.so.0,libgcc_s.so.1}; do
>   cp -d $library $(readlink -f $library) lib
> done
```

The `for` loop loops through the set of libraries, copying them from the Buildroot output directory into our root filesystem. The files listed within braces above are actually symbolic links to the real library files, and we need to copy those as well. This is done using `readlink`, which resolves symbolic links to the filenames of their targets. The *-d* flag to `cp` turns off symbolic link dereferencing to allow us to copy the symbolic link itself rather than what it points to.

The reason for having symbolic links refer to the actual library files is to allow for several incompatible library versions to coexist on a system. The number after the period in the name of the symbolic link is the library's **major version**, and is usually incremented only when backwards-incompatible changes are introduced. The name of the actual library file usually contains additional versioning information, allowing the precise version to be identified[21].

The libraries installed are the following:

libc.so.0 The main uClibc library.

libm.so.0 A collection of math routines that appear in a separate library for historical reasons (many of them *are* part of the ANSI C standard library).

ld-uClibc.so.0 The uClibc dynamic linker for linking in shared libraries at runtime.

libgcc_s.so.0 A library with various helper routines—mostly for arithmetic operations that are complicated to carry out on the target CPU. GCC automatically generates calls to methods in this library as needed.

3.9.3 Creating device nodes

Next, we create the device nodes in the `/dev` directory, using *Documentation/devices.txt* from the Linux source tree as a reference for major and minor numbers.

```
$ mkdir dev
$ mknod dev/console c 5 1
$ mknod dev/fb0 c 29 0
...
$ mknod dev/tty2 c 4 2
```

As noted earlier, device nodes are completely characterized by their type and minor and major numbers, so it is perfectly safe to create the nodes on the development machine (as long as differences in available users and groups are accounted for). We create the following device nodes:

[21] A little-known fact is that many libraries also print out versioning information when run as executable files.

`/dev/console` The Linux console, primarily used as an output device for system messages. The kernel parameter *console* can be used to specify where data written to this device will go.

`/dev/fb0` The first framebuffer device, onto which graphics will be rendered. If we had multiple framebuffers (e.g. for multiple video adapters) we would have additional `fb` device nodes.

`/dev/tty0` The current virtual console (the one that is visible unless some graphical application such as X11 has control of the screen).

`/dev/{tty1,tty2}` The first and second virtual consoles (needed by DirectFB).

`/dev/tty` The "foreground" terminal, corresponding to the controlling TTY of the currently running process.

`/dev/input/event{0-6}` Linux Input Subsystem event queues. We use these to handle mouse and keyboard input inside `/bin/opera`.

3.9.4 Creating mount points for `proc` and `sys`

We need to create mount points for procfs and sysfs. By convention these should be named `/proc` and `/sys`, respectively. `/etc/initscript` will mount procfs and sysfs on these directories during boot.

```
$ mkdir proc sys
```

3.9.5 Creating `/tmp`

Many utilities expect `/tmp` to be available as a location for writing temporary files. During boot a `tmpfs` filesystem, residing entirely in system memory, will be mounted on this directory.

```
$ mkdir tmp
```

3.9.6 Populating `/etc`

The **/etc** directory is used to store the `initscript` and `dhcpevent` shell scripts (see section 3.6 — *The `/etc/initscript` script*):

```
$ mkdir etc
$ cp $SCRIPTS/{initscript,dhcpevent} etc
```

3.9.7 Populating `/bin`

We put our init process `start`, our shutdown program `shutdown`, the BusyBox executable `/bin/busybox`, and the `opera` executable into the `/bin` directory:

```
$ mkdir bin
$ cp $BIN/{start,shutdown,opera} \
>    $BR/output/build/busybox-1.15.3/busybox \
>    bin
```

3.9.8 Installing various resources needed by the Opera Linux Device SDK

The Opera Linux SDK needs various resources such as fonts, language files, and JavaScript files. These can be found in binary distributions of the Opera Linux SDK. The directories needed are `opera_dir`, `opera_home` and `fonts`:

```
$ cp -r <path to opera_dir> .
$ cp -r <path to opera_home> .
$ cp -r <path to fonts directory> .
```

Next, we need to install the kernel and bootloader. The procedure differs between our target platforms, and each will be covered separately.

3.10 Kernel and bootloader installation on PC

In the following, `$KERNEL` will refer to the top-level directory of the 2.6.37.1 kernel tree in which we built the kernel. We use the Grub 0.97[9] bootloader on PC, configured differently depending on if we boot from disk or USB.

3.10.1 Booting from disk

We first install the kernel into the `boot` directory of the target root filesystem:

```
$ mkdir boot
$ cp $KERNEL/arch/x86/boot/bzImage boot
```

The following Grub configuration will be used. It should be saved as `boot/menu.lst`, which is where the stage 2 Grub binary expects to find it.

```
default 0
timeout 0

title Awesom-O (from disk)
root (hd0,0)
kernel /boot/linux-2.6.37.1 root=/dev/sda1 init=/bin/start \
       video=vesa:ywrap,mtrr:3 vga=0x318 rw quiet
```

The `title` command introduces an alternative for booting the system, with the lines that follow it being instructions for how to carry out the boot. The `default 0` and `timeout 0` commands instruct Grub to use the first boot alternative listed by default, and to boot immediately without presenting a menu of boot alternatives to the user; this has the effect of immediately booting Awesom-O.

The first argument to the **kernel** command specifies the kernel image to boot, with the remaining arguments being the kernel command line. The following parameters are used:

root=/dev/sda1 Specifies the device containing the root filesystem[22]. In the following we will assume this device to be `/dev/sda1`, but the correct device will vary depending on setup.

[22] See footnote in section 2.3.6 — *Mounting the root filesystem.*

init=/bin/start The init process to run. We use `/bin/start`, our own init process.

video=vesa:ywrap,mtrr:3 This sets up the Linux framebuffer to use the VESA 2.0 standard, supported on modern PC graphics hardware. **ywrap** enables support for panning by moving the "viewport" on video memory instead of copying data, and specifies that reading/writing past the end of video memory wraps around to the beginning. **mtrr:3** specifies that the CPU should use write-combining when writing to the portion of RAM into which the framebuffer is memory-mapped, which can speed up graphics operations significantly. The documentation for vesafb appears in `Documentation/fb/vesafb.txt` in the Linux source tree.

vga=0x318 This selects the video mode; a reference is available in `Documentation/fb/vesafb.txt`. 0x318 is 1024x768x32.

rw Mount the root filesystem read-write. By default, the Linux kernel will mount the root filesystem read-only in preparation for running an integrity check with `fsck`, which should not be executed on a filesystem mounted read-write. The early version of Awesom-O described in this report does not do filesystem integrity checks, so we pass *rw* to avoid having to remount the root filesystem read-write later.

quiet Turns off the displaying of kernel messages on the system console during boot. Displaying boot messages can often increase boot time significantly (by 1-1.5 seconds for Awesom-O) as updating the console takes time—especially when using a high-resolution graphical console running on the Linux framebuffer device. If we wish to view the boot messages later, we can use the `dmesg` command.

We save the Grub configuration as `/boot/grub/menu.lst` in the target root filesystem, which is what Grub expects.

Finally we need to install the stage1, stage1.5, and stage2 Grub binaries. This is done with the following command (assuming the root filesystem is mounted on `rootfs/` and the disk's device node is `/dev/sda`)[23]:

```
$ grub-install --root-directory=rootfs /dev/sda
```

This completes the installation of Awesom-O on disk.

3.10.2 Booting from USB

Booting from a USB device is trickier than booting from disk, for two reasons:

1. The Linux USB subsystem runs asynchronously and often finishes device initialization at a later time than when the kernel first attempts to mount the root filesystem. This manifests itself as a kernel panic upon boot ("*Kernel panic: VFS: Unable to mount root*").

[23] `grub-install` usually works in practise, but relies on guessing how BIOS routines map drives. To be perfectly safe we should enter the Grub shell `grub` and run e.g. the `find` command to figure out the correct drive mapping.

2. USB devices are often assigned device nodes in an unpredictable fashion. Worse yet, we might have several USB devices plugged in and not know which device node corresponds to our desired boot device. Linux generally does not make use of of the device mapping from the BIOS, so the fact that the USB device was selected as the boot device there does not help.

A solution to the first problem, but not the second, is to make use of the *rootdelay* and *rootwait* kernel parameters. These instruct the kernel to either wait a specified number of seconds (*rootdelay*) before attempting to mount the root filesystem, or to enter a loop and attempt to mount the root over and over again indefinitely (*rootwait*).

A more robust solution is to use an initramfs (see section 2.3.7 — *The initial RAM filesystem (initramfs/initrd)*). Here there are two options:

1. Let the initramfs be a temporary root filesystem, used only to run a script that figures out which device node corresponds to the USB device containing the root and remounts the root as that device.

2. Let the initramfs be the permanent root.

The second option has several drawbacks:

- The initramfs will use up system memory, only making this approach feasible for root filesystems that are very small compared to the total amount of system memory.

- For technical reasons involving how USB booting is handled in the BIOS, reading from USB devices during boot (before specialized drivers have been loaded) is often very slow, and so loading a large initramfs into memory might take a substantial amount of time. For USB 1.1 devices this would be true even assuming optimal transfer rates, as the theoretical upper bound on the transfer rate for USB 1.1 is just 1.5 MiB/s.

- Unless we are okay with all data being lost when the system is powered off, we will have to mount a separate filesystem residing on the USB device anyway, to be used for permanent storage.

For the reasons outlined above we will make use of the first approach, and will attempt to minimize the size of the initramfs to speed up the boot process. Our setup will be the following:

- The USB device will hold a single partition containing the root filesystem.

- Inside the root filesystem we store an archive containing an initramfs—`/boot/initramfs.cpio.gz`—which is loaded during boot. This initramfs contains a few device nodes (`/dev/console` plus the device node from which the root filesystem is mounted below in the `usbboot` source at a minimum), a directory `/root` onto which to mount the real root, and a small statically linked executable `usbboot`. `usbboot` mounts the real root from the USB device once it becomes available, switches to it via the `chdir()` and `chroot()` system calls, and then replaces itself with the real init process `/bin/start`.

The Grub configuration now becomes

```
default 0
timeout 10

title Awesom-O (from USB)
root (hd0,0)
kernel /boot/linux-2.6.37.1 rdinit=/usbboot \
       video=vesa:ywrap,mtrr:3 vga=0x314 rw quiet
initrd /boot/initramfs.cpio.gz
```

Compared to booting from disk, we now load an initramfs using the Grub command `initrd`, specify the init process from this initramfs to run with *rdinit*, and no longer pass a *root* parameter (as `usbboot` will mount the root).

The source code for `usbboot` appears in the following figure, followed by a description of some of the finer details:

```
1  #include <errno.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <termios.h>
8  #include <unistd.h>
9
10 // The "real" init process to replace ourself with
11 #define INIT "/bin/start"
12
13 // In case we fail to mount the root, the number of milliseconds to wait
14 // before trying again
15 #define MOUNT_RETRY_MS 100
16
17 // Turns a macro into a string corresponding to its expansion
18 #define STR(x) STR_(x)
19 #define STR_(x) #x
20
21 static void print_with_errno(const char *msg);
22 static void fail(const char *msg) __attribute__ ((noreturn));
23 static void go_to_sleep() __attribute__ ((noreturn));
24
25 // execve() arguments for running the real init process once we
26 // chdir() + chroot() into the root filesystem
27 static char *init_args[] = {INIT, NULL};
28
29 // Empty environment for execve() call
30 static char *empty_env[] = {NULL};
31
32 int main(int argc, char *argv[]) {
33     const struct timespec mount_retry_ts = {0, MOUNT_RETRY_MS * 1000000};
34     int failed_mounts = 0;
35     char msgbuf[100];
36     int msglen;
37     int serial_fd;
38     struct termios serial_opts;
39
```

```
40      while( mount("/dev/sdb1", "root/", "ext2", 0, NULL) < 0 ) {
41          print_with_errno("Could not mount the root filesystem - retrying "
42                           "in " STR(MOUNT_RETRY_MS) " milliseconds");
43          nanosleep(&mount_retry_ts, NULL);
44          ++failed_mounts;
45      }
46
47      // Remove files before chroot'ing into the new file system so that they
48      // do not use up memory unnecessarily
49
50      unlink("usbboot");
51
52      // Switch to the "real" root
53
54      if( chdir("/root") < 0 )
55          fail("Failed to chdir() into the root filesystem");
56
57      if( chroot("/root") < 0 )
58          fail("Failed to chroot() into the root filesystem");
59
60      // Record time spent waiting for root filesystem to become available
61
62      serial_fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);
63      tcgetattr(serial_fd, &serial_opts);
64      cfsetospeed(&serial_opts, B115200);
65      tcsetattr(serial_fd, TCSANOW, &serial_opts);
66      msglen = sprintf(msgbuf,
67                       "Waited about %d milliseconds for "
68                       "the root to become available\n",
69                       failed_mounts * MOUNT_RETRY_MS);
70      write(serial_fd, msgbuf, msglen);
71      close(serial_fd);
72
73      // Execute the real init process
74      execve(INIT, init_args, empty_env);
75
76      // The above should replace our process, so us being here means
77      // something went wrong
78      fail("Failed to execute the init process " INIT
79           " after chdir()+chroot()'ing into the root filesystem");
80  }
81
82  static void print_with_errno(const char *msg) {
83      char error_msg[256];
84      ssize_t write_res;
85      int msglen;
86      int fd;
87
88      fd = open("/dev/console", O_WRONLY | O_NOCTTY);
89      if( fd == -1 )
90          return;
91
92      msglen =
93          snprintf(error_msg, sizeof(error_msg),
94                   "%s\n errno = %d (%s)\n", msg, errno, strerror(errno));
```

```
95      write_res = write(fd, error_msg, msglen > sizeof(error_msg) ?
96                                       sizeof(error_msg) : msglen);
97      if( write_res != -1 ) {
98          // Write an extra newline just to be sure the message becomes
99          // visible in case it was truncated
100
101         // Assign write_res to suppress GCC warning
102         write_res = write(fd, "\n", 1);
103     }
104
105     // Just to be safe
106     fsync(fd);
107
108     close(fd);
109 }
110
111 static void fail(const char *msg) {
112     print_with_errno(msg);
113     go_to_sleep();
114 }
115
116 static void go_to_sleep() {
117     const struct timespec ts = {60, 0};
118
119     for(;;)
120         nanosleep(&ts, NULL);
121 }
```

`usbboot` does the following:

40-45 Attempt to mount the root over and over again until we succeed. Here the device is hardcoded as `/dev/sdb1` for simplicity and will likely be wrong on many systems and with many configurations. On a production system we would instead create the ext2 filesystem on the USB stick with a specific UUID (Universally Unique Identifier: a (practically) unique global identifier), and then mount the device with that UUID once it becomes available[24]. This could be accomplished for example by compiling a tiny statically-linked BusyBox executable with just `mount` support, pass the UUID to use on the kernel command line, and then invoke `mount` from `usbboot` with that UUID[25]. We would also be more specific about what types of errors we accept from `mount()`.

47-50 Remove the `usbboot` executable as it it no longer needed. This frees up a small amount of RAM as the initramfs lives in system memory.

54-58 Switch to the real root filesystem from the USB device, mounted on `/root`. The `chroot()` system call changes how absolute paths (paths starting with `/`) are resolved,

[24]Modern Linux distributions such as recent Ubuntu versions use UUIDs throughout to avoid problems with unpredictable/changing device-to-node mappings. Here the kernel command line might contain something like `root=UUID=018d2d9e-a6a4-493c-a2f4-f6cc879988e8`, which is actually interpreted by scripts running from the initramfs (recall that parameters not recognized by the kernel are passed on to the `init` process). Recent versions of `mount` usually support mounting filesystems by UUID.

[25]BusyBox's `mount` with a UUID argument only appears to scan `/dev`, so unless `udev` or similar is used to create device nodes we might have to extend it to scan e.g. `/sys/block`, which appears to list all block devices.

in essence making `/root` the new `/`[26]. Processes retain their current working directory even after a `chroot()` call however, so we also need to change that by using `chdir()`; otherwise the `init` process will have a current working directory above the root, leading to various weirdness.

62-71 Print out approximately how much time was spent waiting for the root to become available over the serial line (COM port) at 115200 baud. This is used in chapter 5.

73-74 Once inside the real root, replace ourself with the real `init` process, which is exactly the same as before.

We build `usbboot` with the following command:

```
$ i686-linux-gcc -Os -s -march=i686 -static -static-libgcc usbboot.c \
>                -o usbboot
```

As we are not inside the real root by the time `usbboot` runs, we can not dynamically link against the uClibc shared library stored there. To avoid having to duplicate the entire uClibc library, which would waste space, we link `usbboot` statically using the *-static* and *-static-libgcc* options to GCC.

Finally, we need to create an initramfs. These are stored in a archive format called **cpio**. The following commands will create an initramfs ready to be loaded by Grub and the kernel (as before, we use `Documentation/devices.txt` in the kernel tree as a reference for major and minor numbers for device nodes). `$ROOTFS` refers to the directory where we store the contents of the real root.

```
$ mkdir iramfs
$ cd iramfs
$ cp $BIN/usbboot .
$ mkdir root dev
$ mknod dev/console c 5 1
$ mknod dev/sdb1 b 8 17
$ find . -print0 | cpio --null -ov --format=newc | \
>      gzip -9 > $ROOTFS/boot/initramfs.cpio.gz
```

Finally we need to install the stage1, stage1.5, and stage2 Grub binaries, just as on PC. The command is similar, only substituting the device node corresponding to the USB device[27]:

```
$ grub-install --root-directory=rootfs /dev/sdb
```

This completes the installation of Awesom-O on a USB mass storage device.

3.11 Kernel and bootloader installation on BeagleBoard

The BeagleBoard expects the different bootloader stages, the kernel, and the root filesystem to appear at specific addresses on its built-in NAND Flash. The memory map is as follows:

[26] It also makes the '`..`' entry from `/root` point to the new root when inside the `chroot()`'ed filesystem.

[27] The caveat related to BIOS disk mappings mentioned in the footnote for Grub installation on PC (footnote 23) applies here as well, and could be more serious as Grub might have a harder time guessing BIOS mappings for USB devices.

```
0x00000000-0x00080000 X-Loader
0x00080000-0x00260000 U-Boot
0x00260000-0x00280000 U-Boot Env
0x00280000-0x00680000 Kernel
0x00680000-0x10000000 Filesystem
```

3.11.1 Building X-Loader

We first download and compile **X-Loader**, which handles the first stage (not counting the boot ROM) of the boot process. We will use the version available from `http://gitorious.org/beagleboard-validation/` (*retrieved Dec. 15, 2011*), which is the same version that comes pre-installed on the BeagleBoard, though possibly more recent. The following commands will build the X-Loader image:

```
$ git clone git://gitorious.org/beagleboard-validation/x-load.git
$ cd x-load
$ make CROSS_COMPILE=arm-linux- distclean
$ make CROSS_COMPILE=arm-linux- omap3530beagle_config
$ make CROSS_COMPILE=arm-linux-
```

The resulting image is called `x-load.bin`. If we are to run it from Flash (the alternative is an SD/MMC card), extra information (load address and size) needs to be appended. This is done via the **signGP** tool, found at `http://beagleboard.googlecode.com/files/signGP.c` (*retrieved Dec. 15, 2011*). We compile and run it as follows:

```
$ gcc signGP.c -o signGP
$ ./signGP x-load.bin
```

The resulting image, ready to be written to flash, is called `x-load.bin.ift`.

3.11.2 Building Das U-Boot

Next we download and build U-Boot. Like for X-Loader, we will use the version available from `http://gitorious.org/beagleboard-validation/` (*retrieved Dec. 15, 2011*). The build process is similar as well:

```
$ git clone git://gitorious.org/beagleboard-validation/u-boot.git
$ cd u-boot
$ make CROSS_COMPILE=arm-linux- distclean
$ make CROSS_COMPILE=arm-linux- omap3_beagle_config
$ make CROSS_COMPILE=arm-linux-
```

The resulting image is called `u-boot.bin`.

3.11.3 Writing the system to NAND Flash

We are now ready to write X-Loader, U-Boot, the Linux kernel, and the root filesystem to NAND Flash on the BeagleBoard. The memory map is given at the start of this section.

We will make use of a serial **RS232** link for communicating with U-Boot on the BeagleBoard, and will use a preexisting U-Boot installation living either on NAND Flash or on an SD/MMC card to install our system. The X-Load (`x-load.bin.ift`), U-Boot (`u-boot.bin`), kernel (`uImage`), and root filesystem (`disk.img`) images will come from a FAT32-formatted

SD/MMC card inserted into the BeagleBoard's SD/MMC port. The partition table on the SD/MMC card will have to adhere to a special format for U-Boot to be able to read it; see `http://code.google.com/p/beagleboard/wiki/LinuxBootDiskFormat` (*retrieved Dec. 15, 2011*). When following that guide, note that we only need to create a single partition for our purposes.

In the following it is assumed that we have booted into the U-Boot shell, communicating over RS232, and that our SD/MMC card is inserted into the BeagleBoard.

We run the following commands to write our X-Loader image to NAND Flash:

```
; Initialize U-Boot's MMC subsystem so that we can read from the SD card
# mmc init

; Load x-load.bin.ift from the SD card to the temporary memory location
; 0x84000000 (the 'nand write' command can only copy from memory)
# fatload mmc 0:1 84000000 x-load.bin.ift

; Use hardware-based ECC (the boot ROM expects this)
# nandecc hw

; Erase the X-Loader partition
# nand erase 0 80000

; Copy our new X-Loader image from the temporary memory location to NAND
; Flash
# nand write 84000000 0 80000
```

The following commands will write U-Boot (we will assume `mmc init` has already been run from here on):

```
; Load u-boot.bin to memory address 0x84000000
# fatload mmc 0:1 84000000 u-boot.bin

; Use software-based ECC (X-Loader expects this)
# nandecc sw

; Erase the U-Boot partition
# nand erase 80000 160000

; Write our new U-Boot image to NAND Flash
# nand write 84000000 80000 160000
```

The following commands write the kernel:

```
# fatload mmc 0:1 84000000 uImage
# nandecc sw
# nand erase 280000 400000
# nand write 84000000 280000 400000
```

The following commands write the root filesystem image:

```
# fatload mmc 0:1 84000000 disk.img
# nandecc sw
# nand erase 680000 F980000
# nand write 84000000 680000 6400000
```

3.11.4 Bootloader configuration (Das U-Boot)

We now remove the SD card and reboot our system. If everything goes right, this will land us at the shell of our freshly-installed U-Boot. Here we need to set two internal U-Boot variables that control the boot process:

- The *bootcmd* variable contains a sequence of commands that are automatically run by U-Boot upon boot unless a key is pressed within a user-specifiable timeout period.
- The *bootargs* variable holds the kernel command line to use.

We initialize these as follows, and save the settings to NAND Flash (into the `U-Boot Env` partition seen in the memory map) with the `saveenv` command.

```
# setenv bootcmd "nand read.i 0x80300000 0x280000 0x400000; \
>                 bootm 0x80300000"
# setenv bootargs "root=/dev/mtdblock4 rootfstype=ext2 \
>                  console=ttyO2,115200n8 console=tty0 \
>                  init=/bin/start \
>                  omapfb.mode=dvi:1024x768MR-24@60 \
>                  rw quiet"
# saveenv
```

The *bootcmd* commands will load the kernel from NAND Flash to memory address 0x80300000 and begin executing it using the *bootm* command. From the header of the `uImage` kernel image *bootm* will deduce that it is dealing with a Linux kernel and perform the proper initialization and loading procedure.

The kernel parameters used are the following:

root=/dev/mtdblock4 Specifies that our root filesystem lives on `/dev/mtdblock4`, which corresponds to the fifth (due to zero-indexing) NAND Flash partition (see the memory map at the beginning of the section).

rootfstype=ext2 Specifies the type of our root filesystem, saving the kernel the (mostly negligible) time to figure it out for itself[28].

console=ttyO2,115200n8 Specifies that system messages should be written to `ttyO2`[29], corresponding to our RS232 serial port, at 115200 Baud with no parity and with 8 data bits. This allows us to view system messages over our RS232 link, which is handy during development.

console=tty0 Specifies that system messages should additionally go to `tty0`, the foreground virtual terminal. We should preferably put this last among the *console* parameters, as the last assignment determines where messages go when user-space processes write directly to `/dev/console` (as opposed to messages coming from the kernel);
see `Documentation/serial-console.txt` in the kernel tree.

init=/bin/start As before, specifies that `/bin/start` should be used as the `init` process.

[28] ext2 is used for simplicity here, but is generally a poor choice on Flash-based devices as it does not do wear leveling, tending to produce defects at a faster rate than purpose-made filesystems such as ubifs and jffs2.

[29] On some older kernels "`ttyS2`" will need to be used instead.

omapfb.mode=dvi:1024x768MR-24@60 Specifies that the **omapfb** framebuffer video driver should be used, and sets the video mode to 1024x768x24 at 60Hz. For details on the format of this string, see `drivers/video/modedb.c` in the OMAP kernel tree.

rw As before, specifies that the root filesystem should be mounted read-write. This saves us a remount as this early version of Awesom-O does not perform consistency checks on disks.

quiet As before, suppresses the printing of system messages on the console during boot, which slows down the boot process.

This concludes the installation of Awesom-O on BeagleBoard. We now have a working Awesom-O running on all our target platforms.

Chapter 4

Boot time optimization through semi-automatic kernel minimization

4.1 Overview

Boot time profiling done at this point showed that initial time spent in the bootloader and kernel before the execution of the `init` process accounted for significantly more than 50% of the boot time on all platforms (see chapter 5 for the precise numbers and methodology), so focus was put on optimizing this part of the boot process.

This chapter describes a technique used to reduce the boot time of these stages on PC (due to lack of good BeagleBoard emulators as of writing (*Dec. 2011*), the technique would be cumbersome to use on that platform), based on automatic configuration, building, and testing of Linux kernels. At the core of the technique lies a program written in Python that parses the Linux Kconfig configuration language, going through configuration options one by one and turning them off. If Awesom-O is able to boot and fetch/render a page across the network when run in an emulator (QEMU[1]) with the option disabled, it is deemed *unessential*; otherwise it is *essential*, and will be re-enabled before the script considers other options.

4.2 Introduction to Kconfig: the Linux kernel's configuration language

Before compiling the kernel, the set of features and drivers to include can be customized in great detail, and hundreds of parameters (not to be confused with kernel parameters) that control the operation of the kernel can be set. The kernel configuration interfaces are invoked by running commands of the form `make *config` in the top-level kernel source directory, where the most commonly used interface is `make menuconfig`: a menu-driven configuration interface.

Internally, configuration options and their relations are described by files written in a language called **Kconfig**, documented in `Documentation/kbuild/kconfig-language.txt` in the kernel source tree. The `make *config` commands invoke C programs stored in the `scripts/kconfig/` directory that in turn process scripts written in the Kconfig language and present a configuration interface. The output of the configuration is a file `.config` containing

a list of assignments to variables. This file is read by the kernel makefiles[1] during the build process. The following is an example Kconfig file:

```
config FOO
    bool
    prompt "Support for foo"
    default y

menu "A menu"

config BAR
    bool
    prompt "Support for bar"
    default FOO

config BAZ
    string
    prompt "Value for baz"
    depends on BAR
    default "String value"

endmenu
```

This file declares *FOO*, *BAR* and *BAZ* to be three configuration symbols. *FOO* and *BAR* have type *bool*, meaning they take on boolean values with true and false represented by 'y' and 'n', respectively. *BAZ* has type *string* and takes on a text string as its value. All three configuration symbols have prompts that will identify them in e.g. the `make menuconfig` interface, and also have default values that will be used if the user does not explicitly specify a different value. The default value of *BAR* is set to be the value of *FOO*.

The *depends on* clause in the definition of *BAZ* is an example of a dependency between symbols. Unless *BAR* has the value 'y', *BAZ* will not have a value[2] and will not be displayed in the `make menuconfig` interface. The last two configuration variables appear within their own submenu, titled "A menu".

The following `.config` will be generated from the above Kconfig if the default value is used for all symbols:

```
CONFIG_FOO=y
CONFIG_BAR=y
CONFIG_BAZ="String value"
```

4.3 Kconfiglib: a Python Kconfig interpreter

In order to automatically generate and test kernel configurations, a way is needed to process Kconfig files. I initially considered modifying the Kconfig C sources, but after a bit of experimentation it appeared simpler to instead implement my own interpreter in Python. After having implemented the parser I went off on a long tangent that eventually turned out a general-purpose library for exploring, debugging, and scripting Kconfig-based configuration systems called Kconfiglib; see [14] for the message containing the initial release of this library. The kernel minimizer is implemented as a Python script that makes use of Kconfiglib.

[1] This is done simply by including it, as `.config` obeys Make syntax.

[2] Or rather, it will have the empty string as its value, but will not be written to the `.config`.

4.4 The kernel minimization script

A pseudo-code version of the kernel minimization script appears in the following figure, with comments indicated by a # at the beginning of a line. The process starts from a known working kernel configuration. The complete source code for the script appears in Appendix A.

```
Parse Kconfig configuration for X86

# req is the set of symbols determined to be essential
Set req to the empty set
Loop
  If there is a symbol S that is not in req and that can be disabled
    Disable S
  Else
    Done - exit the script

  Write a new .config
  Compile the kernel

  Set up a listening TCP socket on port 1234

  # Awesom-O has been configured so that /bin/opera attempts to
  # load a page by connecting to port 1234 on the development
  # machine
  Launch Awesom-O in an emulator (QEMU)

  If we do not get a connection attempt to port 1234 within 10 seconds
    # /bin/opera appears to have failed to initialize itself and
    # connect
    Enable S
    Add S to req

  Kill the QEMU process
```

In addition to the above, the actual script will not disable a symbol if doing so is found to increase the size of the kernel; will record how many bytes are saved by turning off each unessential feature; supports manually marking features as essential (why this is useful is explained later); and supports stopping and resuming the minimization process, which is handy as building and testing a large number of kernels is time consuming.

Under the hood, Kconfiglib keeps track of dependencies and invalidates/reevaluates symbols as needed, just as if we had carried out the above process manually in the `make menuconfig` interface.

4.5 Testing the kernel

To use the above method we need a way to automatically test if a kernel configuration is functional. "Functional" here means that the system should be able to boot in an emulator, run the `/bin/opera` executable, and have it fetch a page over the network; it should also support basic functionality such as graphics and mouse and keyboard input. To automate this process, a version of Awesom-O was produced that automatically attempts to load a page from an address on the local network from within `/bin/opera`. This was done by modifying

`/etc/initscript`, replacing the line

```
/bin/opera -u file:///resources/start-page.html
```

with

```
/bin/opera -u http://192.168.0.2:1234
```

where 192.168.0.2 is the IP address of the development machine. The kernel minimizer script listens for connection attempts on port 1234 and deems the boot successful if one is detected within the timeout period. To make sure graphics and keyboard/mouse input is working, `/bin/opera` was modified to abort before attempting to load the page in case of problems with graphics or input device initialization.

QEMU was chosen as the emulator mainly because it supports selecting a kernel to use via the *-kernel* command line option (that is, it provides its own bootloader), avoiding the need to install a new kernel in the virtual machine's filesystem for each new configuration.

4.6 Caveats

Though the kernel minimization process will arrive at a kernel that is in some sense as minimal as we can get without modifying kernel source code, it is almost certain to turn off features that we would want to keep enabled for performance, stability, convenience, or other reasons. For example, swap support is almost certain to get disabled, as well as various features that work around CPU bugs. Hence, some amount of manual tweaking will always be required, which is why this method is only *semi*-automatic. Having a list of truly essential features and knowing approximately how much space can be saved by turning off various unessential features is still likely to save many days of tedious manual testing.

It turned out to be handy to be able to manually mark a feature as essential and restart the minimization process from some arbitrary point. To facilitate this workflow, the minimization script writes out a list of essential and unessential features in the `.config` files it generates, and will respect any such list in a `.config` file it reads in (when resuming from a configuration). Marking a feature as essential is simply a matter of manually adding it to the list of essential features.

A list of how many bytes were saved by turning off various features, produced with the help of the kernel minimization script, appears in Appendix B.

Chapter 5

Measurement techniques and results

The following techniques were used to measure the disk footprint and boot time of the finished system.

5.1 Measuring disk footprint

Disk footprint was measured by running

```
$ du -b
```

in the directories corresponding to the different Awesom-O root filesystems on the development machine. The *-b* flag instructs `du` to sum up the size in bytes of the contents of all files, giving a measure independent of the filesystem used.

The actual amount of space occupied by files on disk might be more or less than the size of their contents, depending on characteristics of the filesystem used. For example, few filesystems will put more than one file into a sector, so that any file—no matter how small—will use up at least 512 bytes[1]. Often filesystems will store files in units that are some multiple of the sector size—called the filesystem's **block size**—to reduce the amount of metadata that needs to be stored, wasting even more space for small files.

On the other hand, a file might occupy *less* space on disk than the size of its content. This might happen for example with a **compressed filesystem**, which will dynamically compress and decompress files as they are written to and read from disk. It might also happen for filesystems that support **sparse files**, allowing long stretches of 0's to be indicated by flags rather than being stored explicitly on disk (data stored this way is often referred to as a **hole** in the file).

For the Awesom-O filesystems the ext2 block size was 1 KiB[2]. The actual size on disk was about 0.6% larger than the `du -b` measurement.

5.2 Measuring boot time

We split the boot process up into three different stages:

[1] 4 KiB on disks that have switched to 4 KiB sectors. Additionally, an inode and perhaps other metadata needs to be stored for each file.

[2] The `mke2fs` utility uses settings from `/etc/mke2fs.cong` to determine what block size to use, usually defaulting to 1 KiB for disks at most 512 MiB in size and 4 KiB for larger disks.

Bootloader The time spent before kernel code begins executing. This includes initialization done in the bootloader and the loading of the kernel and an eventual initramfs. On PC, the initial time spent in the BIOS boot code is omitted, as it varies wildly between system and is not affected by Awesom-O (BIOS modifications are outside the scope of this thesis, but see section 6.1.6 — *Custom BIOSes*).

Kernel The time spent in the kernel before the init process `/bin/start` is executed.

User space The time spent running `/bin/start`, the initialization script, and `/bin/opera`, up until the point where we have a completely initialized web browser that responds to user input[3].

For booting from USB on PC, we additionally record the time `usbboot` spends waiting for the USB mass storage device to become available (labelled "**USB wait**" in the graphs below).

To measure the time spent in the different stages a null modem cable[4] was connected between the system running Awesom-O and a development machine. The Awesom-O system was configured to send text strings over the serial line at key points, which would then be read and timestamped automatically. The following command was used on the development machine:

```
$ picocom -b 115200 -d 8 /dev/ttyS0 | ptx_ts
```

This sets up `picocom` to listen on the serial port (`/dev/ttyS0`) at 115200 Baud with 8 data bits and send any incoming data to the `ptx_ts`[7] utility: a small Perl script that adds a timestamp to each line of text it receives.

On the Awesom-O end, the `/bin/start`, `/bin/opera` and `usbboot` programs were modified to print text strings over the serial port (just after starting for `/bin/start`; just before idling waiting for user input for `/bin/opera`; and after having mounted the filesystem for `usbboot`— see the source code for `usbboot` in section 3.10.2 — *Booting from USB*). The maximum reliably supported baud rate (115200) was used throughout to minimize the impact of sending data over the serial line on the boot time, and as before *quiet* was passed on the kernel command line to suppress boot messages. To differentiate time spent in the bootloader from time spent executing kernel code, the `/proc/uptime` file, containing the time since the kernel began executing on PC, was output over the serial line.

Five boots were timed for each configuration (with e.g. 'Celsius + USB + minimized kernel' being a single configuration) and the average value was used, rounded to the nearest tenth of a second. Except for the bootloader stage (see below) all measurements for the same configuration stayed within 60 milliseconds of each other across boots (with an average deviance of about 20 milliseconds from the mean), where most of the variation is likely due to non-determinism having to do with disk drives and other hardware. Even assuming a completely deterministic boot, errors are unlikely to surpass ± 0.1 s, and are not cumulative across different stages.

A small wart in the above method turned out to be Grub Legacy on PC, which although capable of sending messages over the serial lines does not display a message before loading

[3]Due to having to wait for network interfaces to be initialized via DHCP, the network might become available later (usually within a few seconds).

[4]A null modem cable is a serial cable connected directly between two systems with the send and receive lines crossed. The name originates from the days of serial modems, where "null" designates the absence of a modem in the case of a direct connection.

the kernel. The best solution would be to either modify Grub to display a message or switch to Grub2, which supposedly has more complete debugging output facilities. A semi-manual approach was used instead that involved pressing a key at the same time on the development and Awesom-O machines. Five measurements were all within 70 milliseconds of each other (with an average deviance of about 25 milliseconds from the mean), so the error introduced is likely to be small, and only affects the measurement of the time spent in the bootloader. BeagleBoard does not have the same problem as U-Boot is verbose in its output.

5.3 Disk footprint on PC

On PC, the total size of the root filesystem is 56.14 MiB, which breaks down as follows in order of decreasing size:

- 45.5 MiB for various resources needed by the Devices SDK, with the majority of space (44.5 MiB) taken up by font files.
- 8.75 MiB for `/bin/opera`, which links statically against DirectFB.
- 1.27 MiB for the minimized 2.6.37.1 kernel (3.94 MiB for the `make defconfig`-based kernel). This number varies slightly depending on what network interface driver is used.
- 334 KiB for the `/lib` dynamic libraries, with just 234 KiB taken up by uClibc.
- 186 KiB for the customized `/bin/busybox` executable.
- 10.6 KiB for the `/boot/initramfs.cpio.gz` initramfs (18.2 KiB for the uncompressed `usbboot` executable).
- 4.11 KiB and 3.63 KiB for the `/bin/start` and `/bin/shutdown` executables, respectively.

Excluding `/bin/opera` and related resources, the size of the root filesystem is just 1.83 MiB.

5.4 Disk footprint on BeagleBoard

On the BeagleBoard the bootloader and kernel are stored in their own partitions on the NAND Flash and so do not add to the size of the root filesystem. The size of the 2.6.32-based OpenEmbedded BeagleBoard kernel is 2.26 MiB with the Angstrom configuration.

The total size of the root filesystem on BeagleBoard is 56.1 MiB, which breaks down as follows in order of decreasing size:

- 45.5 MiB for various resources needed by the Devices SDK, with the majority of space (44.5 MiB) taken up by font files.
- 9.74 MiB for `/bin/opera`, which links statically against DirectFB.
- 385 KiB for the `/lib` dynamic libraries, with 264 KiB taken up by uClibc.
- 230 KiB for the customized `/bin/busybox` executable.

- 4.24 KiB and 3.69 KiB for the `/bin/start` and `/bin/shutdown` executables, respectively.

Excluding `/bin/opera` and related resources, the size of the root filesystem on BeagleBoard is just 661 KiB.

As expected due to the RISC nature and default fixed-width instruction set of the BeagleBoard's ARM CPU, the binaries are larger than on x86. As mentioned in section 1.7.3 — *Thumb-2 instructions*, the sizes of the executables could likely be reduced significantly by using the Thumb-2 instruction set.

5.5 Systems used to test boot time

For PC, measurements were done on the following two systems:

1. An Acer Travelmate 8200 laptop with the following system specifications:
   - Intel Core Duo T2500 CPU at 2.0 GHz
   - Intel 945PM Express chipset
   - 2 GB DDR2 SDRAM at 533.0 MHz
   - Momentus 5400.2 SATA 1.5Gb/s 120 GB (ST9120821AS) hard drive
2. A Celsius M460 workstation with the following system specifications:
   - Intel Core 2 Quad Q9300 CPU at 2.5 GHz
   - Intel 946GZ chipset
   - 3 GB DDR2 SDRAM at 677.0 MHz
   - Barracuda 7200.11 SATA 3.0Gb/s 500 GB (ST3500320AS) hard drive

The Acer Travelmate 8200 was picked because it roughly corresponds performance-wise to what one could expect to find in a 2011 vintage ~300 US dollars netbook[5] (compare with e.g. the Samsung RV510) and also because it happened to support a COM port via a PCI Express extension module[6].

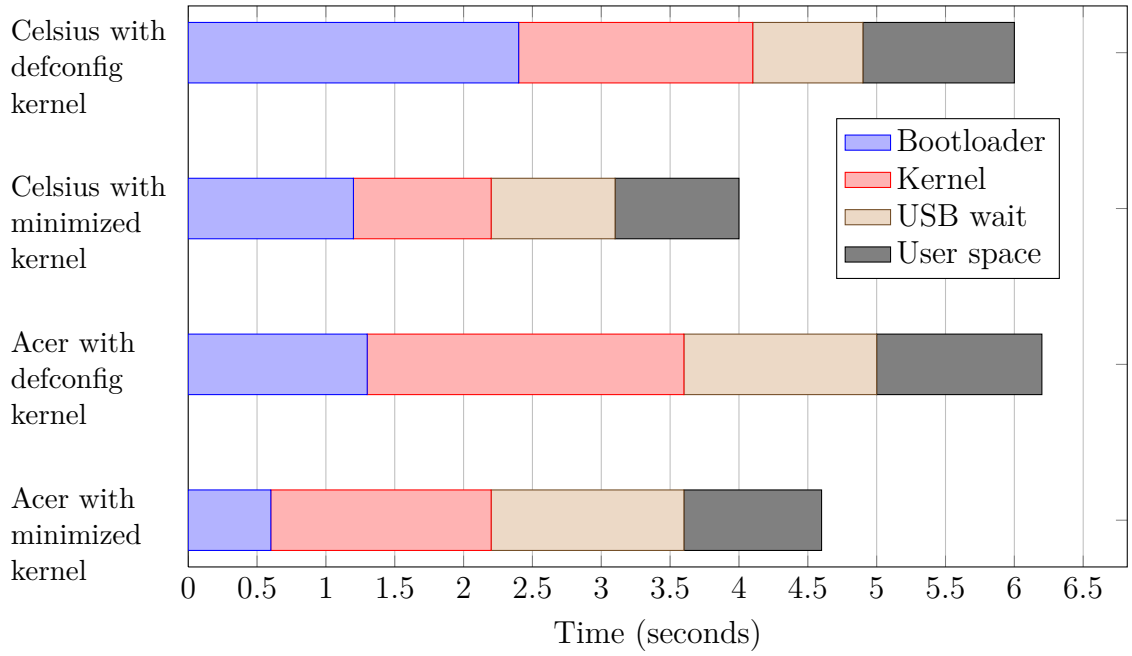
The Celsius M460 was picked to have a beefier system as a reference point, in order to get some idea of what the variability of the results might be, and also to see how the system could perform on future netbook hardware.

5.6 Boot time on PC from a USB device

The following graph shows the average boot times across the five measurements with the `make defconfig` and minimized kernels when booting from a USB 2.0 stick (ChipsBank CBM2080):

[5] For its time (2006) it was a very high-end laptop.

[6] USB-to-COM adapters usually require specialized drivers that are not available during early boot.

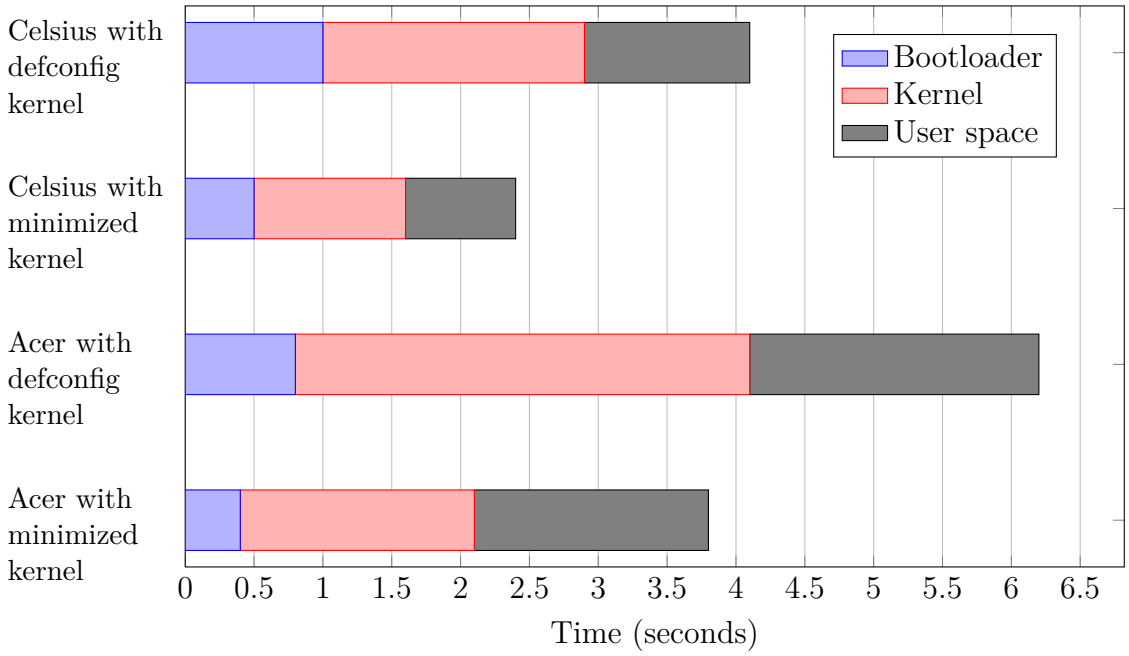


Compared to the `make defconfig` kernel, the boot time from USB on the Celsius and Acer systems with the minimized kernel is 33% and 26% shorter, respectively. The two systems spend between 0.8 and 1.4 seconds in `usbboot` waiting for the USB device to become available, with the duration being about the same for the `make defconfig` and minimized kernels. This suggests that it might be possible to further improve the boot time by attempting to move USB initialization to an earlier point in the kernel initialization process.

Interestingly, the bootloader stage of the more powerful Celsius system is considerably slower than on the Acer, likely due to BIOS idiosyncrasies related to booting from USB devices (considering the bootloader stages are much more closely tied when booting from disk).

5.7 Boot time on PC from disk

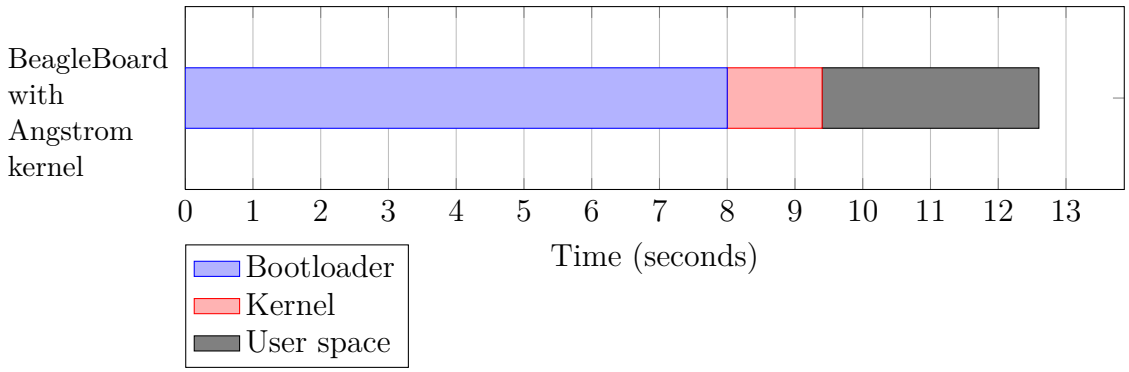
The following graph shows the average boot time across five measurements on PC when Awesom-O was installed on disk:



Here, the minimized kernel lowers the boot time by 41% and 39% on the Celsius and Acer systems, respectively. Interestingly, the time spent in user space is considerably (50% with the minimized kernel) longer on the Acer machine when booting from disk compared to USB. This might be due to characteristics of the Acer's 5400 RPM hard drive, and it might be possible to lower the boot time by packing essential files in a way that avoids redundant seeks, which for a hard disk drive involves physically moving a read/write head across the drive's surface (Flash-based hard drives such as SSD drives generally have much better seek performance).

5.8 Boot time on BeagleBoard

The following graph shows the average boot time across five measurements on BeagleBoard:



As the above graph makes clear, an obvious future focus area for improving the boot time on the BeagleBoard would be U-Boot. This would involve further breaking down the bootloader stage (made easy by U-Boot's relatively low-level mode of operation and verbose output), identifying where time is spent, and then drilling down into the source code to figure out precisely what is being done and how it can be made more efficient.

An educated guess from looking at projects such as **SwiftBeagle**[5], achieving a three-second power-on-to-shell-prompt boot time, is that most of the slowness here can be attributed to loading the kernel via polled I/O rather than DMA. DMA transfers often require highly device-specific code, which might explain why it has not been implemented for U-Boot on the BeagleBoard.

Chapter 6

Conclusions and future works

As shown in the preceding chapter, WebOS boot times of less than four seconds (excluding BIOS) are feasible on commodity netbook-class PC hardware without modifying kernel source code. The results also demonstrate the effectiveness of the kernel minimization technique devised in this thesis (see chapter 4).

Of note is the big decrease in time spent in the kernel following minimization, which shows that the primary benefit of kernel minimization on many systems could be less time spent in the kernel, as opposed to a smaller kernel for the bootloader to load.

Compared to the initial goals stated in section 1.3 — *Goals* (boot time under 10 seconds, disk usage under 50 MiB), the system has clearly reached the goal for boot time on PC. No initial boot time goal was stated for BeagleBoard, but with bootloader optimization it seems very likely we could see similar figures for that system. With respect to disk usage, the system falls just short of the goal at about 56 MiB, but we had likely neglected to consider the size of Devices SDK resources—in particular fonts—when coming up with that goal. The size of the root filesystem excluding web browser resources is just 1.86 MiB on PC.

Even though no great effort could be put into optimizing the bootloader stage of the boot process on the BeagleBoard due to limited time, projects such as SwiftBeagle[5], achieving a three-second power-on-to-shell-prompt boot time, suggest that boot times measured in a few seconds could be achieved on that platform as well[1].

An interesting data point to have would have been booting from disk on a PC with a solid-state hard drive (SSD), which typically have much better performance than traditional magnetic disk hard drives, in particular having superior random-access (seek) performance. As much of the speed of the boot process is disk-bound, we would likely see a significantly faster boot on such a system, assuming otherwise equal hardware.

Awesom-O probably owes much of its short boot time to the fact that no graphical windowing system such as X11 is used, which would increase initialization time and the amount of data that needs to be loaded from disk during boot. One advantage of using X11 would be that driver support for hardware acceleration (particularly for 3D graphics) is broader and more mature than for DirectFB. Optimizing X11 startup time could very well be a fairly large project in its own right.

[1]This becomes especially feasible considering that the pre-optimized boot time of SwiftBeagle is listed as about 15 seconds, whereas Awesom-O boots in 12.6 seconds on the BeagleBoard, and in about 9.4 seconds to a shell.

6.1 Future investigations and improvements

The following sections lists techniques and ideas for lowering the boot time of Awesom-O further, but that I either never got the time to investigate or that fall outside the scope of this thesis.

6.1.1 BeagleBoard bootloader optimization

As already mentioned, looking into U-Boot optimization would be an obvious next step on BeagleBoard. Writing a custom bootloader from scratch would be an option if an optimized U-Boot still gives unsatisfactory results.

6.1.2 Extending the kernel minimization script

The kernel minimization script could be extended to record the actual boot time rather than just the size of the kernel, as it might be the case that disabling certain features could increase the total boot time even though the size of the kernel decreases. Unfortunately this might mean having to abandon the emulation approach, as the boot time in an emulator is not necessarily representative of the boot time on a real system (for example, Awesom-O boots in about a second when run in the VirtualBox virtualization software on the Celsius system).

6.1.3 Alternative filesystems

Not counting in-memory and pseudo filesystems, the only filesystem currently used in Awesom-O is ext2; other filesystems should be investigated well. For example, putting read-only resources and executables in a **cramfs** (a read-only, compressed filesystem) could speed up the boot process provided the decompression overhead is lower than the time saved by reading less data from disk. On the BeagleBoard we should use a filesystem tailored to Flash, such as ubifs or jffs2 (cramfs would obviously be fine as well, as only writes are problematic).

6.1.4 Detailed kernel boot-up profiling

Having a detailed profile of the Linux kernel's boot process would likely be very helpful in identifying profitable avenues for optimization when diving into kernel source code. This would involve investigating existing instrumentation facilities in the Linux kernel and if need be adding more. Especially useful would be a profile that included parallel charts for Linux kernel threads[2], as this might help us rearrange the initialization process to make certain tasks finish earlier (USB device initialization comes to mind). In the simplest case, `dmesg` output could be used for profiling, especially if the kernel is configured to produce lots of debugging output.

For user space boot profiling there is Bootchart[15], which unfortunately is not of much help due to Awesom-O's exceedingly simple user space.

[2]Linux kernel threads are threads that run inside the kernel, generally invisible from user space. In the output of the `ps -aux` and `top` commands, kernel threads are enclosed in square brackets ([]).

6.1.5 Kernel system tailoring

The Linux kernel is designed to run on a wide range of hardware and so often has to assume the worst when it comes to timing characteristics of hardware peripherals. It also has to perform some amount of probing during boot to determine what hardware is available. If Awesom-O was to be run only on systems with a known hardware configuration, we could improve boot time by adjusting wait times in accordance with known hardware specifications, and could omit most probing altogether.

6.1.6 Custom BIOSes

If we know that a particular PC system is to run Awesom-O, we could modify its BIOS (or other bootloader firmware) to only do the minimum amount of hardware initialization required by Awesom-O and let the kernel handle the rest[3]. As mentioned in section 2.3.3 — *Hardware initialization (PC and BeagleBoard)*, PC BIOSes will typically initialize hardware that is later reinitialized by the Linux kernel, wasting time. This technique is used to reduce boot time on some netbooks running Google's **Chrome OS**[11].

6.1.7 Late-loading of drivers and subsystems

To further reduce the size of the kernel beyond what is possible with the minimization technique described in chapter 4, certain "essential" drivers and subsystems could be loaded from modules stored in the root filesystem after (or parallel to) launching the user interface. For example, this could be done for network support, which would likely save a significant amount of space. In order to use this approach parts of the boot process would have to be reengineered, as e.g. `udhcpc` is not happy about being launched before network interfaces are available.

This technique would only reduce the time to first user interaction, and might actually increase the time until the network becomes available.

6.1.8 Loading the user interface in stages

An interesting technique for reducing the time to first user interaction as well as subsequent load times is to load the user interface in stages, only loading precisely what is needed for the current kind of user interaction at each stage. For example, if the system has a login screen, we could load just the resources it needs before launching it, and then keep loading the rest of the system in the background while the user types his username and password. This trades system complexity for shorter load times.

6.1.9 Resume-from-hibernation as booting

Given adequate hardware support (in particular, being able to extract and quickly restore hardware state), the theoretically fastest way to boot a system would be to simply load the device state (memory contents, status of hardware registers and buffers, etc.) of a booted machine directly. This is reminiscent of waking from sleep on operating systems that support

[3]Such a BIOS could of course still offer a compatibility mode as an option for running other operating systems.

hibernation—the saving and later restoring of machine state on disk—though even those systems need to perform a certain amount of dynamic reinitialization.

An interesting hybrid approach would be to combine this technique with loading the user interface in stages. The device state of the system at the login screen (which would likely be faster to load than that of a fully initialized system) could be recorded and restored at boot, and the rest of the system loaded while the user types his username and password.

Downsides to this approach might be increased complexity and brittleness when upgrading software and hardware on the system. Parts of the boot process would likely always have to be "dynamic".

6.2 Concluding remarks

Though this thesis has the word 'Web' in its title, the methodology and techniques discussed herein are fairly generic, and could be applied to pretty much any Linux system that needs to have a very short boot time and a small disk footprint—not just to netbooks. One area where these are very common requirements is in small and embedded systems, where users have come to expect a very fast boot and where built-in storage is often limited. With the increasingly advanced state of hardware, Linux and other Unix-based systems—which have traditionally been considered too "heavy" for small platforms—have lately made great inroads into this sector, with **smartphones** and platforms such as **Android** and **iOS** being of particular note.

By far the most time-consuming part of the thesis was the development of Kconfiglib, which turned into its own side project and was eventually released as an open-source general-purpose library for exploring and scripting Kconfig-based configuration systems (see [14]). In retrospect I had greatly underestimated the complexity of the Kconfig language, and modifying the existing C implementation would likely have been a much quicker way to implement the kernel minimizer. On the upside, modifying the C implementation is unlikely to have yielded a general-purpose tool.

Kconfiglib was used by Ingo Feinerer et al. in the paper *Efficient configuration and verification of software product lines*[8] to extract and analyze a dependency graph of the Linux kernel's configuration options, and will also be used in an upcoming paper by the same author. Kconfiglib is also being looked into by the Freetz[18] project, which produces an open-source firmware extension for the AVM Fritz!Box[10] line of routers.

Appendix A

The complete kernel minimization script

The complete source code of the kernel minimizer described in chapter 4 appears below. It makes use of Kconfiglib, the latest version of which can be found at `https://lkml.org/lkml/2011/2/6/151` (*retrieved Dec. 15, 2011*).

```
import kconfiglib
import os
import re
import shlex
import socket
import subprocess
import sys

# Timeout in seconds before a boot is considered unsuccessful
timeout = 10

# Kernel command line
kernel_cmdline = \
"root=/dev/sda1 rootfstype=ext2 init=/bin/start " \
"video=vesa:ywrap,mtrr vga=0x344 console=/dev/ttyS0 console=tty0 rw"

# Command to start QEMU with
qemu_cmd = """
qemu -hda ../pcdisk.img
     -vga std
     -serial stdio
     -net user -net nic,model=e1000
     -m 1024M
     -usbdevice mouse
     -kernel arch/x86/boot/bzImage
     -append '{0}'
""".format(kernel_cmdline)

# Command to compile the kernel with
compile_cmd = "make CROSS_COMPILE=i686-linux- -j8"

# Headers of sections listing essential/unessential symbols in the
# .config's
req_header     = "Required:"
```

```
not_req_header = "Not required:"

# Names of symbols we have determined must be enabled in order for the
# Devices SDK to be functional
req = set()

# Names of symbols we have determined need not be enabled, mapped to the
# size in bytes of the kernel image saved by disabling them
not_req = {}

#
# Helper functions
#

def run_cmd(cmd, cwd = None):
    """Runs the command 'cmd', split up as in a Bourne shell, and returns
    its exit code. Correctly handles commands that generate a lot of
    output."""
    try:
        process = subprocess.Popen(shlex.split(cmd),
                                   cwd = cwd,
                                   stdout = subprocess.PIPE,
                                   stderr = subprocess.PIPE)
        process.communicate()
    except OSError, e:
        sys.stderr.write("Failed to execute '{0}': {1}".format(cmd, e))
        sys.exit(1)

    return process.returncode

def get_process(cmd):
    """Like run_cmd(), but return the process object instead of waiting for
    the process to finish."""
    try:
        process = subprocess.Popen(shlex.split(cmd),
                                   stdout = subprocess.PIPE,
                                   stderr = subprocess.PIPE)
        return process
    except OSError, e:
        sys.stderr.write("Failed to execute '{0}': {1}".format(cmd, e))
        sys.exit(1)

def write_config(conf, filename):
    not_req_items = sorted(["{0} ({1})".format(sym, size) for
                            (sym, size) in not_req.iteritems()])

    conf.write_config(filename,
                      "\n".join([req_header] +
                                sorted(req) +
                                [not_req_header] +
                                not_req_items))

# Main script logic

# Set up environment variables used by Kconfig files
```

```
os.environ["ARCH"] = "i386"
os.environ["SRCARCH"] = "x86"
os.environ["KERNELVERSION"] = "2"

print "Parsing Kconfig"
conf = kconfiglib.Config("Kconfig", base_dir = ".")

# Check if we should resume from a configuration in configs/

if not os.path.exists("configs"):
    os.mkdir("configs")

# Find .config with maximum index in configs/ (presumably the most recent
# one)
conf_index = -1
for c in os.listdir("configs"):
    match = re.match(r"^config_(\d+)$", c)
    if match:
        index = int(match.group(1))
        conf_index = max(conf_index, index)

if conf_index == -1:
    print "Starting from known good configuration"
    run_cmd("cp .goodconfig .config")
else:
    print "Resuming from configs/config_{0}".format(conf_index)
    run_cmd("cp configs/config_{0} .config".format(conf_index))

conf.load_config(".config")

print "Parsing .config header"

DISCARD, ADD_REQ, ADD_NOT_REQ = 0, 1, 2
state = DISCARD

lines = conf.get_config_header().splitlines()
for line in lines:
    line = line.strip()

    if line == "":
        continue

    if line in (req_header, not_req_header):
        state = ADD_REQ if line == req_header else ADD_NOT_REQ
        continue

    if state == DISCARD:
        continue

    sym_name = line.split()[0]

    (req if state == ADD_REQ else not_req).add(line)

# Compile initial kernel to get reference for size (it is assumed to boot
# successfully)
```

```
print "Compiling initial kernel (assumed to be good) to get reference" \
      "for size"

run_cmd(compile_cmd)

# Size of the previously built kernel in bytes
old_kernel_size = os.path.getsize("arch/x86/boot/bzImage")

# Main loop - disable symbols one by one and test the resulting kernels

while True:
    print "Disabling modules"
    # Run in a loop in case disabling one module enables other modules
    # which can then be disabled in turn (unlikely, but just to be safe)
    while True:
        for sym in conf:
            if not sym.is_choice_item() and \
               sym.calc_value() == "m"  and \
               sym.get_lower_bound() == "n":
                sym.set_value("n")
        else:
            break

    # Search for symbol to disable
    for sym in conf:
        if not sym.is_choice_item() and \
           not sym.get_name() in req and \
           sym.get_type() in (kconfiglib.BOOL, kconfiglib.TRISTATE) and \
           sym.calc_value() == "y":

            val = sym.calc_value()
            # Get the lowerst value the symbol can be assigned, with "n",
            # "m", and "y" being arranged from lowest to highest
            lower_bound = sym.get_lower_bound()

            if lower_bound is not None and \
               kconfiglib.tri_less(lower_bound, val):
                print "Lowering the value of {0} from '{1}' to '{2}'" \
                      .format(sym.get_name(), val, lower_bound)
                sym.set_value(lower_bound)
                break
    else:
        print "Done - no more symbols can be disabled"
        break

    # Test the kernel

    write_config(conf, ".config")

    print "Compiling kernel"
    run_cmd(compile_cmd)

    # Compare size of kernel to size of previous kernel to make sure it has
    # decreased
```

```
kernel_size = os.path.getsize("arch/x86/boot/bzImage")
if kernel_size >= old_kernel_size:
    print "Disabling {0} did not decrease the size of the kernel. " \
          "Re-enabling it.".format(sym.get_name())
    sym.set_value("y")
    req.add(sym.get_name())
    continue

listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
# This avoids some situations in which we fail to reuse the port after
# restarting the script because the connection is in the TIME_WAIT
# state.
listen_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
listen_sock.settimeout(timeout)

# The empty string in the host portion is treated like INADDR_ANY
listen_sock.bind(("", 1234))
listen_sock.listen(10)

print "Running system in QEMU"
qemu_process = get_process(qemu_cmd)

print "Waiting for the system to contact us"

try:
    sock, addr = listen_sock.accept()
    boot_successful = True
    sock.close()
except socket.timeout, e:
    boot_successful = False

if boot_successful:
    # Write a new configuration
    print "Boot successful! Disabling {0} saved {1} bytes.".\
          format(sym.get_name(), old_kernel_size - kernel_size)
    conf_index += 1
    not_req[sym.get_name()] = old_kernel_size - kernel_size
    write_config(conf, "configs/config_{0}".format(conf_index))
    old_kernel_size = kernel_size
else:
    print "Boot failed! Turning {0} back on".format(sym.get_name())
    sym.set_value("y")
    req.add(sym.get_name())

print "Terminating QEMU"
qemu_process.terminate()

listen_sock.close()
```

Appendix B

Sizes for unessential kernel features

The following table lists the number of bytes saved by turning off various "unessential" (see chapter 4) kernel features in the 2.6.37.1 kernel, as recorded by the kernel minimization script (some of these were later manually marked as essential). One caveat to the sizes recorded here is that some could depend on what other features were enabled at the point when a particular feature was disabled (e.g., the one for *MODULES*). The following could be handy information when reducing the size of a kernel by hand.

Feature	Size (bytes)
SOUND	218688
PM	211840
RELOCATABLE	205536
SMP	194656
NETWORK_FILESYSTEMS	192608
AGP	143168
IPV6	138464
CFG80211	120192
FIRMWARE_IN_KERNEL	113088
MODULES	101920
AUDIT	86464
FTRACE	85984
MD	68032
NETFILTER	60896
DRM	47744
TIGON3	46976
HID_SUPPORT	46368
CRYPTO	39264
NET_ETHERNET	37856
PCCARD	28192
BNX2	26176
EXPERIMENTAL	23104
CGROUPS	22816
VFAT_FS	20960
FRAME_POINTER	20832
BLK_DEV_SR	17568

USB_DEBUG	17408
KPROBES	16352
SKY2	15424
QUOTA	13600
NETLABEL	13376
R8169	12512
NET_SCHED	12384
KEYS	12288
HUGETLBFS	12192
ISO9660_FS	12096
X86_MCE	11936
SYSVIPC	11680
SCSI_CONSTANTS	11040
IP_MULTICAST	10624
BLK_DEV	9696
USB_OHCI_HCD	8800
CHR_DEV_SG	8288
UNIX	8128
LBDAF	7872
CPU_FREQ	7776
AUTOFS4_FS	7744
PCIEPORTBUS	7456
XFRM_USER	7392
RTC_CLASS	6880
USB_MON	6656
INOTIFY_USER	6592
TASKSTATS	6144
SECURITY	6144
IP_PNP	5984
PHYLIB	5600
NO_HZ	5600
KEXEC	5568
HIGH_RES_TIMERS	5504
SCSI_SPI_ATTRS	5472
EFI	5472
PROFILING	5408
IP_ADVANCED_ROUTER	5344
I2C	5216
PARTITION_ADVANCED	4576
NETCONSOLE	4544
MICROCODE	4544
DEBUG_KERNEL	4448
TR	4416
SATA_PMP	4416
NET_NS	4352
FB_MODE_HELPERS	4320

RFKILL	4256
HPET_TIMER	4128
LOGO	4064
HW_RANDOM	4000
CPU_IDLE	3680
BLK_DEV_BSG	3680
CRASH_DUMP	3552
THERMAL	3296
DEBUG_FS	3168
MSDOS_FS	2944
SYN_COOKIES	2912
NEW_LEDS	2880
USB_DEVICEFS	2624
HOTPLUG_PCI	2464
USB_PRINTER	2432
RELAY	2432
MAGIC_SYSRQ	2400
SCSI_PROC_FS	2304
POWER_SUPPLY	2176
I2C_I801	2144
EARLY_PRINTK_DBGP	1984
SERIAL_8250_CONSOLE	1824
INPUT_FF_MEMLESS	1824
INET_LRO	1824
PROC_KCORE	1792
CONNECTOR	1664
BINFMT_MISC	1632
X86_CHECK_BIOS_CORRUPTION	1600
DNS_RESOLVER	1568
BACKLIGHT_LCD_SUPPORT	1568
MII	1536
ATA_VERBOSE_ERROR	1536
TCP_CONG_CUBIC	1504
PID_NS	1408
PATA_AMD	1248
NETWORK_SECMARK	1152
MACINTOSH_DRIVERS	992
DNOTIFY	960
DMIID	896
TMPFS_POSIX_ACL	832
SERIO_SERPORT	768
NLS_CODEPAGE_437	768
VGACON_SOFT_SCROLLBACK	736
DEVKMEM	736
SYSCTL_SYSCALL_CHECK	704
PROVIDE_OHCI1394_DMA_INIT	704

NVRAM	672
INPUT_SPARSEKMAP	640
PATA_OLDPIIX	576
USB_LIBUSUAL	544
CRC_T10DIF	544
HWMON	512
INPUT_POLLDEV	448
HIGHPTE	448
FB_TILEBLITTING	448
VT_HW_CONSOLE_BINDING	320
PATA_SCH	256
PATA_MPIIX	256
SECCOMP	224
VIDEO_OUTPUT_CONTROL	160
NLS_ASCII	160
SERIAL_8250_EXTENDED	128
NLS_ISO8859_1	128
UTS_NS	96
FDDI	64
EDAC	64
NLS_UTF8	32

Appendix C

Bugs fixed

The following three bugs/problems in the kernel's Kconfig implementation and DirectFB were fixed as part of working on this thesis and Kconfiglib:

1. Kconfig: fix MODULES-related bug in case of no .config (`http://lkml.org/lkml/2010/7/27/301`).
2. Kconfig: undefined symbols can crash dependency loop detection (`https://lkml.org/lkml/2011/2/4/248`).
3. DirectFB: fix hang on shutdown with Linux Input driver (`http://mail.directfb.org/pipermail/directfb-dev/2011-March/006108.html`).

Bibliography

[1] Fabrice Bellard et al. *QEMU, a generic open source machine emulator and virtualizer*. Dec. 15, 2011. URL: http://qemu.org.

[2] Peter Korsgaard et al. *Buildroot*. Dec. 15, 2011. URL: http://buildroot.uclibc.org.

[3] Opera Software ASA. *Opera Devices SDK*. Dec. 15, 2011. URL: http://dev/opera.com/sdk/.

[4] Andries E. Brouwer. *Some remarks on the Linux Kernel – Chapter 10, Processes*. Dec. 15, 2011. URL: http://www.win.tue.nl/~aeb/linux/lk/lk-10.html.

[5] Hui Chen and Keji Ren. *BeagleBoard Fast Boot*. Dec. 15, 2011. URL: http://code.google.com/p/swiftbeagle.

[6] Intel Corporation. *Extensible Firmware Interface (EFI)*. Dec. 15, 2011. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-homepage-general-technology.html.

[7] Pengutronix e.K. *ptx_ts utility*. Dec. 15, 2011. URL: http://www.pengutronix.de/software/ptx_ts/index_en.html.

[8] Ingo Feinerer. "Efficient configuration and verification of software product lines". In: *Proceedings of the 15th International Software Product Line Conference, Munich, Germany, August 22–26, 2011*. Ed. by Ina Schaefer, Isabel John, and Klaus Schmid. Association for Computing Machinery, 2011, p. 49. DOI: 10.1145/2019136.2019193.

[9] GNU Project Free Software Foundation. *GRUB Legacy*. Dec. 15, 2011. URL: http://www.gnu.org/software/grub/grub-legacy.en.html.

[10] AVM GmbH. *FRITZ!Box*. Dec. 15, 2011. URL: www.avm.de/en/Produkte/FRITZBox/index.html.

[11] Google Inc. *Chrome OS*. Dec. 15, 2011. URL: http://chromium.org/chromium-os.

[12] Texas Instruments Inc. *BeagleBoard*. Dec. 15, 2011. URL: http://www.beagleboard.org.

[13] Canonical Ltd./Scott James. *Upstart - Event-based init daemon*. Dec. 15, 2011. URL: http://upstart.ubuntu.com.

[14] Ulf Magnusson. *Kconfiglib: A flexible Python Kconfig parser*. Dec. 15, 2011. URL: https://lkml.org/lkml/2011/2/1/439.

[15] Ziga Mahkovec. *Bootchart*. Dec. 15, 2011. URL: http://www.bootchart.org.

[16] Angstrom team. *The Angstrom distribution*. Dec. 15, 2011. URL: http://www.angstrom-distribution.org.

[17] FHS team. *Filesystem Hierarchy Standard*. Dec. 15, 2011. URL: www.pathname.com/fhs/.

[18] Freetz team. *Freetz - a firmware extension for the AVM Fritz!Box*. Dec. 15, 2011. URL: http://www.freetz.org.

[19] Linux OMAP team. *Linux OMAP Kernel Project*. Dec. 15, 2011. URL: http://www.omappedia.org/wiki/Linux_OMAP_Kernel_Project.

[20] OpenEmbedded team. *OpenEmbedded – the build framework for embedded Linux*. Dec. 15, 2011. URL: http://www.openembedded.org.