Mid Project Report

Matthew J. Fleming
School of Computing
Leeds
scs5mjf@comp.leeds.ac.uk

December 6, 2007

Contents

Chapter 1

Introduction

1.1 Aim of Project

The aim of the project is to modify the NetBSD operating system to dynamically create device special files, used for communicating with devices, when devices are attached to a machine. This will require modification to the device configuration subsystem and implementation of a device file system (*devfs*) along with a device daemon which will create these device special files when the kernel detects new devices. Furthermore, the filenames for these device special files will be configurable via a configuration file that is used with the device daemon.

1.2 Objectives

- To modify the device configuration subsystem to announce when new devices are attached
- To implement a file system specifically for use with device special files
- To implement a device daemon to create device special files in the *devfs* when new devices are attached
- To implement rule matching code inside the daemon to read and parse a config file that specifies device node attributes

1.3 Deliverables

Along with a report this project will also produce a collection of files containing source code. This collection of files will implement the objectives. These files will hopefully be incorporated into the NetBSD operating system once the project is complete. A demonstration will also be performed for this project's supervisor showing the code running with the NetBSD operating system.

1.4 Project Schedule

- 15/11/2007 - Finish background reading

- 16/11/2007 - Write stubs for *devfs* and daemon
- 23/11/2007 - Implement rule matching code in daemon
- 07/12/2007 - Submit mid-project report and give the daemon knowledge of devices currently attached to the system
  - Provide a device driver /dev/dctl to get device information from an in-kernel list of devices that have not been handled by the daemon and provide a means of removing devices from that list once they have received attention
- 25/01/2008 - Implement the device file system and code to allow the daemon to create device special files in the file system
- 22/02/2008 - Start final report
- 07/03/2008 - Submit Table of Contents and draft chapter
- 23/04/2008 - Submit project report

1.5 Progress Report

So far, research has been done into the implementations that other operating systems offer for device file systems and a design has been formulated for providing a solution to the problem that the NetBSD operating system faces. The design of this solution aims to improve upon the shortcomings of other implementations and their design faults whilst also implementing features that many NetBSD developers deemed necessary. Code has been written in order to get the operating system to compile with the interfaces that this project will provide, however not all the implementations for these interfaces have yet been written.

Chapter 2

Background work

An operating system is the most fundamental system program of a computer. It is the job of the operating system to control all the computer's resources and provide a base upon which application programs can be written. This base provides an abstraction from the hardware of the machine and alleviates application programmers from concerning themselves with exactly how their program interacts with the hardware [1].

2.1 The UNIX Operating System

The UNIX operating system is the ancestor of many of today's operating systems. Its original incarnation was as a stripped down version of MULTICS, an operating system that Bell Labs were jointly designing with researchers at M.I.T and General Electric. This version of UNIX was called UNICS and was developed by a researcher at Bell Labs, Ken Thompson. The name was later changed to UNIX. Thompson was later joined by Dennis Ritchie, another researcher at Bell Labs and later by his entire department. Many versions of UNIX have been developed over the years but all were large and complicated systems from a programmer's point of view. Professor Andrew Tanenbaum wanted an operating system that was simple enough to be understood by a student. He created MINIX, a UNIX-like operating system that was simple enough that a student could understand it in its entirety. MINIX was freely available and because of this many people became interested in it and began asking the author for more features. However, these features were not implemented as the author stated that this would make the operating system too complex to be used as teaching matrial. Later, a Finnish student named Linus Torvalds created an operating system that was closely based on MINIX but with the goal that it would support some things that MINIX was lacking and provide a production system. He named this operating system Linux. Linux grew and has now become the operating system of choice in many businesses and universities.

Many organisations (and people) have modified the UNIX operating system or some descendant of UNIX. One such group was the University of California at Berkeley. Software from Berkeley is released in Berkeley Software Distributions (BSD) [2].

2.2 The NetBSD Operating System

The NetBSD project was founded by Chris Demetriou, Theo de Raadt, Adam Glass and Charles M. Hannum. It was based on 4.3BSD from the University of California

in Berkeley and 386BSD. The reason that NetBSD was forked from 386BSD was that its founders wanted to concentrate on multi-platform support. Since then, NetBSD has imported changes and source code from many other sources, including 4.4BSD Lite. NetBSD has also been used as the basis of other derivatives, including the University of Utah's Lites (on top of Mach), and Apple's Rhapsody. The NetBSD project is a collaborative project made up of developers from all around the world. The aim of the project is to produce a freely available UNIX-like operating system. This operating system is named NetBSD. Whilst the majority of the operating system consists of work contributed and written by NetBSD developers, other software, such as 4.4BSD Lite from the University of California in Berkelely, is also included in the operating system.

The NetBSD project aims to produce an operating system that is supported across a wide variety of platforms. To aid with this aim the operating system is intended to be extermely portable to ease porting to new platforms. NetBSD also implements many standard APIs and network protocols, and emulates many other systems' ABIs (application binary interfaces) [3]. This ensures that many standard system calls and library functions that are available on other UNIX-like systems are available on NetBSD. Examples of such interfaces are the socket functions, select(), read(), write(), etc. This allows programs to be run on NetBSD with a minimum of modification.

Each operating system in use today has some niche that makes it different from other operating systems available. NetBSD tries to run on a variety of platforms, Linux aims to provide a UNIX-like operating system suitable for use in production and MINIX is a small operating system suitable for study by students.

2.3 Licenses

Operating system source code has been released under a wide range of licenses. The source code in the NetBSD operating system is usually licensed under the Berkeley license. This license allows the source code to be modified and redistributed as long as credit is given to the author and the author's name is not used to promote any products based on their work. This liberal license also allows people to modify the source code and not redistribute it. The reason that this license is used is because it allows anyone to use the code for whatever reason they want, for example in commercial products. People are free to sell the code in any form, including binaries, without requiring that the source code be distributed as well. This is opposite to the GNU General Public License (GPL) license which requires that if you distribute binaries you must also give away the source code to build those binaries.

2.4 Computer Devices

A device is a piece of hardware that is attached to a computer system and has some function. Examples of devices include hard disks, printers, mice, keyboards. Devices can be classified in two ways, as characters devices or as block devices. Block devices include hard disks, magnetic tape devices and most random-access devices. Data can be read from or written to block devices in fixed-sized 'blocks' and each block has its own address. This means that each block can be written or read independently of all other blocks. Character devices accept or emit a stream of characters. Examples of character devices are mice, keyboards, network interfaces and printers. These devices are not addressable and have no seek operation which would allow individual addresses to be

read. Both types of devices plug into device controllers [4]. Input/Output (hereafter refered to as I/O) units usually consist of an electronic component and a mechanical component. The device controller is the electronic component and the device is the mechanical component. For example, a CD-ROM drive is mechanical, which attaches to an electronic IDE bus. The controller interacts with the device by reading or writing data to the device. When reading data from the device the controller performs error correction. The CPU interacts with the device controller by writing to and reading from control registers situated on the device controller. The CPU can write to the control registers to command the device to accept data, deliver data, switch itself on or off and perform other actions. If the CPU reads from the control registers it can learn the device's state.

2.4.1 Device Drivers

In order to keep track of which commands a device controller understands we have device drivers. For example, a disk driver must understand which commands to specify to move the arm on the hard disk to the location of data it wishes to read. This means that each I/O device that is attached to a computer requires some device-specific code for controlling it. This code is known as a device driver. Device drivers are usually contained in the kernel which allows them to run efficiently. Device drivers alleviate the need for the application programmer to write code that would deal with every device that their application may be run with, which would be an impossible task. As new devices are released by manufacturers drivers can be written for them for operating systems.

The way that UNIX operating systems traditionally access devices is by using files, usually in the "/dev" directory. These device nodes are created statically via a shell script such as NetBSD's MAKEDEV script that resides in the same directory. This MAKEDEV script is platform dependent with each platform specifying different major and minor numbers to be used for each device. Major and minor numbers are used by the kernel to specify exactly which device to communicate with. Each device driver uses the major and minor numbers differently. Usually, the device driver is assigned the major number and the driver then can use the minor number in any way it wants. An example of this is the SCSI/ATAPI tape device driver, st(4). This driver provides support for SCSI and Advanced Technology Attachment Packet Interface (ATAPI) tape drives. With this driver the minor numbers specify the mode that the driver should use for the device [5],

> "MODES AND SUB-MODES
>
> There are several different 'operation' modes. These are controlled by bits 2 and 3 of the minor number and are designed to allow users to eas- ily read and write different formats of tape on devices that allow multi- ple formats".

Each driver can be assigned a block and character major.

Now we've discussed how to locate a device driver we need some way to locate an actual device. Locating a device is done using the combination of major and minor numbers. We use a 32-bit type called a "dev_t" which is constructed as follows

> mmmmmmmmmmmmMMMMMMMMMMMMmmmmmmmm

where m = minor bit and M = major bit. Each device can be uniquely represented as a dev_t value. The kernel maintains two "switch tables", one for character devices and

one for block devices. These switch tables contain pointers to a structure that contains function pointers. These point to functions that allow a device to be manipulated, such as causing the device to be opened, closed, read, written to, etc. These switch tables are indexed by major numbers. A dev_t type is passed as an argument to each of the functions.

2.5 The Problem

The problem with the traditional UNIX approach is that nodes in ”/dev” are not created dynamically but instead are statically created with the MAKEDEV script. This means that all the devices that the operating system supports have a device node present in ”/dev” regardless of whether that device is attached to the system or not. These large numbers of nodes mean that it takes longer to search the ”/dev” directory for a particular node. Also, when new device drivers are added to the operating system the corresponding node must also be manually created in the ”/dev” directory. Another problem is that the number of major and minor numbers is finite and will eventually run out. Dynamically assigning the numbers at runtime is a much better use of the numbers as very few machines will have the need to use the entire range of possible combinations. The NetBSD operating system suffers from this traditional approach to device nodes.

2.6 Other Implementations

Other implementations of device file systems can be found for a variety of operating systems including Linux, FreeBSD, Solaris and Mac OS X. The details of each of these implementations differs and the following section will describe some of them.

2.6.1 FreeBSD

FreeBSD’s devfs is implemented entirely in the kernel. It has a rule matching feature that allows rules to be specified for certain devices, such as their permissions and their visibility to userland. These rules are specified via configuration files that can be found in the ”/etc” directory. The kernel maintains a set of data structures that describe various aspects of the devices that are currently present on the system. Unlike the traditional UNIX operating systems FreeBSD creates device nodes in the ”/dev” directory dynamically when they are connected to the system.

2.6.2 Linux

Linux’s udev handles devices in a different way. It makes use of several different subsystems in order to represent devices to the userland. When devices are detected by the kernel an entry is created in the ”sysfs” file system and a message is sent to a userland daemon like ”/sbin/hotplug” to notify userland that a new device has been discovered. The udev daemon that is running in userspace is sent a message with details of the new device. When the udev daemon is notified of this new device is looks in the sysfs file system for information about the device. This information is used to apply any rules that have been specified in ”/etc” for this particular device and then a node is created in ”/dev”.

Chapter 3

A Solution for NetBSD

This chapter will explain the design of the device file system and the daemon and how they interact. The reader should note that at this stage this design is not fixed and may differ from the final version. The design is made up of three major components, each of which will be discussed in detail in this chapter.

3.1 The *devfsd* Daemon

The *devfsd* daemon will be spawned after the init process starts and will run in the background. When it starts up it will read a configuration file that will contain rules that specify attributes for device special files. These rules will be matched against devices that are attached to the system and if a match is found the attributes for that rule will be used to construct a device special node for that file. The *devfsd* daemon finds out about new devices that are attached to the system by interrogating the kernel through a device driver, *dctl*. *ioctl* operations are performed on this driver to request entries on an internal list the kernel has of devices that are present on the system but for which *devfsd* has not yet processed. When *devfsd* has created a corresponding device special file it makes a call to the *ioctl* function for the *dctl* device driver to remove that device from the kernel's list.

3.1.1 Rules

A rule is comprised of two parts, a match list and a list of attributes. A match list specifies how to match a rule against a device. Rules can be matched by label or serial number (such as is available for SCSI and USB devices), manufacturer and model and finally a device can be specified by using the old-style device name. The old-style device name is the name that devices are currently assigned in NetBSD which is made up of the device driver name, for example wd, and an instance number. For example if a hard drive is attached to the NetBSD machine the corresponding device special file in the "/dev" direcotry will be, wd0. If another hard drive is attached the device special file for that device will be wd1, where the 'wd' portion of the name is the device driver name and the '1' is the instance number.

The list of attributes specify properties of the device special file. Attributes that can be configured are the permissions for that device special file, the filename, the owner and group for that file and the device special files visibility to userland. Visilbility

refers to whether or not a file is created in the *devfs* file system. For example, if the attribute for visibility states that this file should be invisible, no device special file will be created and userland applications will not be able to access this device.

3.2 The *dctl* Device Driver

This device driver controls a pseudo device that will allow communication between the kernel and userland daemon, *devfsd*. The *devfsd* daemon will open the *dctl* device and poll for any new data regarding new devices that have been attached to the system. The kernel will add any new devices that are attached to the system to its internal list of devices that need attention from *devfsd*. When the kernel adds a new device to the list it will notify the *dctl* device. When *devfsd* discovers a new device, it queries its rules for any that match and creates an internal representation of a device special file. It then passes this as an argument to *dctl*. *dctl* proceeds to create an entry for the device special file in the *devfs* file system.

3.3 The *devfs* File System

The *devfs* file system is based on the tmpfs file system that is avaible on NetBSD. The tmpfs file system is an efficient memory file system. Due to the nature of device special files, namely that they are merely gateways through which applications and users can access devices (and whose attributes can be constructed from the *devfsd* config file), there is no need to write them to disk. This file system is different from almost all other file systems because files will not be allowed to be created in the file system from userland processes. Instead the *dctl* device will create the files in the file system.

Bibliography

[1] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Pearson Prentice Hall, 2006.

[2] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[3] The NetBSD Project, 2007. About the NetBSD Project - http://www.netbsd.org/about/.

[4] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, 2001.

[5] The NetBSD Project, 2007. NetBSD Manual Pages - http://netbsd.gw.com/cgi-bin/man-cgi?st++NetBSD-current.