University of Alberta

TRIGGER SCRIPTS FOR EXTENSIBLE FILE SYSTEMS

by

A. Cameron Macdonell 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-81435-1

Canada

University of Alberta

Library Release Form

Name of Author: A. Cameron Macdonell

Title of Thesis: Trigger Scripts for Extensible File Systems

Degree: Master of Science

Year this Degree Granted: 2002

A. Cameron Macdonell
11445 79th Ave
Edmonton, AB
Canada

Date: Sept. 26/2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Trigger Scripts for Extensible File Systems** submitted by A. Cameron Macdonell in partial fulfillment of the requirements for the degree of **Master of Science**.

Paul Lu
Supervisor

Pawel Gburzynski

Mike Carbonaro

Date: Sept 19/2002

Abstract

We introduce *trigger scripts* as a mechanism for extensible file systems that are implemented at the user level. We describe the design, implementation, and evaluation of trigger scripts in the Linux/VFS-based Scruf framework. Using trigger scripts, we have prototyped five simple file systems: instrumented, compressed, encrypted, version-controlled, and a novel remote data access file system. Using a modified Andrew Benchmark and other tests, we show that the overheads of Scruf vary from low to moderate, depending on an application's I/O patterns.

For developers, the advantages of trigger scripts include modularity, factoring complexity out of the kernel, and minimal operating system modifications. For system administrators and non-privileged users, the advantages of trigger scripts include simplicity, flexibility, and compatibility with unmodified applications. We will demonstrate that useful extensions can be implemented quickly and easily using the Scruf framework and trigger scripts.

Acknowledgements

Although this work is published under my name, I would be remiss not to mention the key contributions and support of several people. Most notably, I must thank my supervisor Paul Lu for providing unending guidance, advice, and motivation in helping me to achieve this degree. Jeremy Handcock for his contribution to the Scruf-Trellis portion of this work. In my day-to-day toils, I owe a great debt of gratitude to my lab mates Steve, Ernie, John, Matt, Maria, Kit, and James for their help and humour, which made the lab a wonderful environment in which to work. Finally, my family for their continuous love and support despite the long distance between us.

Ad Majorem Dei Gloriam

To Mom and Dad

Contents

1 Introduction 1
- 1.1 Design Goals and Features . . . 1
- 1.2 Overview . . . 3

2 Overview 4
- 2.1 A Motivating Example . . . 4
- 2.2 Contributions . . . 6
- 2.3 Architecture and Basic Concepts . . . 8
  - 2.3.1 VFS trigger scripts . . . 9
  - 2.3.2 Path Scripts . . . 10
- 2.4 "Hello World" with VFS Scripts . . . 11

3 Related Work 13
- 3.1 Introduction . . . 13
- 3.2 Other Research Projects . . . 15
  - 3.2.1 Microkernels . . . 15
  - 3.2.2 Stackable File Systems . . . 16
  - 3.2.3 User-Level File Systems . . . 16
- 3.3 Concluding Remarks . . . 19

4 Trigger Scripts for Extensible File Systems 20
- 4.1 Introduction . . . 20
- 4.2 Trigger Script Operation . . . 20
- 4.3 File System Extensions in Scruf . . . 22
  - 4.3.1 Scruf-Trace: An Instrumented File System . . . 22
  - 4.3.2 Scruf-Compress: A Compression File System . . . 25
  - 4.3.3 Scruf-Crypt: An Encrypted File System . . . 26
  - 4.3.4 Scruf-CVS: A Version-Controlled File System . . . 27
  - 4.3.5 Scruf-Trellis: A Remote Data Access File System . . . 27
- 4.4 Discussion . . . 29

5 The Scruf Framework 30
- 5.1 Introduction . . . 30
- 5.2 File System Basics . . . 31
  - 5.2.1 Linux's Virtual File System . . . 31
  - 5.2.2 Mount Points . . . 32
  - 5.2.3 VFS Data Structures . . . 34
  - 5.2.4 Pathname Lookup . . . 35

5.2.5 Linux Kernel Loadable Modules . . . 35
5.3 OS Pathways . . . 36
5.4 A More Detailed Example . . . 37
5.5 Scruf-VFS: The Scruf Kernel Module . . . 39
5.5.1 Data Structures . . . 40
5.5.2 Code Modifications . . . 42
5.6 Kernel-Daemon Communication . . . 55
5.6.1 The `/proc` File System . . . 55
5.6.2 From the Kernel to the Daemon . . . 59
5.6.3 From the Daemon to the Kernel . . . 61
5.6.4 Out-of-band Communication . . . 62
5.7 The User-Level Daemon . . . 63
5.7.1 Daemon-Script API . . . 64
5.7.2 Daemon-Script Communication . . . 66
5.8 Improving the Performance of VFS scripts . . . 67
5.8.1 Symbolic Links . . . 67
5.8.2 Inheritance . . . 68
5.8.3 Filtering . . . 70
5.8.4 Persistent Scripts . . . 71
5.8.5 Synchronous v. Asynchronous Scripts . . . 73
5.9 Concluding Remarks . . . 74

6 Experimentation and Performance **75**
6.1 Microbenchmarks . . . 75
6.1.1 Synchronous v. Asynchronous Scripts . . . 76
6.1.2 Persistent Scripts . . . 78
6.2 Building the Linux Kernel . . . 79
6.3 A Modified Andrew Benchmark . . . 80
6.4 Running Scruf without Scripts . . . 82
6.5 File Systems Which Use Redirection . . . 82
6.6 Scruf-Trellis . . . 84
6.7 Discussion . . . 85

7 Concluding Remarks **86**
7.1 Summary of the Scruf Framework . . . 86
7.2 Future Work . . . 87

A Scruf-Trace Script **88**

B Scruf-Compress Scripts **90**
B.1 The Scruf-Compress .on.open script . . . 90
B.2 The Scruf-Compress .on.release script . . . 90
B.3 The Scruf-Compress .on.write script . . . 91

C Scruf-Crypt Scripts **92**
C.1 The Scruf-Crypt .on.open script . . . 92
C.2 The Scruf-Crypt .on.release script . . . 92
C.3 The Scruf-Crypt .on.write script . . . 93

D Scruf-CVS Script **94**
D.1 The Scruf-CVS .on.open script 94

E Scruf-Trellis Path Script **95**

Bibliography **96**

List of Tables

3.1 Example Design Options for Extensible File Systems 17

4.1 Example Scruf File Systems Using Trigger Files 21
4.2 Summary of VFS Trigger Scripts Required for Extensions. `.on.open` scripts perform either Interposed (`I`) or Redirection (`R`) Action. 29

5.1 Summary of Trigger Script API 66

6.1 Scruf-Trace vs. `ext2` building the Linux 2.4.18 Kernel. Times given are in seconds. 79
6.2 Scruf-Trace vs. `ext2` under a modified Andrew Benchmark. Times given are in seconds. 80
6.3 Scruf (no trigger scripts found) vs. `ext2` building the Linux 2.4.18 Kernel. Times given are in seconds. 82
6.4 Scruf (no trigger scripts found) vs. `ext2` under a modified Andrew Benchmark. Times given are in seconds. 83
6.5 Remote Data Access using Trellis scripts. Times given are in seconds. . . . 84

List of Figures

2.1 An example directory hierarchy and trigger scripts 6
2.2 The Design of the Scruf File System 8
2.3 A "Hello World" trigger script 11

3.1 Extensible File System Models 14

4.1 Scruf-Trace: Understanding a Compilation: Time Line (top) and File Coverage (bottom) 24

5.1 Linux's Virtual File System (VFS) 31
5.2 Structure of VFS file operations table 32
5.3 An Example of Mounted Partitions 33
5.4 Scruf File System and Trigger Scripts 37
5.5 The Bounded Buffer used in Scruf-VFS 40
5.6 Structure of data in a ring element 41
5.7 Structure of data passed to the daemon 41
5.8 Structure of data passed back to the Scruf module 42
5.9 Structure of data to manage ring 42
5.10 `ext2` and Scruf-VFS's file operations tables 43
5.11 Scruf lseek operation function 47
5.12 Scruf open operation function (a) 48
5.13 Scruf open operation function (b) 49
5.14 Scruf read operation function 50
5.15 Scruf write operation function 51
5.16 Scruf release operation function 52
5.17 A Path Script's Basic Operation 53
5.18 The dentry hash function (a) 56
5.19 The dentry hash function (b) 57
5.20 Scruf-VFS `/proc` read function 59
5.21 A read of `/proc/scruf` 60
5.22 Scruf-VFS `/proc` write function 61
5.23 A write to `/proc/scruf` 62
5.24 An Example of Scruf's Inheritance Mechanism 69
5.25 An Example of Scruf's Filtering Mechanism 71
5.26 A persistent script 72
5.27 IPC between the Daemon and a Synchronous VFS Script 73
5.28 IPC between the Daemon and an Asynchronous VFS Script 74

6.1 The IPC Pathway in Scruf 76

6.2 Synchronous v. Asynchronous Scripts: Sending 100,000 Messages 78
6.3 Scruf-Compress vs `ext2`: Running GhostScript on Compressed Files . . . 83

Chapter 1

Introduction

A common and important theme in systems research is how to increase the flexibility and modularity of operating systems (OS) and the services an OS provides. System administrators and developers may want to selectively configure and extend the basic OS functionality as they are able to with window managers, web browsers, email clients, and other user-level applications. Also, non-privileged users may want to customize different OS services. For example, user-recoverable file systems which store previous versions of files are desirable. The desired functionality can vary from user to user and from one part of the directory hierarchy to another. The typical challenges for any OS extensibility mechanism include generality, flexibility, security, performance, and compatibility with existing OS architecture.

For the particular domain of file systems, we propose *trigger scripts* as a simple, flexible, and low-overhead approach to extensible file systems. Trigger scripts are user-level programs that are invoked in key file system pathways in order to implement new functionality. We have implemented trigger scripts as part of the Scruf (Scriptable User-Level File System) framework. Our changes to the Linux `ext2` loadable module are simple and localized, which should make it easy to implement Scruf for other OSes. To introduce this work, we will begin by describing the design goals of the Scruf framework and how trigger scripts are the main component of its design.

1.1 Design Goals and Features

When considering how to extend file system functionality and allow individual users to choose their own extensions, one of the major decisions is whether the extensions should be part of the kernel, or exist in user-level and communicate with the kernel via interprocess communication (IPC). Arguments have been made against user-level file systems due to the

high cost of the IPC and the continuous context switches between user and kernel level that these systems require [26]. Arguments have also been made in favour of user-level support not only for file systems, but also for other OS functionality, in the research of microkernels [16]. However, the alternative explored in this work is to enable the file system to become selectively user-level, that is, only portions interact with the user-level and only when needed. In this way, some user-level overhead can be avoided. The Scruf framework implements this design in order to keep its performance as close as possible to a kernel-level file system, while still providing the extensibility of a user-level file system.

There are also other considerations of an extensible user-level file system that the design of Scruf takes into account. First, once the kernel-side installation of trigger scripts—called Scruf-VFS—is complete, modifications may be made to the user-level portion without administrator intervention. Second, the same kernel infrastructure is used for all extensions, meaning that no further kernel modifications are necessary for additional extensions.

Trigger scripts are regular Unix files, stored in user directories. They are simply executable scripts which can extend the operation of the underlying file system with functions that are not normally part of kernel-level file systems. With trigger scripts, a user is free to experiment with the user-level code at his or her discretion, while working towards desired functionality, without affecting other users or compromising the system in general. Users should also be able to easily add or remove extensions from a running system, either in part or all together. When the extensions are removed, the system must operate at a performance level similar to a typical kernel-level file system, in this case `ext2`. Since trigger scripts are autonomous programs which run as normal user-level programs, it is important that the extension have a simple and well-designed interface to interact with the underlying file system. If an interface is too difficult or awkward, it will discourage users from using Scruf. Scruf has been designed with an application programming interface (API) that is simple and language-independent, and that allows users to begin with extremely simple trigger scripts while, at the same time, providing more options for advanced users. The API has also been designed to reduce the file system understanding a Scruf user would require. By presenting user-level file system concepts such as opening and closing files, and hiding kernel-level concepts such as inodes, disk blocks, and other low level file system implementation details, Scruf allows a user to modify the file system operations without requiring extensive file systems implementation knowledge.

Using trigger scripts, we have implemented a remote data access file system and a version-controlled file system which allow users to recover previous versions of their files.

The new functionality can be used by unmodified, binary applications. Both systems are examples of frequently-requested file system functionality. To explore the generality of trigger scripts as an OS mechanism, we have also implemented three other types of file systems: instrumented, compressed, and encrypted.

Since trigger scripts run with non-privileged permissions, security issues are minimized. And, since the kernel modifications required to support trigger scripts are minimal, the trigger script concept is compatible with many mainstream OSes. New functions that require a lot of engineering and OS-specific changes are not as likely to be adopted in practice.

1.2 Overview

In the next chapter, we will begin by presenting a high-level overview of the Scruf file system framework and trigger scripts. Chapter 3 discusses other research projects that have explored extensible file systems, and how those projects compare to Scruf. In Chapter 4 we present the five file system extensions we have implemented using the Scruf framework and trigger scripts. We will then present the Scruf framework in both its design and implementation. Chapter 6 will describe our experiments to measure the performance of our extensions, as well as some measurements of the overheads inherent to the Scruf framework, such as IPC. We will conclude with a brief summary of the Scruf framework and future goals for Scruf.

Chapter 2

Overview

Normally, file systems are fairly passive components of an operating system, which makes it convenient to put data into stable storage, and to find and access the data via a namespace. Traditional research topics in file systems include optimizations related to physical data layout, I/O scheduling, and caching within the OS. These issues remain important. An orthogonal goal is making file systems more active and providing higher-level functionality. The Scruf framework is designed to make file system extensions as simple as possible to implement, while allowing significant functionality in the extensions.

2.1 A Motivating Example

Why would a user want an extensible file system? Here is an example: It is common for a user to accidentally delete a file, or for a software developer to want to rollback to a previous version of a file. Typically, these situations are handled by recovering from backup tapes (or copies) or by the disciplined use of a version-control system (e.g., CVS [7]). However, automated backups are typically done only once a day and the amount of discipline required to do backups and use a version-control system is high. Version-control systems require users to update the backups on their own. For example, with CVS, a user must execute the command `cvs commit` *filename* to save a version of a file. To do this continually is burdensome, and users often go long periods without updates, which can be problematic when trying to recover a previous version. One solution would be to have the file system automatically maintain frequent backups and versions. This is an example of a file system extension.

As mentioned, a *commit* operation creates a new version of the file in the repository which contains the changes since the last commit. Functionally, a basic version-controlled file system would be equivalent to doing a `cvs commit` of every file before it is used

(i.e., opened) or after it is modified (i.e., closed). If there have been no changes to the file during the last open-to-close access, then there is nothing to commit to the file repository. Therefore, the core functionality of a version-controlled file system using CVS is simple; in fact, it is one line,

`cvs commit` *filename*

What is needed is the ability to have this one line executed every time a file is opened. This is the type of task a user would usually delegate to a script, however it would be desirable to avoid having to explicitly run the script every time a file is accessed. It would be preferable to have it done automatically by the file system. These automatically executed scripts are what we now refer to as trigger scripts.

A similar approach is taken by many applications with resource configuration files (also called RC files). To explain, text editors such as Emacs and Vim are highly configurable, with literally thousands of options, and no user wants to input their preferences every time they start the editor. To avoid this, a user can store their preferences in a RC file named, for example .vimrc and .emacs. The names of the RC files are built into the application and so the user must use the correct RC filename. Every time a particular application is started it looks for its RC file, reads it, and sets the options accordingly. The application does this automatically and the user only needs to setup the RC file once. The two important differences for trigger scripts are that instead of storing preferences, a trigger script stores executable commands. And secondly, trigger scripts do not require the applications to be changed; instead, the file system handles their execution.

For a version-controlled file system, a user needs a way to tell the file system to run this trigger script every time a file is opened. Consider Figure 2.1. In the directory **/usr/alberta/** there is a file named `.on.open`. In Scruf, the name `.on.open` tells the file system that this file is a trigger script and that it should be executed every time a file is opened in that directory. To implement simple version control with CVS, `.on.open` can be written as a simple shell script, such as

```
#!/bin/sh

cvs commit $1
```

Every time a file is opened in that directory, the Scruf file system will search for an `.on.open` script, execute it, and pass in the filename being opened (as the `$1` parameter), along with other information describing the open. By having this simple script executed

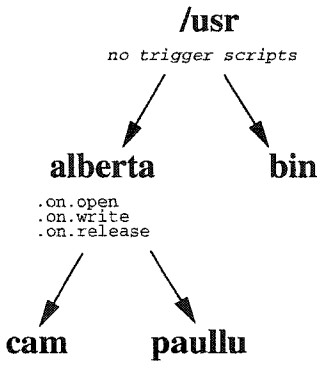

Figure 2.1: An example directory hierarchy and trigger scripts

every time a file is opened, the user will achieve the version control desired. This concept of running a user-defined executable script when a file access, such as an open occurs, is the basis for the Scruf file system and `.on.open` is an example of a trigger script.

Using trigger scripts, we have implemented a basic, version-controlled file system, similar to the one described above, called Scruf-CVS. Scruf-CVS is layered on top of an existing local file system in order to take advantage of all the low-level layout, scheduling, and caching optimizations. Again, those features remain important and we do not want to have to re-implement them. And since Scruf-CVS uses CVS, it does not have to re-implement that functionality either. In addition, due to the architecture and design of trigger scripts, Scruf-CVS is simple to implement, it is configurable on a directory-by-directory basis, and each user is able to customize it. Scruf-CVS will create a lot of versions of a file; however—given the rapidly improving price-density of storage technologies—it is entirely practical to save many versions of a file [23], especially if only the changes (not a whole new copy of the file) are saved between versions, as with CVS.

2.2 Contributions

New applications (e.g., grid computing) and new trends (e.g., cheap storage) present new challenges to the traditional topic of file systems. Classic problems such as device formats, scheduling, and caching are still important. However, higher levels of abstraction are also in demand with no clear mechanism or means to provide the new functionality. The topic of extensible file systems remains a worthwhile research area.

The contributions of this work include:

1. **The introduction of trigger scripts as a new, fundamental mechanism for ex-**

tensible file systems. Similar concepts of implicit behaviour have been used elsewhere in systems research (e.g., cascading deletes in PostgreSQL [21], RC files), but they have not been (to our knowledge) generalized and applied to extensible file systems. Also, there are no widely-used implementations of extensible file systems that are similar to Scruf. Trigger scripts have other advantages over other OS extension mechanisms, including flexibility, security, and compatibility with unmodified applications.

2. **Demonstrating that trigger scripts are useful across a spectrum of extensions.** The Scruf framework and trigger scripts have been used to implement an instrumented file system, as well as encrypted, compressed and version-controlled file systems. As well, a remote data access file system has also been implemented. More extensions via trigger scripts are planned. The idea behind trigger scripts is simple, but our early experiences are encouraging as to its generality and utility.

3. **Description of a design, implementation, and performance evaluation of trigger scripts** as part of the Scruf framework. We show that, by using trigger scripts, the extensions mentioned above can be implemented with acceptable overhead. Moreover, when no extensions are used, the overhead of the Scruf framework itself is less than 1% that of a conventional file system.

The design goals of trigger scripts and Scruf include:

1. **Simplicity.** Both kernel developers and average users should be able to create useful file system extensions.

   Using the system as implemented allows for trigger scripts to be written in any language that is usable under Linux, and which may be, as we will show, as simple as a single line of code. Most of our trigger scripts are less than 40 lines of code.

2. **Flexibility.** A wide array of extensions should be possible.

   We will discuss the five different file system extensions we have written. Some of our examples are simple (though useful) and the functionality has been proposed, and sometimes implemented, in other systems. However, we show that it is possible to implement basic versions of all the systems using the Scruf framework.

3. **Compatibility.** Unmodified binary applications should be compatible with trigger scripts. Also, the kernel changes required to implement Scruf should be limited and

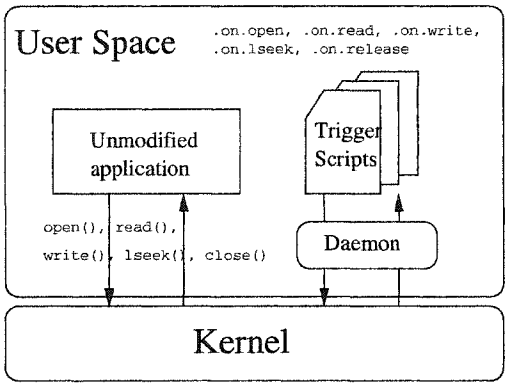

Figure 2.2: The Design of the Scruf File System

portable to existing OSes and OS architectures.

2.3 Architecture and Basic Concepts

Architecturally, Scruf is an extension to the well known Second Extended file system, also called `ext2`. The basic structure of the Scruf framework is shown in Figure 2.2 and is composed of 3 main parts:

1. **A kernel-resident file system module**, Scruf-VFS, that controls the disk device and communicates with the user-level. It is part of the kernel in Figure 2.2.

2. **A user-level daemon**, which receives messages from the kernel module and runs the trigger scripts.

3. **The trigger scripts**, which encapsulate the operations of the extensions, (i.e., CVS commands).

The file system is modified such that selected file operations result in interprocess communication (IPC) with the user-level daemon, which executes the relevant trigger script. Trigger scripts, running as non-privileged processes (which limits security problems), are passed information describing the file operation which has occurred. The trigger script can take appropriate action and returns information to the file system which may modify the original operation.

Triggers scripts are regular programs which interact directly with the user-level daemon via a language-independent API. This means they can be written in almost any programming language the user chooses, so a new library or language does not have to be learned.

An advantage of this is that the user can utilize any existing libraries which are accessible to the language he or she chooses, which can go a long way in helping the user to avoid having to port any functionality—as they would if a new language were designed (assuming it would even be possible in the new language). For example, in the Scruf-CVS case, CVS—well-designed, popular and free versioning software—is used straight out of the box. In the same way, a trigger script is able to use the network—something which would be difficult or impossible in a kernel-level file system.

Trigger scripts provide another advantage over most kernel-level, and even user-level file systems, in that Scruf is designed to be easily changed. The other file systems that we have examined are written exclusively in C and C++. While the changes are localized, a user must have an extensive working understanding of how the file system operates. Scruf limits the users' exposure to the underlying operations to only that which is necessary by way of the API. The user does not need to look at underlying file system structures or operations, only at the operations permitted via the API, which are much more intuitive and easily understood by average users. The scripts control the file system via the API which limits the knowledge required to modify the file system's behaviour, but still retains expressibility for more experienced users. Currently, there are two kinds of trigger scripts: VFS and Path.

2.3.1 VFS trigger scripts

The first class of trigger scripts, called *VFS trigger scripts*, are executables which, by convention, are named `.on.open`, `.on.read`, `.on.write`, `.on.lseek`, and `.on.release` [1]. VFS is an acronym for *Virtual File System*, which is part of the Linux kernel and defines what file operations are valid; it will be discussed in more detail in the next chapter. The leading "`.`" makes a file invisible to an `ls` command by default, so that the trigger scripts do not clutter directories. The name of the trigger script binds it to the respective file operation. A "release" operation occurs when the file is closed by the last process accessing it. The mere presence or absence of a script with a reserved filename configures Scruf's behaviour, which makes installation as simple as placing the scripts in the correct directory and uninstallation as simple as deleting or renaming the scripts. To further illustrate the function of VFS trigger scripts, Figure 2.1 shows a directory hierarchy. One directory, **/usr/alberta/**, contains three trigger scripts. Whenever an open, write, or release operation occurs in that directory the corresponding trigger script will be invoked.

[1] Equivalently, we refer to them as on-open, etc., trigger scripts.

The invoked script may return a value which tells the VFS to modify the file operation, or allow it to go ahead unchanged.

As explained earlier in this chapter, in our version-controlled extension, Scruf-CVS, the trigger script `.on.open` can be as simple as a single line of shell code:

```
cvs commit $1
```

where Scruf passes the name of the file to be opened as the command-line argument `$1`. Note that the same script renamed as `.on.release` would be equally functional, but with slightly different semantics: the file is committed after it is closed, instead of after it is opened. This type of trigger script is an example of one which does not change the file operation; the file is still opened, but the commit action is also executed by the trigger script.

The scripts can be simple or arbitrarily sophisticated; our scripts so far are typically 40 lines long. Any combination of the five possible trigger scripts can exist in any directory. Symbolic links to scripts are also allowed. If there is no trigger script corresponding to a particular VFS file operation in the application's CWD, the Scruf-VFS component filters out those actions to avoid an expensive IPC. Optionally, trigger scripts can be inherited. Using Figure 2.1 as an example, the directory **/usr/alberta/cam** can inherit the scripts from **/usr/alberta/** and they would be invoked as if they were present in **/usr/alberta/cam/**, whenever an open, write or release operation occurs. No symbolic links or copying is required between the ancestor and the inheriting directory, which allows the scripts to have wide effect with minimal setup. In Figure 2.1, the trigger scripts could modify all operations in the entire hierarchy by placing the scripts in the **/usr** directory and the scripts would be inherited into all the lower directories. Inheritance can only come from an ancestor up the hierarchy. Thus, all file operations that occur in **/usr/bin** in Figure 2.1 will be filtered out, since no trigger scripts exist in that directory, nor do any exist in its ancestor(s) **/usr**. Inheriting trigger scripts will be discussed in detail in Section 5.8.2.

2.3.2 Path Scripts

The second class of trigger scripts are not invoked implicitly with respect to file operations. *Path trigger scripts* are invoked if they are explicitly named in the pathname of the file passed to an open system call. For example, if `open()` is called with pathname:

```
/usr/scovil2/mypathscript:path_string
```

and `/usr/scovil2/mypathscript` is an executable, then `/usr/scovil2/mypath script` is executed as a path trigger script with `path_string` as a command-line parameter. The key which Scruf responds to is the colon in the pathname. If a pathname contains a colon, Scruf will assume the path previous to the colon is the pathname of an executable, the path trigger script, and the string following the colon is an argument to be passed to the path script. This path string can contain any characters which are valid in a filename. Unlike VFS trigger scripts, path scripts can have arbitrary names since the user provides their name to invoke them. We will be using path trigger scripts as part of our remote data access file system, called Scruf-Trellis.

2.4 "Hello World" with VFS Scripts

As mentioned earlier, the central mechanism for extension in the Scruf framework consists of the trigger scripts. Individual scripts were chosen as they are a simple concept, understood by any almost Unix programmer, which flattens the learning curve in using Scruf and trigger scripts for file system extensions.

When learning anything new in systems, whether it be a new language or library, or a new utility such as sockets or pipes, it is important that a new user is able to implement simple functionality quickly. For example, the most important step in learning a new language is getting a Hello World program to compile and execute properly. Allowing a user to implement an extension in a language of their choosing (i.e. C, Perl, Python) enables them to feel immediately comfortable with the system and frees them from the difficulty of having to learn new syntax or structure (and since communication with the daemon is via standard in and standard out, almost any language is acceptable). With Scruf, our version of the Hello World program would be the Hello World trigger script in Figure 2.3.

```
#!/bin/sh

echo Hello World >> myLog
```

Figure 2.3: A "Hello World" trigger script

By placing the Hello World trigger script into a file named `.on.open` in a directory which is mounted on a Scruf partition, a line reading "Hello World" will appear in the file `myLog` every time a file is opened in that directory. While the user can slowly add functionality to their scripts, this immediate implementation helps them to understand what

they are doing and to begin to imagine what extensions they could implement using Scruf trigger scripts. To remove the functionality, a user simply removes the `.on.open` script or renames it to an non-trigger script name.

Chapter 3

Related Work

3.1 Introduction

To further understand where Scruf fits in file systems research, a discussion of other extensible file systems which have been described in the literature will now be undertaken. We begin with a description of the three general models of extensible file systems.

The concepts of extensible file systems and user-level file systems are not new. The extensibility mechanisms we have seen in file systems research can be grouped into three different models, demonstrated in Figure 3.1. They are: microkernels, stackable file systems, and user-level file systems. It is important to note that the diagrams are generalizations, however they do summarize how the different models compare. The following discussion of these models is summarized in Table 3.1.

The first model, microkernels, is arguably the most commonly-used mechanism for extensible file systems (e.g., Mach [4] and Mac OS X[17]). For example, microkernels can be combined with different user-level (possibly privileged) servers for file systems, virtual memory paging, and process scheduling. Different file, memory, and process servers can run concurrently, thus providing different functionality to different users. In a microkernel, the file system server exists entirely in user-level. To achieve extensibility the entire file system would have to be duplicated and the extension would be added to create a new server.

In contrast, a monolithic kernel can allow trusted, privileged modules to be loaded when a new file system is dynamically mounted. The Virtual File System (VFS) interface allows different file systems and device formats to be supported by the OS. More recently, there is a substantial body of research on how to extend the functionality of the OS by downloading code into the kernel or by combining building blocks of functionality in a stackable file system (for example, [26, 15]). This bring us to the second model—stackable file systems.

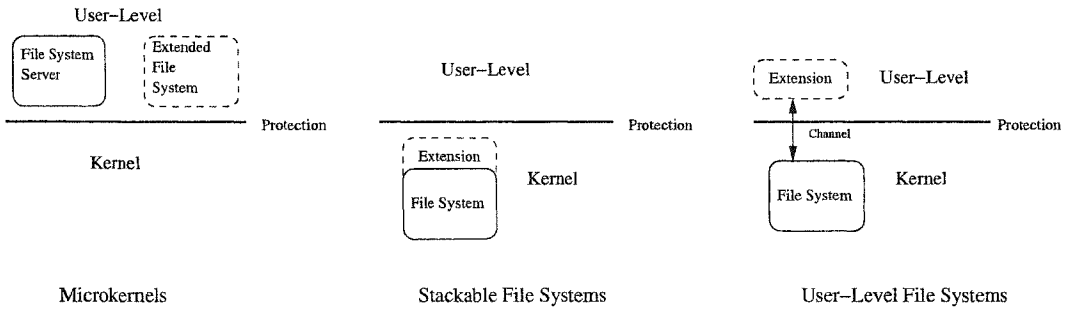

Figure 3.1: Extensible File System Models

They allow users to modify file system reads, writes, and mmaps to manipulate filenames and data in a stackable Vnode interface [14]. Extensions are stacked on top of existing file systems and, when file operations occur, they pass through the stacks, may be modified by the extensions, and then onto the basic file operations, if necessary. Multiple extensions may be stacked on top of each other.

Three important limitations of the above two approaches to extensible file systems are (1) the inability of non-privileged users to create safe, custom extensions, (2) limited functionality in the extensions (e.g., network access), and (3) the relative difficulty of re-engineering an existing OS to support extensions. Downloading code or creating file system stacks usually requires superuser privileges and, possibly, substantial kernel-programming expertise. And, downloaded code may not be allowed to perform arbitrary functions, which limits their generality and flexibility. For example, arbitrary network access may be disallowed due to security concerns or because network operations may take too much time for a kernel function. Lastly, other extensibility mechanisms may require substantial changes to and grafting of code into existing OSes. If a user attempts to directly modify file system code free of any framework, OS code is by no means trivial. Extension would require extensive review to understand the code the user wants to modify.

The third model for extensible file systems is user-level file systems. User-level file systems are designed to deal with the issues above. User-level extensions can be implemented without privileged access, and extensions may be applied to individual users or system-wide. Typically, user-level systems still require some basic kernel-level function primarily to communicate with the user-level extensions. However, their creation and installation is typically simplified to be usable by those without extensive kernel programming experience.

3.2 Other Research Projects

We will now describe other research projects that have explored the topic of extensible file systems and, more generally, extensible operating systems. Microkernels will be discussed first followed by stackable file systems and, finally, user-level file systems.

3.2.1 Microkernels

The concepts of Scruf and user-level file systems are similar to the more general idea of microkernels. Microkernels are operating systems designed to minimize the amount of functionality placed in the kernel and to move as much functionality as possible to user-level servers. The best known microkernel designs are Mach [4], Amoeba [18] and Exokernel [12]. However, while microkernels have had significant impact on operating systems, extending them and the associated file systems still requires significant understanding and development time. Some microkernels have suffered notorious problems with poor performance from the high volume of IPC between the kernel and the user-level servers [16].

The Spring File System [13] is an object-oriented (OO) microkernel. Similar to other microkernels, it implements user-level servers to handle functionality such as file systems. Since it implements an object-oriented model, file systems can be extended much like classes can be in an OO language. For example, as Scruf is built and woven into the `ext2` file system, a similar extension in Spring would inherit the underlying file system objects and replace, extend, or overload file system operations as necessary, which makes for a very clean interface for file system extension. However, Spring's concepts are not easily portable to any OS that is not object-oriented.

The HURD project [6] is an operating system project from the Free Software Foundation which is based on the Mach microkernel. HURD implements *translators* which translate arbitrary pathnames to real pathnames and can create the translated pathnames as needed. For example, a translator would be provided for the mount point `/ftp` and a `cd` (or any file command) to `/ftp/ftp.machine.com` would give transparent access to the remote ftp site as if it were a local directory. This idea is similar to Scruf's trigger scripts. However, HURD requires that code be written in C and that every function that is part of the translator API be implemented. HURD has not been extensively analyzed in literature, however its contributions are still valid and interesting.

3.2.2 Stackable File Systems

The WrapFS interface [25] is an approach to extensible file systems using extensions which are downloaded into the kernel. WrapFS is an example of a stackable file system extension. WrapFS has been used to implement file system extensions such as encryption, access snooping, and file redistribution. However, WrapFS still requires the user to write kernel-level C code, although the code is not as complex as standard file system code. The follow-up project to WrapFS was FiST [26], in which users write file system extensions in the FiST language and file system stacks are generated automatically by its compiler. The generated stack is loaded and run in the kernel to avoid the overhead of communicating with user-level, while still providing extensibility. Positive aspects of FiST include good portability and low overhead, since it runs entirely in the kernel. Some disadvantages of its design are (a) that it requires users to learn a new language, whereas Scruf allows users to modify the file system using a language of their choosing, and (b) FiST requires the user to understand how files work in the kernel, where Scruf allows users to think of files as they are used at the user-level.

3.2.3 User-Level File Systems

The Ufo Project [1] explored file system extensions via user-level extensions. Ufo works by intercepting and changing system calls using the same system call trap mechanism as debuggers. The system call *Catcher* which intercepts and modifies the system calls is a user-level process. Ufo has some advantages over Scruf. First, it requires no modification below the user-level, or intervention from system administrators. Second, it is run entirely at the user-level and does not require its own partition or root access to be installed. Ufo was used to implement a Global File System similar to Scruf-Trellis, which provides access to remote machines via authenticated and anonymous http and ftp protocols. Compared to Scruf however, Ufo also has some disadvantages. First, the overheads of the Catcher-based approach under Ufo's modified Andrew Benchmark for local files, required at least 17% and up to 100% more real time to complete than without Ufo. As we will show in Chapter 6, Scruf's overheads tend to be substantially lower for a comparable test. Second, Scruf can be used to implement several kinds of file system extensions, whereas only the Global File System is described with Ufo. Finally, Ufo runs entirely at the user-level, does not have access to internal kernel structures, and some functions are not able to be implemented. Also, modification by individual users requires understanding of the whole system.

AVFS [2] is a virtual file system built on FUSE, which is an open-source user-level

Design Option	Brief Description	Main Advantages	Current Disadvantages
Microkernel and File Server(s)	File system and server is **external** to the microkernel.	Comprehensive ability to implement file system, including new physical layouts. Can support multiple file systems and servers.	Relative complexity. Non-trivial to develop new servers; not generally intended for regular users. Requires special permissions and expertise to develop and configure. Generally, not an option with monolithic kernels. Not a commonly available option for monolithic kernels, except for user-level file systems as described below.
Monolithic Kernel File Systems	**Internal** kernel interface to a variety of file systems. Can include dynamic, loadable modules.	Basic architecture is widely used (VFS). Can support a wide range of file systems. *Stackable file system* variation increases flexibility.	Not generally intended for regular users Requires special permissions to configure. Non-trivial to develop new modules, building blocks, and file system stacks. Complexity is inside kernel.
User-level File Systems	Redirection of VFS operations to a user-level daemon.	Relative simplicity.	Generally, all VFS functions have to be re-directed, even if only a subset require modification. Not generally intended for regular users; requires special permissions to configure.
Trigger Scripts	Higher levels of abstraction: VFS operations selectively extended to an **external** daemon running user-level scripts. Still uses local file systems (e.g., Linux's `ext2`) for storage on disk.	Simplicity and flexibility. Per-user, per-directory, and per-operation flexibility in customization. Complexity is outside the kernel. Overheads proportional to complexity of new functionality. Support for a variety of new FS-functionality. Does not require superuser permissions to customize. Can be implemented for many OSes and local file systems.	Cannot redefine all file system functionality (e.g., physical block layout). For example, cannot change the on-disk format of the file system. Overheads are greater than with VFS, but comparable to microkernels, when properly optimized.

Table 3.1: Example Design Options for Extensible File Systems

file system that moves almost all file system operation to the user-level. It allows users to open zipped and tarred files, as well as view remote systems. AVFS is an extension built on top of FUSE, so it is more relevant to compare Scruf to FUSE itself as they are frameworks on which extensions can be built. FUSE is a user-level file system shell that allows a user to implement their own file operations (e.g., open, read). FUSE file systems are "virtual" in that a FUSE partition does not contain actual files, but is a pathway to access files on the existing disk-based file systems on the machine. The FUSE file operations can be implemented to read zipped files and access tarred files as if they were directories. FUSE works similarly to Scruf in that it uses the `/proc` file system to communicate between user and kernel level, and has a module portion which must be loaded into the kernel. Scruf has some advantages in that extensions are more easily implemented in Scruf than in FUSE. FUSE is written in C and all file operations must be implemented by the user, whereas Scruf requires a user only to extend the operations they need to. Moreover, while FUSE moves all of the file system operation up to the user-level, Scruf interacts with the user-level only for those operations which require it, by filtering at the kernel level. FUSE is an open-source project and has not been presented in any research literature.

Perhaps the project that has the most in common with Scruf is Watchdogs [3] which was implemented on a 4.3BSD Unix platform. Watchdogs allows a user to attach a process called a *watchdog* to a file and whenever file operations occur on that file, the watchdog is invoked to handle the operation. Like Scruf, the watchdog can modify operations or defer operation to the kernel. Watchdogs contains greater functionality than the current implementation of Scruf. For example, on a read operation a watchdog can redirect the read to a buffer of its own choosing. Watchdogs can be attached to either individual files or to whole directories. Watchdogs also contains an inheritance mechanism. A file can inherit a watchdog that is attached to its parent directory, but a file cannot inherit a watchdog attached to another file. Watchdogs has some disadvantages as well. First, it has a more involved installation in that to *bind* a watchdog to a file requires the user to explicitly start each watchdog process, whereas with Scruf the mere presence of the trigger script in a directory associates it with file operations performed in that directory. Second, where Scruf has a single user-level daemon to mediate between kernel and user-level, Watchdogs require that each watchdog process handle its own communication with the kernel, and requires they be written in C to link the required system calls to bind watchdogs to files, adding to the (necessary) complexity of the watchdogs. Third, each watchdog must also handle all file operations, unlike Scruf, where each operation can be encapsulated within a single trigger

script or left out entirely if the user chooses. Moreover, Watchdogs imposes a limit that says only one watchdog can be associated with each file (however, the same watchdog can be associated with several files), while in Scruf all the trigger scripts apply to any file. Finally, being implemented on a BSD platform, adding Watchdogs' kernel code would require re-compilation and reinstallation of the kernel, whereas the changes to Scruf are localized to a module to take advantage of Linux kernel loadable modules for easier installation.

3.3 Concluding Remarks

This chapter summarizes some other projects exploring extensible file systems that are similar to Scruf. While the concept of extensible file systems is not new, we feel that Scruf introduces a fresh approach that is simple, yet effective and worthy of being explored.

We have outlined the goals of extensible file systems, at a basic level, and described our approach using trigger scripts in the Scruf framework. The next chapter will introduce the five file system extensions we have implemented using Scruf.

Chapter 4

Trigger Scripts for Extensible File Systems

4.1 Introduction

To this point, the Scruf framework has been described at a high-level and compared to similar research projects for extensible file systems. Before describing the implementation details of the framework, we will describe the extensions that have been implemented using the Scruf framework and trigger scripts, and how they work.

4.2 Trigger Script Operation

Trigger scripts can do anything that other processes can do, including compute, maintain state, access the network, and interact with other processes. However, in terms of how trigger scripts interact with Scruf, there are (currently) two main techniques, which can be combined:

1. **Interposed action.** Trigger scripts are called before the associated VFS file operation is allowed to complete. The action of the trigger script is like a hook function in the execution pathway, that may perform computations, but does not change the file operation being performed. For example, a `cvs commit` within an on-open trigger script is an interposed action. After the trigger script is finished, the open operation proceeds without any changes.

2. **Redirection action.** An on-open trigger script can return a pathname to Scruf-VFS that is different from the pathname passed into the script. In essence, although the open file operation was passed one pathname, it is logically redirected to a different pathname. For example, an on-open script can uncompress a file (i.e., an interposed

Name	Brief Description	Implementation Details	Comment
Scruf-Trace	An instrumented file system that generates a trace of file operations.	Fast, benchmarked implementation written in C. Original version written as shell script. Has on-open, on-read, on-write, on-lseek, and on-release (i.e., close) trigger scripts.	Useful for system debugging and understanding. Good stress test of overheads.
Scruf-Compress	A compressed file system; files are uncompressed on demand. A popular idea for file systems.	Written in Python. Key functionality in on-open and on-release trigger scripts to `bunzip2` and `bzip2` files, respectively. on-write updates files.	Useful for archives of large files, such as logs, Postscript documents, and `tar` files. Useful enough that applications like Ghostview and `tar` explicitly handle compressed files. Scruf-Compress works with other, unmodified programs.
Scruf-Crypt	An encrypted file system; files are decrypted on demand. A popular idea for file systems.	Scripts written in Python. Has on-open and on-release trigger scripts to decrypt and encrypt files, respectively, and on-write for updating. Uses GNU Privacy Guard (GPG) with a symmetric cipher.	Current implementation is a proof-of-concept. Future implementation will keep decrypted file contents in non-swappable memory.
Scruf-CVS	A version-controlled file system. User can rollback to or recover previous versions of files. A popular idea for file systems.	On-open script and recovery tools written in Python.	In contrast to Scruf-Compress, version-controlled file system is best suited to frequently-changing, smaller files. Similar in concept to the Elephant File System [23] and much simpler to implement, but not as full-featured.
Scruf-Trellis	A remote data access file system; useful for grid computing and metacomputing.	Scripts written in Python. Uses on-release and path trigger script feature of Scruf. Calls Secure Copy (`scp`) for data movement.	In future, will use Trellis File System Library to support caching, prefetching, and sparse access.

Table 4.1: Example Scruf File Systems Using Trigger Files

action) and redirect the open VFS to open the uncompressed copy of the file, thus transparently implementing a compressed file system.

Trigger scripts are passed information via command-line arguments, as well as through standard input. The arguments and IPC messages passed by the daemon give details about the file operation which has occurred. The data which is passed includes the name of file, the type of operation, the size of the file and the time of the operation. If the amount and position values are not relevant to a particular file system operation, such as open, null values are passed. The second command-line argument is always the name of the file being acted on (i.e., `argv[1]`). An example of what is passed as the third command-line argument (i.e., `argv[2]`), and redundantly via standard input, is:

```
/data/trigger/logger/scruf/camfs_impl.o,11:46:24.513,read,0,4096,29662
```

This string corresponds to a read operation which occurred on the file `camfs_impl.o`, the read began at an offset of 0, and read 4096 bytes. The size of the file at the time of the read was 29662 bytes. As this was a read, this string would have been passed to the `.on.read` script in the directory `/data/trigger/logger/scruf/`. Upon receiving this information, the script can interpret the data and return a code to the kernel, indicating an interposed or redirection action. The API will be discussed in detail later, in Section 5.7.1.

4.3 File System Extensions in Scruf

We now consider a number of different extended file systems implemented using trigger scripts (these extensions are summarized in Table 4.1). Some of the implemented ideas have been sought for a considerable period of time. What is noteworthy is how simple it is to implement the basic functionality, once the Scruf framework is available. The core functionality of the trigger scripts can sometimes be embodied in one line, although more robust implementations can take up to 40 lines of code. Although the systems seem trivial to implement in retrospect, without Scruf, they likely would have required substantial, purpose-specific changes to the OS.

4.3.1 Scruf-Trace: An Instrumented File System

Tracing I/O operations and logging the events can be an important component of program and system understanding. We have implemented a simple trace facility and the tools to post-process the log files. The original Scruf-Trace trigger scripts were written as one-line shell scripts. Recall that `argv[2]` contains the information-rich string discussed above. Therefore, all that is required is:

```
echo $2 >> /usr/scovil3/logfile
```

The above code is simply placed in a plain text file named `.on.open`. Since all five of the trigger scripts do exactly the same thing, only one file, `.on.open`, need be an actual file; the other trigger scripts can be symbolic links to `.on.open`. A directory with this setup looks as follows:

```
[cam@sunset ~/test]$ ls -al
total 16
drwx--S---    2 cam      grad         4096 Aug 15 14:48 .
drwxr-sr-x   69 cam      grad         4096 Aug 15 14:48 ..
lrwxrwxrwx    1 cam      grad            8 Aug 15 14:47 .on.lseek -> .on.open
-rw-------    1 cam      grad           43 Aug 15 14:46 .on.open
lrwxrwxrwx    1 cam      grad            8 Aug 15 14:47 .on.read -> .on.open
```

```
lrwxrwxrwx    1 cam      grad            8 Aug 15 14:47 .on.release -> .on.open
lrwxrwxrwx    1 cam      grad            8 Aug 15 14:47 .on.write -> .on.open
-rw-------    1 cam      grad          951 Aug 15 14:48 anyfile
[cam@sunset ~/test]$ cat .on.open
#!/bin/sh

echo $2 >> /usr/scovil3/logfile
[cam@sunset ~/test]$
```

When the file operations are complete, the file `/usr/scovil3/logfile` contains all the file accesses made to files in the directory containing the trigger scripts. The user could add code to the trigger scripts to filter for specific files. To improve performance, we have since implemented a more sophisticated version of the trigger scripts using C. The new Scruf scripts are asynchronous, they do not write to the log file every time they receive a message; instead they buffer them and write them to the logfile in batches of 1500. The script used is included in Appendix A.

To demonstrate the usefulness of such information, we used Scruf-Trace to log and analyze file accesses during a compilation with the GNU C compiler, `gcc`.

Consider the following scenario: Someone is asked to maintain or modify a large, existing software system. Even with the best in-line and off-line documentation for the system, it is very valuable if the programmer can trace, execute, and observe the system. Although many different interactions and sub-systems are challenging to learn for the uninitiated programmer, we will focus on file system activity. Figure 4.1 shows a post-processed, graphical representation of the trace of the file system activity for a complete compile and link of two source files. The post processing utility used is part of the Carina debugging tool [19]. The graphs shown are automatically generated by this utility which is written in Python and Ploticus, based on a log file. Therefore, the log file is generated on-line, and the graph is automatically generated off-line.

Since Scruf-Trace works on a per-directory basis, only the activity within directories that contain the scripts is logged. This granularity is welcome if only part of a large build needs to be traced. However, the scripts will be inherited into lower directories. Turning off the trace facility is as simple as re-naming or removing the trigger scripts.

On the time line (Figure 4.1, top), the various horizontal lines represent different files within the directory that is being traced. The X-axis represents real time. For each file, all read, write, open, and close (actually, it is a release) events are logged. A horizontal bar begins when the file is opened; it ends when the file is closed. Each time that the file is accessed via a read, a vertical tick is inserted in the graph in the top-most horizontal bar for that file. The location of the tick represents the time of the access. Each time that the file

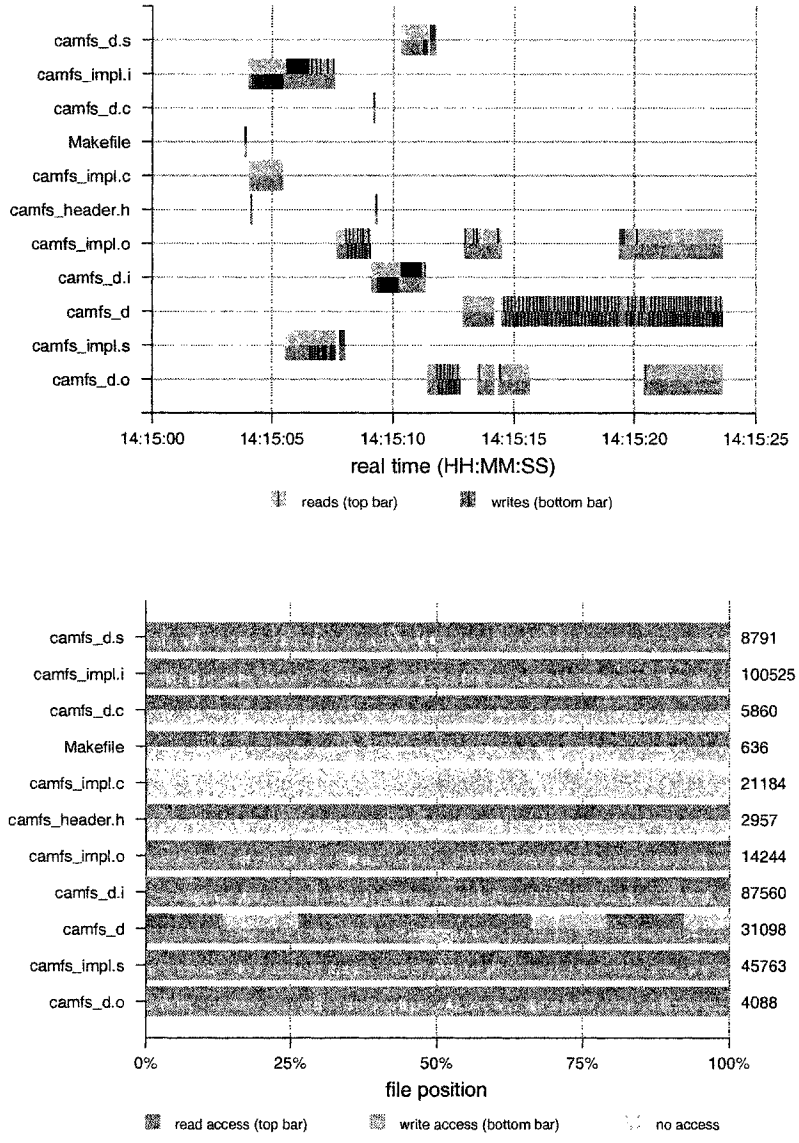

Figure 4.1: Scruf-Trace: Understanding a Compilation: Time Line (top) and File Coverage (bottom)

is accessed via a write, a vertical tick is inserted in the graph in the bottom-most horizontal bar for that file. For example, the file `camfs_impl.i` is accessed from time 14:15:04 onwards. There is a burst of write accesses near the beginning of the file's lifetime, followed by a number of reads through the rest of its lifetime as `gcc` uses it to create the object file `camfs_impl.o`, which begins at approximately 14:15:07. This graph also indicates the interleaving of reads and writes that occur over the first lifetime of object files, and tells us that object files are opened, read and closed in subsequent lifetimes, instead of being left open.

During the compile process, the `Makefile` is accessed first (as expected), read and immediately closed. Then the various source files (i.e., `.c` and `.h` files), `.i` intermediate files, `.s` assembly files, and finally `.o` object files, are accessed. Notice the file `camfs_impl.o` has three bars along its X-axis. This indicates the file has three lifetimes, meaning it is opened and closed three times. It is also interesting that `camfs_impl.c` is not read or written. This is because in `gcc`, C files of over a certain size are not read using the standard `read()` system call; instead they are memory mapped (mmapped).

The file coverage graph (Figure 4.1, bottom) summarizes all of the files that have been accessed during the trace (left Y-axis), the lengths of the files (right Y-axis), and a normalized representation of the regions of the file that are touched. The accesses are broken down by reads versus writes, and parts of the file that are not accessed at all are also indicated. This graph is also generated automatically, off-line, by the Carina post-processing utility working with the trace files generated on-line by Scruf-Trace. The source file `camfs_impl.c` shows no reading or writing, this is because it is mmapped and reading and writing mmapped files does not require system calls.

4.3.2 Scruf-Compress: A Compression File System

Another function that users have long desired in a file system is automatic compression and inflation of large files. Scruf-Compress allows unmodified executables to transparently handle compressed and uncompressed files. If a file is compressed, then opening the file automatically uncompresses the file before returning to the application. When the file is closed, it is compressed again with any changes which may have occurred. Although disk storage is cheap, when transferring data over a network, compression is a necessity to save time as feature-rich editors create documents of increasing size. This need is so apparent that many popular applications have built-in compression and inflation to handle such files. Examples of these applications include vim, tar, and Ghostview, among others. However,

there is still a large body of applications which do not have this capability. It is these applications for which we anticipate Scruf-Compress will provide a simple and effective solution that users could have up and running in minutes.

Scruf-Compress involves three VFS trigger scripts, `.on.open`, `.on.write` and `.on.release`. At the heart of `.on.open` is another one-line command: `bunzip2 -c $1`. However, our actual script is 37 lines of Python code, including some error checking, debugging, and pathname parsing code. Scruf-Compress is the first example to demonstrate an `.on.open` script that will use a redirection action to change the file being opened in the VFS. When an `.on.open` is message is passed to the script, the script first determines if the file is compressed; if it is not, the script's action is an interposed action with no effect and the VFS operation continues unchanged. However, if the file is compressed, the script will uncompress it to a temporary filename and redirect the VFS to open the uncompressed temporary file. The task of the `.on.write` script is to mark files which are updated. This information is used when a release occurs. When `.on.release` is passed a message about a release which has occurred, it checks if it is a Scruf-Compress temporary file; if not, it does nothing. If it is a temporary file, `.on.release` then determines whether the original compressed file needs to be updated. If the temporary file has not been written, then no update is needed and `.on.release` simply removes the temporary file. If a write has occurred, `.on.release` removes the original compressed file and compresses the temporary file which contains the changes to the filename of the original compressed file (now deleted). Once the file is compressed, the temporary file is deleted. In the end all that is left is a compressed file, either the original—if it was not modified—or the updated file. Our versions of `.on.open`, `.on.write` and `.on.release` are 37, 26 and 33 lines of Python code, respectively. Therefore, with about 100 lines of Python, Scruf-Compress is able to allow any unmodified binary application to access `bzip2` compressed files as if they were not compressed. All three scripts are included in Appendix B.

Scruf-Compress does not require that all files be compressed; it works transparently with both compressed and regular files. It is also worth stating that this functionality does not have to be implemented system-wide but, as with any Scruf extension, can be applied to individual directories or across an entire directory tree.

4.3.3 Scruf-Crypt: An Encrypted File System

A user may not want to indefinitely leave sensitive files on the file system in decrypted format. When the file is not being actively accessed, the user wants to encrypt the file.

Thus, we developed a simple encrypted file system called Scruf-Crypt. Admittedly, Scruf-Crypt does not handle all possible security attacks on the file system, but it does reduce the window of vulnerability for cleartext files.

Scruf-Crypt is similar to Scruf-Compress in structure. Trigger scripts `.on.open`, `.on.write` and `.on.release` are required, as with Scruf-Compress. At the heart of `.on.open` is a call to decrypt the file, and `.on.write` and `.on.release` work as before to mark and update files that have been modified. Currently, our system uses the GNU privacy guard, or GPG [10] with a symmetric cipher. While this setup provides a simple form of security, by adding a more complex authorization and encryption model, and by using non-swappable, secure memory, the level of security could easily be increased. The trigger scripts for our implementation of Scruf-Crypt are included in Appendix C.

4.3.4 Scruf-CVS: A Version-Controlled File System

As already discussed in our motivating example in Section 2.1, only one trigger script is needed. Specifically, we implemented an `.on.open` trigger script in 40 lines of Python. The basic strategy assumes that the user has already set up a CVS repository and the CWD is within a checked-out copy of a directory in the repository.

The `.on.open` script makes a copy of the file being opened and, in the background, commits the copy to CVS. There is (almost) no perceivable delay between invoking, say, `vi` on a file within a Scruf-CVS file system and being able to edit the file. Recovering previous versions of a file can be done using normal CVS commands or a simple `scrufcvsrecover` tool that we have written. The `.on.open` script for Scruf-CVS is included in Appendix D.

4.3.5 Scruf-Trellis: A Remote Data Access File System

In the previous four examples, we have used VFS trigger scripts. We now discuss a system, Scruf-Trellis, that uses path trigger scripts to implement remote data access capability. With recent, renewed interest in grid computing and metacomputing, data grids and remote data access functionality is an active area of research.

Previously, we developed a pure user-level library, called Trellis, that supported remote data access [24]. That library has more features than Scruf-Trellis, but our library-based approach also required source code modifications. With Scruf-Trellis, we are trying to support unmodified applications. Over time, Scruf-Trellis will evolve to have the same functionality as the user-level library. In the meantime, it is a meaningful test application

of Scruf and path trigger scripts.

Consider a scenario where a researcher has access to different computer systems at two (or more) different computer centers. Unfortunately, the two systems (call them A and B) are in different administrative domains and there is no distributed file system to allow easy access to remote data. What if the user is logged onto A and their data files are on B? Of course, the user could explicitly copy the data files from B to A using `ftp` or `scp`. However, what if the user wanted the file system to automatically copy files on demand, even though there is no shared file system? If such a remote data access capability was available, unmodified applications could access data, regardless of where they are stored, with something like:

```
cat trellis:scp:black.cs.ualberta.ca:myfile
```

When read as one line, the above pathname means that file `myfile` is currently stored on machine `black.cs.ualberta.ca` and can be accessed using the `scp` application. We assume all authentication and permission details have been taken care of in setting up public and private keys for secure copying. Again, the user could have called `scp` first and then called `cat`. Ideally, we want `cat` to automatically do the `scp` and we do not want to have to change and recompile `cat`.

We have implemented a path trigger script called `trellis`. When the name of a path trigger script appears in the pathname, as shown above, an open operation forces the script to be called with the pathname as a parameter. Specifically, before `cat`'s call to `open()` is allowed to finish, Scruf calls the script `trellis` and passes its path string `scp:black.cs.ualberta.ca:myfile` as a command-line parameter. By design, the script `trellis` parses the command-line argument, and uses `scp` to transfer `myfile` onto the local machine. In essence, this is whole file caching. Using the technique of redirection, script `trellis` then directs the VFS to open up the local copy of the file by returning a new pathname to the file.

While the path shown above may seem overly detailed and inconvenient, a simplification is possible. If a user only wished to access files on the `black.cs.ualberta.ca` machine, that information—as well as the connection method—could be moved into the `trellis` script, and the command line could become the much more convenient command:

```
cat trellis:myfile
```

Name	.on.open	.on.read	.on.lseek	.on.write	.on.release
Scruf-Trace	✓ I	✓	✓	✓	✓
Scruf-CVS	✓ I				
Scruf-Crypt	✓ R			✓	✓
Scruf-Compress	✓ R			✓	✓

Table 4.2: Summary of VFS Trigger Scripts Required for Extensions. `.on.open` scripts perform either Interposed (`I`) or Redirection (`R`) Action.

Using a VFS trigger script for `.on.release`, it is possible to copy the file back to the original computer system when it is closed, as per open-to-close writeback semantics for whole file caching. Enough state information (e.g., where the file came from) can be saved in a shadow file (e.g., `.myfile.metainfo`) by the `trellis` script to allow `.on.release` to work.

The combination of the path trigger script `trellis` and the VFS trigger script `.on.release` allows Scruf-Trellis to implement a whole file caching version of remote data access. Unmodified applications like `cat` can then access files anywhere on the Internet that can be named using Secure Copy's convention. The trellis script is included in Appendix E.

4.4 Discussion

The five file system extensions discussed above demonstrate how trigger scripts, both VFS and path, can be used to implement useful extensions quickly and relatively easily. Partially running at user-level allows programmers to create useful utilities without requiring any kernel-level programming, recompilation, or reinstallation. These extensions also demonstrate that minimal setup is required for trigger scripts. Table 4.2 summarizes the VFS scripts required for each extension built with them, however Scruf-Trellis is excluded as it is implemented with path scripts. Each column contains a checkmark if that script is required for that extension. The `.on.open` column contains an `I` or an `R` as well, indicating if the action by the on-open script is an Interposed action (`I`) or a Redirection action (`R`). These extensions are just a sampling of the many possible extensions that could be implemented using Scruf and trigger scripts.

Chapter 5

The Scruf Framework

5.1 Introduction

In the previous chapters, the utility of Scruf as an extensibility framework has been discussed, as well as the extensions that are possible using Scruf. It is now fitting to describe in detail the implementation of the Scruf framework. The Scruf framework has been implemented with the following three concepts in mind:

1. **Scruf-VFS must be a loadable module.** The functionality that must reside in the kernel to implement user-level extensions must be localized. The functionality must be restricted to a loadable module so the kernel does not have to be recompiled or reinstalled.

2. **No changes to the disk layout.** While Scruf is an extensible file system, the core system must operate as a standard disk-based file system. As such, we chose `ext2` as the basis for Scruf as it is one of the most commonly used Linux file systems.

3. **Move as much of the Scruf framework's complexity as possible to the user-level.** The kernel-level modifications should be made as small as possible. The function of these modifications is to communicate as efficiently as possible with the user-level. All functionality for handling trigger scripts should be at the user-level.

In our current implementation, we are working with Linux 2.4.18, a monolithic kernel, and the internal VFS interface using loadable modules. Since Scruf is built on `ext2`, Scruf disk partitions share the same physical layout as Linux's `ext2` file system. In fact, a simple change in the mount table allows the same partition to be mounted as either a Scruf or an `ext2` file system. Scruf overheads are only incurred if trigger scripts are used and only by the users and in the directories that have or inherit the scripts. In Section 6.4, we show that

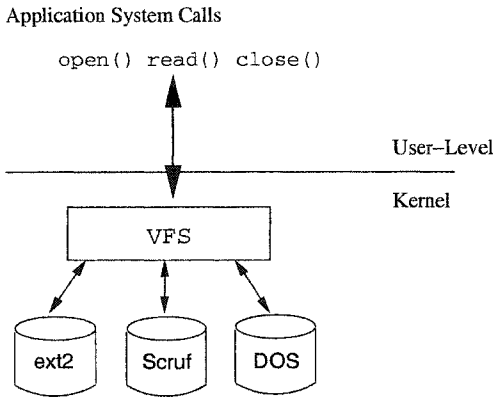

Figure 5.1: Linux's Virtual File System (VFS)

if a directory does not contain any trigger scripts, the overheads introduced by our kernel modifications are less than 1% of the total real time. There is almost no inherent overhead for mounting an `ext2` partition as a Scruf file system.

The discussion of the implementation of the Scruf framework will begin with a basic discussion of operating system pathways and how file systems work, as background information.

5.2 File System Basics

Before describing the details of the Scruf framework's implementation, it is important that the reader have an understanding of some file system basics. First, Linux's Virtual File System and mount points will be described, followed by some file system basics such as how files are looked up, and file system structures. Finally, loadable modules will be described. This will not be a full description of how file systems work, but only to a level that will facilitate understanding of the work presented here. For a more detailed description of file systems in Linux, see Bovet and Cesati [5].

5.2.1 Linux's Virtual File System

File systems in Linux use the Virtual File System or VFS interface, illustrated in Figure 5.1. Other Unix-based operating systems, such as Solaris and BSD, also have VFS interfaces. The VFS is the layer in the kernel that all file system calls pass through. The abstraction provided by VFS gives the kernel a consistent view of different file systems despite the often striking differences in their structures and operation. When file systems are mounted

```
struct file_operations ext2_file_operations = {
    lseek:          generic_file_llseek,
    read:           generic_file_read,
    write:          generic_file_write,
    ioctl:          ext2_ioctl,
    mmap:           generic_file_mmap,
    open:           generic_file_open,
    release:        ext2_release_file,
    fsync:          ext2_sync_file,
}
```

Figure 5.2: Structure of VFS file operations table

in the kernel, they are controlled by the VFS through structures of function pointers, similar to virtual function tables in C++. Each file system has its own function pointer tables. Linux's VFS interface defines tables for directory operations, address operations, inode operations, and, most relevant to Scruf, file operations (i.e., `ext2_file_operations` in `fs/ext2/file.c`, shown in Figure 5.2). This internal interface of the VFS allows file systems to be installed without recompiling or reinstalling the kernel, since the tables can be set up while the kernel is running. The code describing a file system can then be contained in a loadable module, which will be described shortly.

It is important to understand that these file operations are distinct from the system calls which share their name. User-level applications make system calls, such as `open()`, which eventually result in the VFS invoking the corresponding file operation. An `open()` system call will cause the open file operation to be invoked as the system call moves through the kernel, but the two are distinct from each other. Briefly stated, system calls exist in the user-level and file operations exist in the kernel.

5.2.2 Mount Points

Another concept that is important for the reader to understand is how directory hierarchies relate to hard disk partitions. The concept that relates the two is called a *mount point.* To explain, consider Figure 5.3. On the left is a directory hierarchy, on the right is a hard disk made up of four partitions, `/dev/sda1` through `/dev/sdd1`. A partition is added to the hierarchy by *mounting* it. To mount a partition, three important parameters are required: a directory name, a partition, and a file system type. An example of the command to mount a file system is,

```
[root@scovil /]$ mount -t ext2 /dev/sdb1 /tmp
```

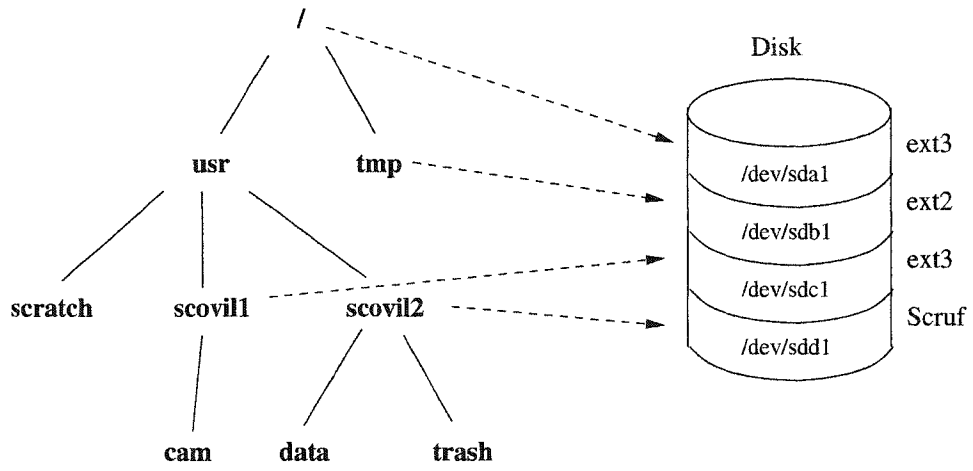

Figure 5.3: An Example of Mounted Partitions

This command mounts the partition `/dev/sdb1` to the directory `/tmp` and tells the VFS it is an `ext2` partition. From then on, all directories and files under `/tmp` in the hierarchy will be stored on `/dev/sdb1` and accessed using `ext2` operations tables.

The only partition that is absolutely required to be mounted is the root directory, `/`, which is mounted at boot time. There is a file that summarizes all mount points on a system, `/etc/mtab`, and it is simply a text file. The `/etc/mtab` file for the system shown in Figure 5.3 is:

```
[cam@scovil scruf]$ cat /etc/mtab
/dev/sda1 / ext3 rw 0 0
none /proc proc rw 0 0
/dev/sdb1 /tmp ext2 rw 0 0
/dev/sdc1 /usr/scovil1 ext3 rw 0 0
/dev/sdd1 /usr/scovil2 scruf rw 0 0
[cam@scovil scruf]$
```

Each row corresponds to a mounted partition. For now, ignore the `/proc` entry; it will be discussed later. In the table, the first three fields of each row are the important ones and their format is:

```
<device> <mount point> <file system type>.
```

For example, the fourth line explains that the `/dev/sdc1` disk partition is mounted on the directory `/usr/scovil1` and the format of `/dev/sdc1` is the `ext3` file system type. The fifth line explains that the `/dev/sdc1` partition is mounted as file system type Scruf on the directory `/usr/scovil2`.

To determine which partition any directory is mounted on, simply trace up the hierarchy until a mount point is reached, and that is the partition. For example, in Figure 5.3, `/usr/scovill/cam` is mounted under `/usr/scovill`, which corresponds to `/dev/sdc1`. `/usr/scratch` is mounted under `/`, which corresponds to `/dev/sda1`, the root file system. If a partition is not mounted, it is not in the hierarchy, and therefore is not listed in `/etc/mtab` and is not visible to any users.

When an application opens a file (call the file A) the VFS must determine what mount point A is on, and which file system type that mount point is. Once the VFS determines the file system which that file is mounted under, it invokes several functions to look up and open A. The VFS does not know the low-level details of how a particular file system stores its files; it does not know how or where A is stored, but only interacts with A's file system using the file, directory, and address operation tables for that file system.

5.2.3 VFS Data Structures

To the average user, a file is conceptually simple: it is like a piece of paper in a filing cabinet. While this is true of the data contained in the file, within the kernel files are far more complex structures. Along with a file is stored the owner's ID, permissions, modification time, and size, as well a great deal of other information. Here, the different structures used to keep track of files in the VFS will be described. Three important structures will be discussed:

1. **Inodes:** For locating files on disk.
2. **Dentries:** For locating files by name in memory.
3. **File Objects:** For tracking files a process has opened.

On disk, a file is represented by a structure called an *inode*. An inode is created when the file is created, and exists for the duration of that file's lifetime. Users may often think that a unique pathname identifies a file, however, this is not entirely true. Each inode is assigned a unique number—the number of its inode. This number never changes, even if the file is renamed or moved to a different directory; only the name changes (unless the file is moved across partitions). The important thing to remember is that an inode, and its associated inode number, uniquely identify each file.

While files are stored on disk as an inode, when they are brought into memory they are stored as another structure as well, called a *dentry*. The name dentry is an abbreviation of

directory entry. As a file's pathname is looked up, every segment of a pathname is stored as a dentry. A dentry contains nearly every detail about a file, since it must represent a file while it is in use. Each dentry is stored in a hash table. Since fetching inodes from disk and setting up dentries can be expensive, Linux maintains a *dentry cache* which caches the most recently used dentries in memory, to reduce future disk accesses if they are referenced frequently.

A *file object* is a structure that a process uses to keep track of a file it has opened. It contains fields describing access flags, a pointer to the current offset, etc. Since several processes can open the same file, several file objects may in fact point to the same inode (remember the inode is unique!). The file object is a temporary structure to describe a file which is opened. When a file is opened by a process, a file object is created and stored in an array of file objects for that process. The index into this array is a *file descriptor* which enables user applications to keep track of open files.

5.2.4 Pathname Lookup

The lookup of a pathname is the series of steps the file system performs to determine if a file exists; it begins at the mount point. The VFS searches the directory structure of the current directory for the next name in the path, whether it be a file or directory. Using Figure 5.3, assume there is an `open()` call to a file **/usr/scovil2/data/myfile.c**. The VFS begins in **/usr/scovil2**, looks in **scovil2**'s directory structure for a directory **data**, and then searches **data**'s entries for a file **myfile.c**. If the file is found, then the open file operation is called and VFS returns a file descriptor to the application. If any of segment of the pathname does not exist, the `open()` call will return an access error.

5.2.5 Linux Kernel Loadable Modules

Another system design which is part of Linux consists of loadable modules, which are important in making Scruf simple and convenient to install and use. Loadable modules are drivers for services such as secondary file system types and devices. They allow code to be dynamically loaded into the kernel while the kernel is running, without requiring recompilation or reinstallation. Loadable modules provide tremendous flexibility to Linux and allow device drivers and file systems to be added to the kernel quickly and easily without interrupting the system. The only file system which cannot be a loadable module is the file system which the root file system is mounted as. In Figure 5.3, `ext3` cannot be a loadable module, it must be compiled into the kernel. The kernel-resident portion of the Scruf File

System is written as a loadable module, which corresponds to Scruf-VFS in Figure 5.4.

5.3 OS Pathways

To understand how Scruf operates, it is important to understand how file system calls from applications, such as `open()` and `read()`, eventually reach the trigger scripts. This section will give a high-level description of the pathway from applications to trigger scripts, and in the next section, we will give a more detailed example of a file operation actually moving through the pathway.

The following pathway through the OS during the execution of an application is diagrammed in Figure 5.4; the numbers given in parentheses correspond to Figure 5.4. (1) An unmodified, binary application is running at user-level. When a file operation is requested, a normal `open()`, `read()`, `write()`, `lseek()`, or `close()` system call traps into the kernel and, eventually, control moves to the VFS layer (2). If the specific disk partition has been mounted as a Scruf file system (e.g., `/usr/scovil2`), then the VFS and Scruf-VFS (3) may interact to search the file hierarchy to find the file if the file operation is an open; otherwise a file descriptor will be provided as a parameter from the application. Control is then passed to the correct Scruf-VFS file operation (3).

The file operation code informs the daemon of the operation via IPC (4) using the `/proc/scruf` pseudo-file that was created by Scruf at the module's load-time. The daemon (5) is responsible for multiplexing and executing (i.e., `fork()`ing) the trigger script as a non-privileged user process.

The trigger scripts (6) can be written in any language or scripting system (e.g., shell, C, Python, Perl) that runs on Linux, and communicate with the daemon via standard in and standard out. If the script is written according to the persistent script protocol (discussed later in Section 5.8.4), the daemon will keep an open pipe IPC channel between itself and the script's process. Since trigger scripts run as normal, full-featured processes, they are capable of doing anything the user is allowed to do (and no more). Thus, trigger scripts are capable of accessing other file systems, creating their own child processes, using the network services, etc.

Trigger scripts can return data to the kernel via the daemon (7). For example, an `.on.open` script can return a pathname back to Scruf-VFS to perform a redirection action (Section 4.2). Depending on whether the kernel has to wait for a response from the trigger script or not, we refer to the scripts as *synchronous* or *asynchronous*. When the file is actu-

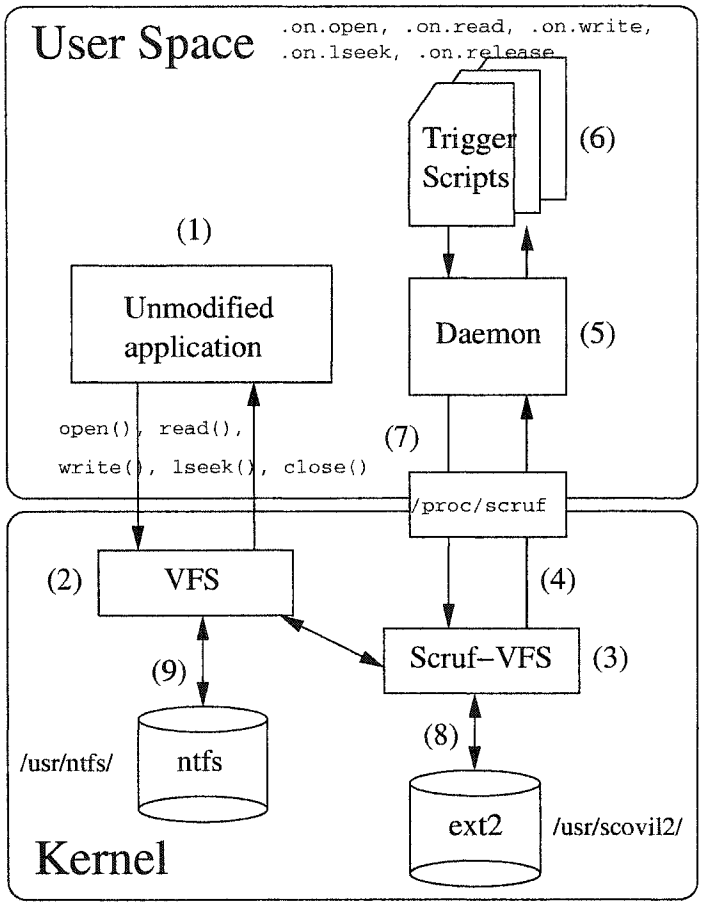

Figure 5.4: Scruf File System and Trigger Scripts

ally transferred from disk (8), the OS is executing unmodified `ext2` code paths. Control is returned to the VFS (2) and back to the application (1).

Note that this pathway to the user-level only applies for partitions mounted with the Scruf file system. Again, referring to Figure 5.4, the pathway from binary executables through the VFS (3) to disk for non-Scruf partitions such as `/usr/ntfs` (9), mounted as type NTFS, is unchanged.

5.4 A More Detailed Example

The section above describes the pathway between applications and trigger scripts. We will now describe an actual operation as it moves through the pathway, again using Figure 5.4.

For this example, we will describe an `open()` system call from an application (1) to open a file called `/usr/scovil2/cam/myfile.txt`. The pathname and flags given

in the `open()` call are passed into the kernel and on to the VFS (2). VFS will check the file system type of the partition, `/usr/scovil2`, and having determined it is a Scruf file system, it then looks up the file using Scruf-VFS's operations tables. The VFS looks up the file by examining each segment of the path, directory by directory, until it reaches the filename. During the lookup, the VFS (2) is invoking Scruf-VFS (3) directory operations using Scruf-VFS's directory operations table to look up each directory in the pathname.

When the VFS determines that the file exists, it creates a file object and passes it to the open file operation in the Scruf-VFS file operations table. The open file operation is invoked and control moves to Scruf-VFS (3). The open function must now communicate with the user-level daemon. It does so by sending a message through `/proc/scruf` (4). The daemon (5) reads the message which contains the full file pathname, the type of operation (an open in this case), the file size, time of the operation, etc. From this information the daemon proceeds to invoke the `.on.open` script in the directory `/usr/scovil2/cam`. It invokes the `.on.open` script (6) and sends the file operation information to the script. The script runs, analyzes the data it receives (the filename, size, time, etc.), may take action, and returns an API code (actual API codes will be described later in Section 5.7.1). Control then returns to the daemon. The daemon returns the API code to Scruf-VFS, again using `/proc/scruf` (7). The user-level daemon is only a multiplexer and does not modify the operations or interpret them beyond what is needed to determine which script to run. When the API code is returned, we are again back in Scruf-VFS's open file operation (3). Only now, after the trigger script is run, does any disk action take place. The API return code may indicate an interposed action, in which case, the open file operation continues without modification; the `/usr/scovil2/cam/myfile.txt` is opened, control returns to VFS (2) and a file descriptor is returned by the `open()` call in the application (1). However, the API return code may tell the open file operation to perform a redirection action to open a different file instead, for example `/usr/scovil2/cam/anotherfile.txt`. In this case, the open file operation looks up and retrieves the new file object and returns a file descriptor for that file to the application (1).

The application itself (1) is unaware to the extra work Scruf does versus a conventional file system. The application makes an `open()` call, and waits for a return value and file descriptor if the lookup is successful. It does not know if the file was changed by redirection. The application proceeds with the file descriptor it received back from the `open()` call. If a file does not exist, no trigger script is invoked. The lookup fails during the interaction between (2) and (3), the open file operation is never called and so the `.on.open` trigger

script is not run.

The other file operations' pathways are similar to an open operation, except the lookup is not necessary since all other file system calls require a file descriptor as a parameter, which is stored by the application, and so the file path is not used.

5.5 Scruf-VFS: The Scruf Kernel Module

Now that we have presented the basics of file system operations and the OS pathways Scruf uses, we will begin describing the implementation details of the Scruf framework. First, the implementation at the kernel level will be discussed. This section describes what we have referred to as Scruf-VFS, which is a Linux kernel-loadable module built on the `ext2` file system.

The design of VFS made the Scruf extensions relatively simple, due to the modularity of the file operations and the code in general. We choose `ext2` as our starting point because of its widespread use in the Linux community and we theorize that if our system could be a simple upgrade to `ext2`, then it would be easily and immediately usable on partitions previously mounted as `ext2` without requiring reformatting. The major design goals that remain are to add the necessary data structures and implement mechanisms for communicating with a daemon running at user-level. It is also important that the changes in the kernel be restricted to the loadable module so they do not require recompilation or reinstallation of the kernel. In this section, we describe the changes to `ext2` that were necessary in order to implement the Scruf framework. The data structures that were added are used for both VFS and path scripts. The code added for VFS scripts and the code for path scripts are orthogonal to each other and do not interact.

In total, 1,500 lines of C code are added to the code base for `ext2` to create Scruf-VFS. Of that, 1,300 lines correspond to new, orthogonal functions to manage the `/proc/scruf` IPC channel and filter VFS operations. These auxiliary functions can be cloned into other loadable modules, such as `ext3`. The remaining 200 lines of code are interleaved with the existing `ext2` code and require more expertise to port to a different file system.

The modifications were made to the five file operations that Scruf will allow users to extend: open, read, write, lseek and release. The code added does not change the underlying storage or access to files. Also, no other operations, such as directory or address operations, were changed. For VFS trigger scripts, the code that is added communicates with the user-level daemon to indicate when a file operation has occurred, and waits for a response from

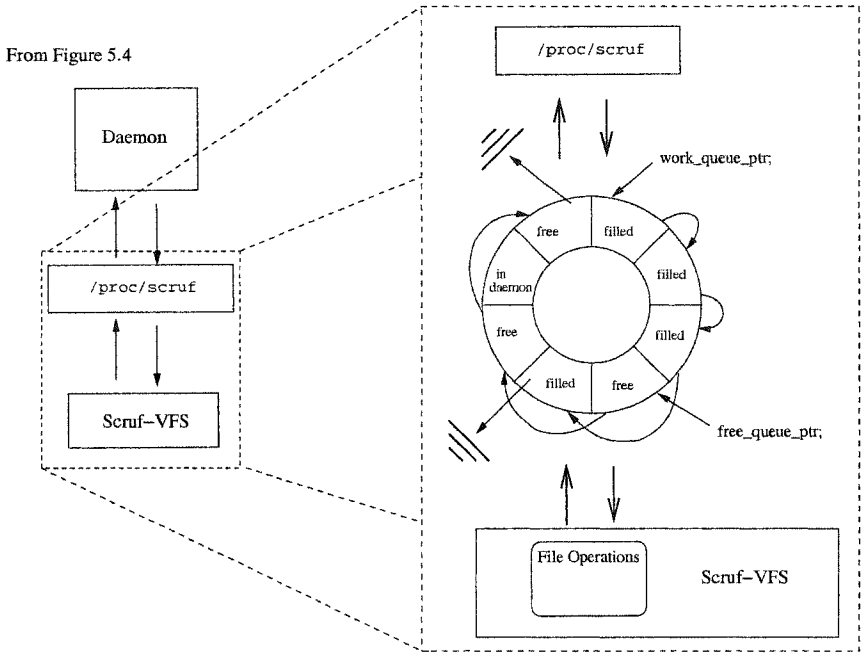

Figure 5.5: The Bounded Buffer used in Scruf-VFS

the associated VFS trigger script, whose response is returned to the file operation in Scruf-VFS. For path scripts, the code communicates with the daemon when a filename contains a colon, and waits for the response from the path script given in the pathname.

5.5.1 Data Structures

The data structures that were added focus on a single goal: to allow the module to efficiently communicate with the user-level daemon. One of the standard structures that is used for communication is a bounded buffer or ring. A ring is simply a structure that contains slots, each slot being single, self-contained packet. All the data pertaining to a particular message is stored within a single packet. A diagram of the bounded buffer, or ring, mechanism shown is in Figure 5.5, which corresponds to (4) in Figure 5.4.

All IPC between Scruf-VFS and the user-level daemon operates in a request-response manner, with Scruf-VFS always the initiator of a request, where the request is for the daemon to run the relevant trigger script. It is therefore intuitive that Scruf uses the same ring for communicating both to and from the user-level daemon. Since every incoming message (response) from the daemon corresponds to a previously sent outgoing message (request), it makes sense to use the same slot in the ring to both send and receive the data. This way, when a file operation occurs and fills a ring slot with a message to be sent to the daemon, the

```
1:  struct scruf_ring_elem {
2:      __u16 id;
3:      struct kernel_to_user_t to_user;
4:      struct user_to_kernel_t from_user;
5:      struct semaphore sema;
6:      struct scruf_ring_elem * next;
7:  };
```

Figure 5.6: Structure of data in a ring element

```
1:  struct kernel_to_user_t {
2:      int ringslot;
3:      int code;
4:      char file[128];
5:      char op[16];
6:      int amount;
7:      int pos;
8:      int filesize;
9:      int blocked;
10:     struct timeval time1;
11:     short hasfilter;
12: };
```

Figure 5.7: Structure of data passed to the daemon

trigger script's response will be returned to the same slot, and the code only needs to synchronize on that slot; this simplifies the synchronization and communication mechanisms that are required. If separate send and receive rings were used, additional control would be needed to coordinate which packet received corresponds to which packet sent and, in turn, which file operation. The current implementation of the ring has 16 slots. The number of slots seems to be bounded by the amount of memory that the ring occupies. If the system tries to initialize more than 16 slots, a segmentation fault occurs.

Each ring element (Figure 5.6) contains an ID number (line 2), structures for sending data to (line 3) and receiving data from (line 4) the daemon, a semaphore for synchronization (line 5), and a pointer for queue management (line 6). The structure which contains the data that is sent to the daemon is shown in Figure 5.7. The data structure which is received from the daemon into the kernel is shown in Figure 5.8.

The structure to manage the ring is shown in Figure 5.9. It contains pointers to manage free (line 2) and work (line 3) queues (Figure 5.5), the ring itself (line 8), and several semaphores. In total, three semaphores are used. The first semaphore, `queue_lock` (line 6) is a mutex which makes sure only one operation modifies the queues at a time. The

```
1:  struct user_to_kernel_t {
2:      int code;
3:      int ringslot;
4:      int returnval;
5:      int filter;
6:      int blocked;
7:      char file[128];
8:  };
```

Figure 5.8: Structure of data passed back to the Scruf module

```
1:  struct scruf_mem {
2:      struct cams_ring_elem * free_head, * free_tail,
3:              * work_head, * work_tail;
4:      int daemon_pid;
5:      struct semaphore daemon_wait;
6:      struct semaphore queue_lock;
7:      struct semaphore buffer_limit;
8:      struct cams_ring_elem ring[RING_SIZE];
9:  };
```

Figure 5.9: Structure of data to manage ring

second, `daemon_wait` (line 5), is a counting semaphore which keeps track of the file operation processes waiting to send a message to the user-level. The third, `buffer_limit` (line 7), also a counting semaphore, is used to ensure that only 16 processes use the ring at any time.

For a file operation to send a message to the daemon, the code waits for an available ring slot. To do this, the file operation code performs a *down* on the `buffer_limit` semaphore. This will block the operation if all the ring slots are full; when the *down* call returns an empty slot is available. The code takes the first element in the free queue and fills in the fields in the `to_user` structure (Figure 5.7). It then adds the ring element to the work queue and performs an *up* on the `daemon_wait` semaphore and blocks on the `sema` field (line 5, Figure 5.6) to wait for its response. It must be noted that there are currently situations where a file operation may block indefinitely, if, for example, the daemon crashes. These will have to be remedied as Scruf moves from a prototype to a production system.

5.5.2 Code Modifications

Since Scruf was designed to be identical to `ext2` at the disk block level, we can keep all the low-level code and extend the high-level functions to work around it. Because of this, the

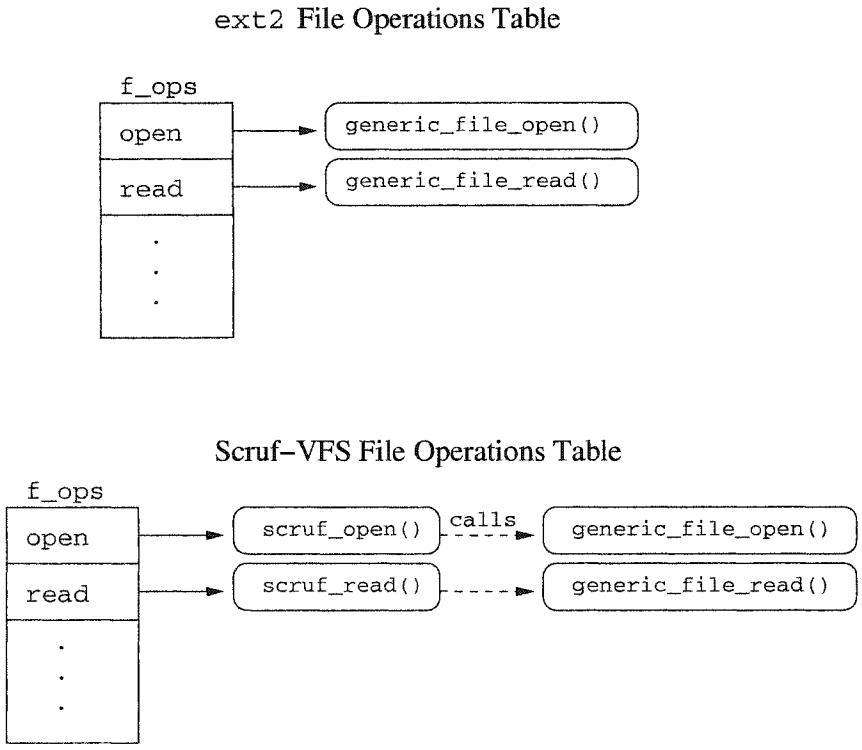

Figure 5.10: ext2 and Scruf-VFS's file operations tables

Scruf-VFS file operations which support VFS trigger scripts are best described as *wrappers* around the original ext2 file operation code. The code added for path trigger scripts does not affect the file operations, but is placed earlier in the OS pathway, during filename lookup and is part of the directory operations table.

Adding Code for VFS Scripts

To support VFS scripts, the new functionality that is required in the kernel is for messages to be sent up to the user-level, using the data structures described in the previous section, whenever a file operation occurs. As mentioned, the added code is a wrapper to first communicate with the user-level daemon and then allow the file operation to continue depending on what is returned from the trigger script. Figure 5.10 shows, intuitively, how the extensions work. Since Scruf uses ext2's underlying storage structure, the Scruf file operations conclude with calls to the original file operations used by ext2. The Scruf-VFS file operations encapsulate the code, which communicates with the user-level daemon before calling the ext2 file operations. This is important since if the underlying operations change, the Scruf code does not need to change. Moreover, this makes Scruf more encapsulated and

more portable to other OSes.

The file operation code for lseek, read, write, and release are shown in Figures 5.11, 5.14, 5.15 and 5.16, respectively. The open file operation code is longer than the others and so is split over Figures 5.12 and 5.13. The new file operations are all similar in their basic structure. First, they communicate with the user-level daemon and conclude with a call to the original `ext2` file operation. For a more detailed explanation, we refer to the code for lseek; the line numbers given refer to Figure 5.11.

The first operations that are performed are a series of three checks:

1. Check that the daemon is running at user-level.
2. Check that this file is not being accessed by the daemon.
3. Check that the file being accessed is not a VFS trigger script.

The `fromDaemon()` function (line 12) performs the first two checks. The first check is important in that the user-level daemon obviously must be running for Scruf to work, or else Scruf-VFS has nothing to communicate with at the user-level. If the daemon is not running, the system functions exactly like an `ext2` partition, as the original `ext2` file operation is still invoked (line 44); only the Scruf code is by-passed.

The second check prevents infinite loops. To demonstrate how an infinite loop might occur, consider the logging extension Scruf-Trace. If a file is read, this will cause a line to be written to the log file. This, in turn, will trigger a message to be sent to the `.on.write` script indicating the log file was written, the log file will again be written to, which will trigger another message, and this cycle will continue. To handle this, we assume that the user cares about only the files accessed by applications, not those that Scruf modifies. To avoid this looping, files accessed by the daemon or by the trigger scripts do not result in calls to trigger scripts. This eliminates messages for operations performed on the scripts (since they are opened, read and released by the daemon when they are invoked) as well as messages for files accessed by the scripts themselves such as log files or temporary files which may created by the scripts.

The third check (line 15) eliminates messages being sent to VFS trigger scripts for file operations performed on the VFS trigger scripts by applications other than the daemon. This eliminates the problem of executing the trigger scripts when they are being edited. In general, this check stops messages being sent whenever a user, or any process, modifies the scripts. Consider when a user edits, for example, the `.on.write` script; this will trigger a

message to the `.on.write` script. However, if the trigger script is being modified, it may be in an incomplete state when it is written, so executing the trigger script at that point is something the user will want to avoid. For this reason, we feel it is appropriate to eliminate messages being sent for operations on trigger scripts. To summarize when we do not want VFS trigger scripts to be run, the second and third checks stop calls to trigger scripts for two cases:

1. Files which are accessed by the daemon (i.e., trigger scripts) and files accessed by the trigger scripts such as log files or temporary files. This prevents infinite loops.

2. Any access to trigger scripts by processes other than the daemon. This means when a user edits a trigger script, no trigger script will be fired. This prevents execution of partially modified trigger scripts.

Once Scruf determines that the daemon is running and the file being accessed should generate a message, the `if` statement beginning at line 14 of Figure 5.11 will evaluate to true and flow will enter its true clause.

It is now worth mentioning that there is a mechanism implemented in Scruf to limit unnecessary IPC for VFS scripts. If a script does not exist in a given directory, it would be preferable for Scruf-VFS not to send further notifications for that operation. This filtering of unnecessary messages will be significant for systems where few trigger scripts are in place. Filters are determined by the user-level daemon, passed down to Scruf-VFS as part of the return structure (line 5, Figure 5.8), stored in a hash table in Scruf-VFS, and used to filter unnecessary IPC.

Using these filters the code must now check that a script exists for this file. To do this, it checks the filter for the directory the file exists in. When the filter information is retrieved (line 22) and it is confirmed that this particular access should be sent to the daemon (lines 24 and 25), the code enters the true clause (lines 26 to 32), the next available ring slot's `kernel_to_user_t` structure (Figure 5.7) is filled, and that ring slot is placed on the work queue. If no filter is found, the code will enter the false clause (lines 34 to 40) and the message will be sent with a flag set (line 11, Fig. 5.7) indicating to the daemon that the kernel needs a filter; a filter will then be returned with the reply. More details on the filtering mechanism are discussed in Section 5.8.3.

Once the message is sent, the file operation will block by performing a *down* on the semaphore in the ring slot (line 31 or 38), and wait to be unblocked when the reply returns from the daemon. If the message requested a filter, the filter is added to the hash table (line

39). The last Scruf-related task is to place the ring slot the file operation used back on to the free queue (line 32 or 40). The function completes with a call to the original `ext2` lseek file operation (line 44), which is actually not part of `ext2`'s implementation, but is a generic lseek function.

The only special case is the release file operation. If a filter is found and a message sent, the release operation will not block since at present there seems to be no reason for a release to block. However, if a filter is requested the operation will block to wait for the filter to be returned. All other file operations always block in the kernel when a message is sent, whether or not a filter is requested.

Redirecting an Open Call

In Figures 5.12 and 5.13, it is clear that the code for a Scruf-VFS open file operation is more complex than that of the other four file operations. The reason for this is that the open operation has support for redirection action, described in Section 4.2, which redirects the open operation from the current file to a different file when an `.on.open` trigger script tells it to do so. Currently, a redirection action can only come from an `.on.open` script.

The open file operation has the same communication and blocking behaviour as the other operations (lines 8 to 28, Figure 5.12). Its added complexity is for handling redirection actions, which it checks for (line 31, Figure 5.12) when the data is returned from the daemon. To redirect the VFS to a new file, Scruf-VFS must close the current file, which involves releasing its inode, looking up the new file's inode, and replacing the current file object's fields with values for the new file. As mentioned earlier, a file object points to a structure that contains several fields describing an open file.

Releasing the old file's inode and dentry is a simple call (line 71, Figure 5.13). The next step in a redirection is to look up the new filename. To do this, the new pathname must be created in a character array and passed along with the dentry of the current directory, which is retrieved from the original file's file object, to the lookup function (line 62, Figure 5.13). The lookup function returns the dentry of the new file, if the file exists. If the file does not exist, then Scruf returns an error to the application. This may be a point of confusion for users. To explain, consider an application opening a file A which exists. If the `.on.open` trigger scripts redirects the open to a file B which does not exist, the `open()` call will return with an error to the application saying file A does not exist. If the new file does exist, it is transplanted into the file object which was passed as a parameter to the open file operation (line 1, Figure 5.12). This is done by replacing the inode and the dentry of the file

```
1  static loff_t camfs_file_lseek(
2      struct file *file,
3      loff_t offset,
4      int origin)
5  {
6      struct inode *inode = file->f_dentry->d_inode;
7      struct inode *p_inode = file->f_dentry->d_parent->d_inode;
8
9      /* SCRUF - Cam */
10     int checkpid;
11
12     checkpid=fromDaemon();
13
14     if ((checkpid==1)&&
15             (strncmp(file->f_dentry->d_name.name,".on.",4)!=0)) {
16
17         unsigned int filter;
18         int found;
19         char thepath[128];
20         int ringID;
21
22         found=getFilter(p_inode->i_ino,&filter);
23
24         if (found) {
25             if (filter&SCRUF_LSEEK_MASK){
26
27                 /* this is for the trigger file */
28                 camfs_getpath(file->f_dentry,thepath);
29                 ringID=write_to_daemon("lseek",thepath,offset,origin,
30                                        inode,CAMFS_BLOCK,found);
31                 down(&cams_mem.ring[ringID].sema);
32                 add_to_free_queue(ringID);
33             }
34         } else {
35             camfs_getpath(file->f_dentry,thepath);
36             ringID=write_to_daemon("lseek",thepath,offset,
37                                    origin,inode,CAMFS_BLOCK,found);
38             down(&cams_mem.ring[ringID].sema);
39             addFilter(p_inode->i_ino,
40                     cams_mem.ring[ringID].from_user.filter);
41             add_to_free_queue(ringID);
42         }
43     }
44
45     return generic_file_llseek(file, offset, origin);
46
47 }
```

Figure 5.11: Scruf lseek operation function

```
1  static int scruf_open_file (struct inode * inode, struct file * filp)
2  {
3      int checkpid;
4      struct inode * p_inode=filp->f_dentry->d_parent->d_inode;
5
6      /* SCRUF - Cam */
7
8      checkpid=fromDaemon();
9
10     if ((checkpid==1)&&
11             (strncmp(filp->f_dentry->d_name.name,".on.",4)!=0)){
12
13         unsigned int filter;
14         int found;
15
16         found=getFilter(p_inode->i_ino,&filter);
17
18         if ((!found)||((found)&&(filter&SCRUF_OPEN_MASK))){
19             char thepath[128];
20             int ringID;
21
22             scruf_getpath(filp->f_dentry,thepath);
23             ringID=write_to_daemon("open",thepath,0,0,
24                                    inode,SCRUF_BLOCK,found);
25
26             down(&cams_mem.ring[ringID].sema);
27
28             if (!found){
29                 addFilter(p_inode->i_ino,
30                             cams_mem.ring[ringID].from_user.filter);
31             }
32
33             /* redirect the file if we are told to do this */
34             if (cams_mem.ring[ringID].from_user.returnval==SCRUF_RELOAD){
35                 struct qstr this;
36                 struct dentry * mydentry;
37                 unsigned long hash;
38                 unsigned char * name;
39                 unsigned int c;
40
41                 /* from path init */
42                 name=(char *)kmalloc(64,GFP_KERNEL);
43                 read_lock(&current->fs->lock);
44                 read_unlock(&current->fs->lock);
45
46                 /* check that 'this' has space allocated*/
47                 strcpy(name,cams_mem.ring[ringID].from_user.file);
48
```

Figure 5.12: Scruf open operation function (a)

```
49                  /* should check if we leak the old 'name' */
50                  this.name = name;
51
52                  c = *(const unsigned char *)name;
53                  hash = init_name_hash();
54                  this.len=0;
55                  do {
56                      name++;
57                      this.len++;
58                      hash = partial_name_hash(c, hash);
59                      c = *(const unsigned char *)name;
60                  } while (c && (c != '/'));
61                  this.hash = end_name_hash(hash);
62
63                  printk("before lookup, parent is %s\n",
64                           filp->f_dentry->d_parent->d_name.name);
65                  mydentry=lookup_hash(&this,filp->f_dentry->d_parent);
66                  if (mydentry->d_inode==NULL) printk("inode is NULL\n");
67                  if (mydentry->d_inode!=NULL){
68
69                      unsigned int error,accmode;
70
71                      if (filp->f_mode & FMODE_WRITE){
72                          put_write_access(inode);
73                      }
74                      dput(filp->f_dentry);
75                      inode=mydentry->d_inode;
76                      filp->f_dentry=mydentry;
77                      accmode=ACC_MODE(filp->f_mode);
78                      error=permission(inode,accmode);
79
80                      if (error) {
81                          add_to_free_queue(ringID);
82                          return error;
83                      }
84                  } else {
85                          /*Are we handling the dentry we replace properly,
86                            if not this could cause a memory leak */
87                      add_to_free_queue(ringID);
88                      return -ENOENT;
89                  }
90              }
91              add_to_free_queue(ringID);
92          }
93      }
94
95      return generic_file_open(filp,inode);
96  }
```

Figure 5.13: Scruf open operation function (b)

```
1  static ssize_t camfs_file_read(struct file * filp, char * buf,
2                                 size_t count, loff_t *ppos){
3
4        /* SCRUF - Cam */
5      struct inode * inode=filp->f_dentry->d_inode;
6      struct inode * p_inode=filp->f_dentry->d_parent->d_inode;
7
8      if (trig_read_on){
9          int ringID, checkpid;
10         char thepath[128];
11
12         checkpid=fromDaemon(); /* don't trigger on files daemon accesses*/
13
14         if ((strncmp(filp->f_dentry->d_name.name,".on.",4)!=0)&&
15             (checkpid==1)){
16
17             unsigned int filter;
18             int found;
19
20             found=getFilter(p_inode->i_ino,&filter);
21
22             if (found){
23                 if (filter&SCRUF_READ_MASK){
24                     camfs_getpath(filp->f_dentry,thepath);
25                     ringID=write_to_daemon("read",thepath,count,*ppos,
26                                            inode,CAMFS_BLOCK,found);
27                     down(&cams_mem.ring[ringID].sema);
28                     add_to_free_queue(ringID);
29                 }
30             } else {
31                 camfs_getpath(filp->f_dentry,thepath);
32                 ringID=write_to_daemon("read",thepath,count,*ppos,
33                                        inode,CAMFS_BLOCK,found);
34                 down(&cams_mem.ring[ringID].sema);
35                 addFilter(p_inode->i_ino,
36                                 cams_mem.ring[ringID].from_user.filter);
37                 add_to_free_queue(ringID);
38             }
39
40         }
41     }
42
43     return generic_file_read(filp,buf,count,ppos);
44
45 }
```

Figure 5.14: Scruf read operation function

```
1  ssize_t
2  camfs_file_write(struct file *file,const char *buf,
3                              size_t count,loff_t *ppos){
4
5
6      struct inode * p_inode=file->f_dentry->d_parent->d_inode;
7      struct inode * inode=file->f_dentry->d_inode;
8          /* SCRUF - Cam */
9
10     int checkpid;
11
12     checkpid=fromDaemon();
13
14     if ((strncmp(file->f_dentry->d_name.name,".on.",4)!=0)&&
15         (checkpid==1)) {
16
17         unsigned int filter;
18         int found, ringID;
19         char thepath[128];
20
21         found=getFilter(p_inode->i_ino,&filter);
22
23         if (found){
24             if (filter&SCRUF_WRITE_MASK){
25                 camfs_getpath(file->f_dentry,thepath);
26                 ringID=write_to_daemon("write",thepath,count,*ppos,
27                                        inode,CAMFS_BLOCK,found);
28                 down(&cams_mem.ring[ringID].sema);
29                 add_to_free_queue(ringID);
30             }
31         } else {
32             camfs_getpath(file->f_dentry,thepath);
33             ringID=write_to_daemon("write",thepath,count,*ppos,
34                                    inode,CAMFS_BLOCK,found);
35             down(&cams_mem.ring[ringID].sema);
36             addFilter(p_inode->i_ino,
37                               cams_mem.ring[ringID].from_user.filter);
38             add_to_free_queue(ringID);
39         }
40     }
41
42     return generic_file_write(file,buf,count,ppos);
43 }
```

Figure 5.15: Scruf write operation function

```
1  static int scruf_release_file (struct inode * inode, struct file * filp)
2  {
3
4          /* this will have to be made more robust to handle more than scp
5             and filenames which happen to contain scp but are not remote */
6
7          int checkpid;
8          struct inode * p_inode=filp->f_dentry->d_parent->d_inode;
9
10         /* SCRUF - Cam */
11
12         if (trig_release_on){
13             checkpid=fromDaemon();
14
15             if ((checkpid==1)&&
16                     (strncmp(filp->f_dentry->d_name.name,".on.",4)!=0)) {
17
18                 unsigned int filter;
19                 int found;
20                 char thepath[128];
21                 int ringID;
22
23                 found=getFilter(p_inode->i_ino,&filter);
24
25                 if (found){
26                     if (filter&SCRUF_RELEASE_MASK){
27                         camfs_getpath(filp->f_dentry,thepath);
28                         ringID=write_to_daemon("release",thepath,0,0,inode,
29                                                CAMFS_NO_BLOCK,found);
30                     }
31                 } else {
32                     camfs_getpath(filp->f_dentry,thepath);
33                     ringID=write_to_daemon("release",thepath,0,0,inode,
34                                                    CAMFS_NO_BLOCK,found);
35                 }
36             }
37         }
38
39         return ext2_release_file(inode,filp);
40
41 }
```

Figure 5.16: Scruf release operation function

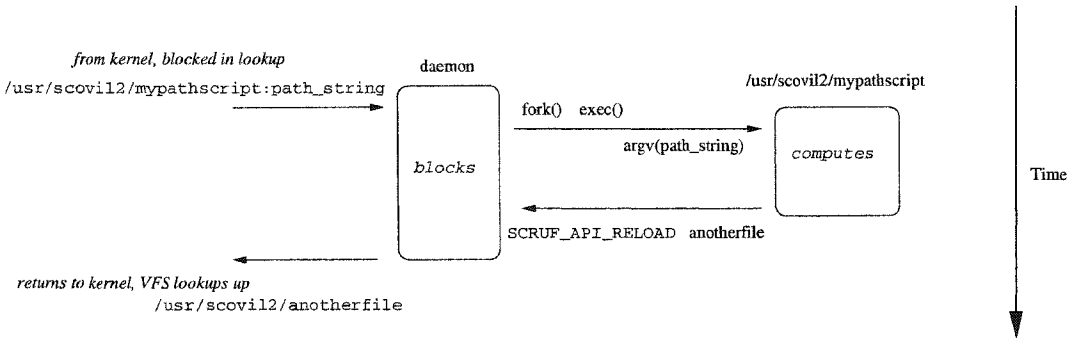

Figure 5.17: A Path Script's Basic Operation

object's structure. The old dentry is released and the open operation returns. The user-level application never sees the original file. When the `open()` call returns in the application, a file descriptor to the new file is returned as if it was the file being opened all along.

Adding Code for Path Scripts

The second type of trigger scripts that are part of Scruf are path trigger scripts. Their general operation is illustrated in Figure 5.17. Path scripts differ from VFS trigger scripts in that they are named explicitly. The path script performs some computation and may redirect Scruf-VFS to a different file. In Figure 5.17, a regular Unix `open()` system call is made on the file `mypathscript:path_string` in the directory `/usr/scovil2`. The path script is invoked by the user-level daemon and should return a new filename which is then used as if it were provided originally. The application which made the `open()` call is unaware of this redirection.

Path scripts are activated at a different point than VFS scripts. When a path script is given in a file's pathname (indicated by the presence of a colon i.e., `trellis:myfile.c`), Scruf-VFS blocks the execution of the VFS during pathname lookup (see Section 5.2) and sends a message describing the path script to the user-level daemon. The fact that path scripts block Scruf-VFS and invoke the path script prior to the pathname lookup is necessary for one important reason. The difference between a path trigger script and an `.on.open` script which redirects a file is that the `.on.open` script is not called if the original file being opened by the application does not exist. If a user attempts to open a file that does not exist, the `open()` system call fails during the pathname lookup, the open file operation is never called, no message is sent to the daemon and the `.on.open` script is not invoked. By having path scripts take effect at lookup time, a user needs to provide only the name of

the path script and a path string (if needed), and the execution of the script will determine the pathname. Path scripts, then, are an explicit action as opposed to the implicit action of the VFS trigger scripts, in that they are explicitly named.

There is an important design in the Linux VFS that makes Scruf path scripts possible. The Linux VFS is designed to allow the underlying file systems to implement varying degrees of functionality. If an underlying file system implements a certain function, such as pathname lookup, the VFS will use that file system's function. If the underlying file system does not implement an operation, the VFS will use a generic function. It is this design that allows us to implement path scripts. While the VFS can drive the lookup of a file itself, it contains a call to a function in the directory operations table of the lower-level file system to allow it to look up the filename on its own. This function's pointer is `d_hash`, which is part of the directory operations table. The name is short for *dentry hash*. As described in Section 5.2, when an inode is a brought into memory from disk, the VFS creates a dentry object which is stored in the dentry cache which is a hash table. The purpose of the dentry hash function is to allow the underlying file system to generate its own hash value. The parent directory and filename are passed to `d_hash` as parameters to allow it to do this lookup and hash generation.

For `ext2`, the directory operations do not implement a `d_hash` function (the VFS therefore uses a generic one). A `d_hash` function was added to the directory operations for Scruf-VFS. The name of the function is `scruf_dentry_hash()` and it is shown in Figures 5.18 and 5.19. However, Scruf-VFS does not implement this function's intended functionality—to look up and hash the directory entries of the path. Instead, Scruf-VFS uses this function to change the path being looked up. To do this, `scruf_dentry_hash()` sends a message up to the user-level daemon (line 43), in the same manner as VFS scripts. It tells the daemon which path script to execute and, in addition, provides the path string following the colon in the pathname. The code blocks in the lookup and waits for a response. When the daemon returns the reply from the path script, a pathname should be provided. If no path is given, Scruf-VFS returns a "no such file or directory" error. If a path is provided, that path is copied (line 52) into the filename to replace the original path script name and path string. When the new pathname is in place, `scruf_dentry_hash()` returns and the new path is looked up by VFS using its generic lookup method used by `ext2`.

It may seem unclear why the `scruf_dentry_hash()` function can be used for something other than its intended purpose. The reason is that, although VFS allows the underlying file system to perform certain functions on its own, the VFS will still use the generic

function if the provided function fails. If `scruf_dentry_hash()` does not return a directory entry, which is the case in Scruf-VFS, then the VFS continues as it would normally, doing the lookups and hashing into the dentry cache using the generic method. However, Scruf-VFS has changed the pathname which was passed into `scruf_dentry_hash()` and that new path is now looked up using the generic method. The existence of the `d_hash` pointer in the VFS is important. Otherwise, the implementation of path scripts may have required modifying the VFS itself—contrary to the design of Scruf-VFS, which limits kernel-level changes to the module. Changing the VFS would require patching and recompilation since the VFS is part of the monolithic kernel. This fact may affect the portability of path scripts, depending on how other OSes implement pathname lookups. If another OSes VFS does not implement a similar lookup method, a new mechanism for modifying the pathname would be required and, if one is not found, path scripts may not be portable to that particular OS.

5.6 Kernel-Daemon Communication

One of the most important design decisions of a user-level file system is the mechanism used to allow the kernel level code to communicate with user-level code. In this section, we will explain the choice of the `/proc` file system and its basic function. This will be followed by a detailed explanation of the actual procedure for communicating, using `/proc`. Communicating from kernel to daemon will be discussed first, followed by daemon to kernel communication.

5.6.1 The `/proc` File System

Linux has three methods for communicating between user-level applications and the kernel: `ioctl()` calls, the `/proc`, and `/dev` file systems. While none offers a complete solution, each offers its own advantages whether it be speed, elegance, or ease of development. Developers must choose the right method for their systems, and it is rare that one is a perfect fit.

For Scruf, we decided to use the `/proc` file system for standard kernel/user-level communication. `ioctl()` calls are not designed for high throughput communication, however we do use `ioctl()` for infrequent out-of-band communication with the kernel. At the time of this work, the `/dev` file system [8] was still experimental in many ways and so `/proc` was chosen for its stability and ease of use.

```
1  static int scruf_dentry_hash(struct dentry * dentry,struct qstr * name)
2  {
3      /* SCRUF - Cam */
4
5      char * op_ptr, *place_ptr, * prevslash;
6      char orig[128], op[16];
7      int dist,i,len, op_len, checkpid;
8      unsigned int c, nf;
9      char * mystr;
10     unsigned char opencmd[136];
11
12     if (cams_mem.daemon_pid==-1) return 0;
13
14     checkpid=fromDaemon();
15
16     op_ptr=strchr(name->name,':');
17
18     if ((checkpid==1)&&(op_ptr)){
19
20         prevslash=op_ptr;
21         /* search back to the previous slash */
22         while (*prevslash!='/') prevslash--;
23         prevslash++;
24         op_len=(int)op_ptr-(int)prevslash;
25         strncpy(op,prevslash,op_len);
26         op[op_len]='\0';
27
28         /* orig is a copy of the name */
29         strcpy(orig,name->name);
30
31         /* pass the filename to the userlevel daemon */
32
33         /* get next available buffer and
34            increment the buffer, within a lock */
35
36         down(&cams_mem.global_sema);
37         nf=get_next_to_fill();
38         up(&cams_mem.global_sema);
39         cams_mem.ring[nf].to_user.blocked=SCRUF_BLOCK;
40         cams_mem.ring[nf].to_user.code=TRELLIS;
41         cams_mem.ring[nf].to_user.buffer=nf;
42         sprintf(cams_mem.ring[nf].to_user.file,"%s\0",orig);
43         add_to_process_queue(nf);
44
45         up(&cams_mem.daemon_wait);
46         down(&cams_mem.ring[nf].sema);
47
```

Figure 5.18: The dentry hash function (a)

```
48          /* make the changes after the first slash */
49          place_ptr=strchr(name->name,'/');
50          place_ptr++;
51
52          strcpy(place_ptr,cams_mem.ring[nf].from_user.file);
53          if (prevslash==name->name){
54              strcpy(prevslash,"./");
55              name->len=1;
56          } else {
57              name->len=7;
58          }
59
60          add_to_free_queue(nf);
61
62          /* rehash the first directory in the filename */
63
64          name->hash = init_name_hash();
65          mystr=(unsigned char *)name->name;
66          do {
67              mystr++;
68              name->hash = partial_name_hash(c, name->hash);
69              c = *(const unsigned char *)mystr;
70          } while (c && (c != '/'));
71
72          name->hash = end_name_hash(name->hash);
73
74      } else {
75          /* This code checks to kill the daemon */
76
77          if ((checkpid==1)&&(strncmp(name->name,"kill",4)==0)){
78              printk("[SCRUF] killing daemon\n");
79
80              /* get next available buffer and
81                 increment the buffer, within a lock */
82              down(&cams_mem.global_sema);
83              nf=get_next_to_fill();
84              up(&cams_mem.global_sema);
85
86              cams_mem.ring[nf].to_user.code=SCRUF_KILL;
87              add_to_process_queue(nf);
88
89              up(&cams_mem.daemon_wait);
90          }
91      }
92
93      return 0;
94  }
```

Figure 5.19: The dentry hash function (b)

The /proc file system is unique with respect to most other file systems, in that the data it represents does not correspond to a real file in any way. Recall, the file mount table in Figure 5.3 contains an entry for /proc file system (line 2). The /proc mount point's device entry (the first field) reads none, as it does not correspond to a real disk device. The /proc file system is used to retrieve data from the kernel and to present it in a manner exactly like that of plain text file—as if it were a conventional Linux file system. The /proc file system is a simplified file system; it maintains the view of files similar to any Unix file system. It is accessed through the standard open(), read(), write(), and close() system calls. As a result, its implementation is a stripped-down file system compared to others, such as ext2.

Examples of /proc *pseudo-files* are /proc/cpuinfo and /proc/pci which, when read, present data about the CPU and PCI devices, respectively. The information which is presented does not come from a file, but is system-specific data stored in the kernel. The kernel code produces the data that is sent to the user via the /proc pseudo-file as though it were a regular file, for convenience and transparency. By simply executing cat /proc/cpuinfo in a shell, a user is presented with CPU type and clock speed, as well as memory and other information in plain text format. All /proc pseudo-files are stored under the /proc directory.

While /proc/cpuinfo and /proc/pci are examples of specific uses, more generally, the /proc file system is a mechanism by which the kernel can interact with the user-level via file system calls. Since /proc does not actually store or retrieve files on disk, one might think that /proc is a read-only file system; however, /proc is available for writing as well. It can be used to send data to and receive data from the kernel, and writing to a /proc pseudo-file is simply a method of communicating to kernel level code.

/proc files do not require functions to handle directory operations or low-level disk management; the only operations which really matter are read and write. The code for a /proc file system is contained within a module which, in our case, is part of Scruf-VFS. In this way, Scruf-VFS actually contains code to handle two file systems: the Scruf file system and Scruf's /proc file system. To explain, when a partition is mounted as type Scruf, the Scruf-VFS module is loaded into the kernel. First, the function to register the Scruf file system is called; this hooks Scruf-VFS into the VFS. The VFS is then able to access files from the disk partition. Second, the initialization function for the /proc file system is called, which creates the actual /proc/scruf pseudo-file, and sets function pointers to the customized read and write operations. The implementation of Scruf-VFS's /proc file

```
1  static int proc_read_scruf(char * page, char ** start,
2                             off_t off, int count,
3                             int *eof, void * data){
4
5      int nb;
6
7      if (cams_mem.daemon_pid!=-1){
8
9          down(&cams_mem.daemon_wait);
10         nb=get_next_to_process();
11
12         MOD_INC_USE_COUNT;
13         *start=&cams_mem.ring[nb].to_user;
14         MOD_DEC_USE_COUNT;
15
16         /* if the operation is not blocked move the
17                     ringslot to the free queue */
18         if (cams_mem.ring[nb].to_user.blocked==CAMFS_NO_BLOCK){
19             add_to_free_queue(nb);
20         }
21     }
22
23     return sizeof(struct kernel_to_user_t);
24 }
```

Figure 5.20: Scruf-VFS `/proc` read function

system read and write operations will be discussed over the next two sections.

Currently, one issue with our current implementation of the `/proc` file system is that only one user may use Scruf at a time; the daemon does not currently support multiple users. A problem arises if the user-level daemon is started by user A, and user B performs a file operation which will send a message to the daemon. Since the daemon is running under user A's permissions, it will not be able to access the trigger scripts in user B's directory without setting group or world file permissions, neither of which is acceptable. The current implementation of Scruf is a proof of concept that trigger scripts are a sound model for implementing file system extensions. This issue will be addressed in the future as Scruf progresses to become a file system for regular use.

5.6.2 From the Kernel to the Daemon

While the interaction between Scruf-VFS and the daemon was earlier described as request-response, the one-way communication from the kernel to the daemon ((4) in Figure 5.4) can be described as a producer-consumer process. When a file operation occurs (i.e., VFS trigger script), or a pathname contains a colon (i.e., path trigger script), the kernel must communicate with the daemon. To do this, the kernel code (the producer) copies the data

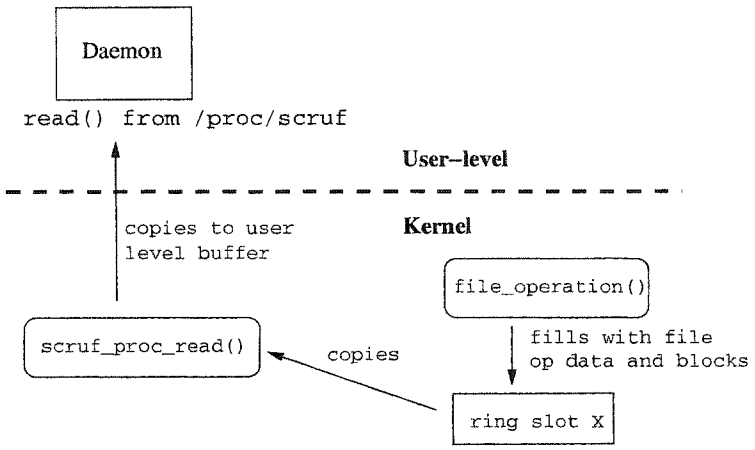

Figure 5.21: A read of `/proc/scruf`

describing the operation or path script into a ring slot and places it in the work queue. The user-level daemon (the consumer) takes messages off this queue and, using the data they contain, invokes the appropriate trigger scripts.

To retrieve a message from the kernel, the daemon performs a `read()` of the pseudo-file `/proc/scruf`. This is illustrated in Figure 5.21. As part of initialization of the daemon, the daemon opens the pseudo-file using a standard `open()` system call. A read of the `/proc/scruf` file invokes the `/proc` read operation (Figure 5.20) in the kernel, which is set up when the module is loaded, as described earlier.

In this description of the Scruf-VFS `/proc` read operation, all line numbers given refer to Figure 5.20. The purpose of the read operation is to copy the next message, stored at the head of the work queue, up to the daemon (Figure 5.5). To do this, the code first blocks to wait for a message on the `daemon_wait` semaphore (line 9) which is a counting semaphore which queues up the file operations. When a message is ready to be copied, the operation is unblocked. It takes the ring slot at the head of the work queue (line 10) and assigns a pointer (line 13) to the `kernel_to_user_t` structure (Figure 5.7) in that slot which will copy the structure up to the user-level. The pointer which is assigned is passed in as an argument into the `/proc` read operation. The read operation returns and the data is copied by the system up to the user-level buffer. When the `read()` system call returns to the daemon, the buffer passed into the system call contains the data.

One special case occurs in the `/proc` read operation. Normally, a Scruf-VFS file operation places the ring slot it used onto the free queue when it is finished with it (for example, lines 32 or 40, Figure 5.11). However, if a file operation does not block (currently

```
1  static int proc_write_scruf(struct file * file, const char * buffer,
2                              unsigned long count, void * data){
3
4      int len,returnval,buf;
5      struct scruf_data_t * sf_data = (struct scruf_data_t *)data;
6      struct user_to_kernel_t ringslot;
7
8      copy_from_user(&ringslot,buffer,sizeof(struct user_to_kernel_t));
9      len=sizeof(struct user_to_kernel_t);
10     returnval=ringslot.returnval;
11
12     buf=ringslot.buffer;
13
14     if ((buf>=0)&&(buf<RING_SIZE)){
15         memcpy(&cams_mem.ring[buf].from_user,&ringslot,len);
16         if (ringslot.blocked==CAMFS_BLOCK) {
17             up(&cams_mem.ring[buf].sema);
18         }
19     } else {
20         printk("**why is the buffer out of range?\n");
21     }
22
23     return len;
24 }
```

Figure 5.22: Scruf-VFS `/proc` write function

only release may not block), the `/proc` read operation will place the ring slot onto the free queue once it has copied the data from it to user-level. This is performed by the `if` statement (lines 18 to 20).

5.6.3 From the Daemon to the Kernel

When the script has finished its processing, or if no script exists, the daemon must return this information to the kernel ((7) in Figure 5.4). Figure 5.22 illustrates how this is done through `/proc/scruf`. To return information to the kernel, the daemon performs a `write()` operation on the `/proc/scruf` pseudo-file. It writes a `user_to_kernel_t` structure (Figure 5.8). When a write occurs to `/proc/scruf` pseudo-file, the Scruf-VFS `/proc` write function (Figure 5.22) is invoked. In this description of the `/proc` write function, all line numbers refer to Figure 5.22.

One of the parameters passed to this function is a pointer to an address in user space, which is the buffer passed to the standard Unix `write()` system call. The `/proc` write function copies from this address using the Unix function `copy_from_user()` (line 8) and copies the number of bytes equal to the size of the `user_to_kernel_t` structure (Figure 5.8). The data is first copied to a temporary structure. The ring slot ID number is read

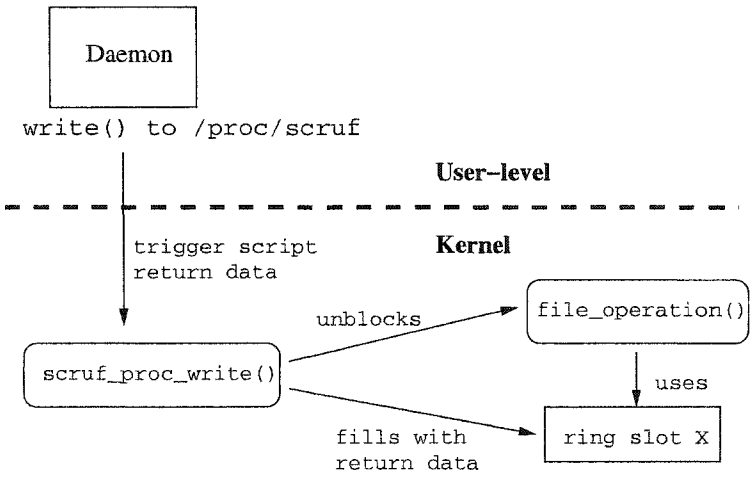

Figure 5.23: A write to `/proc/scruf`

(line 12) and then the whole structure is copied to the correct ring slot (line 15). If the file operation is blocked waiting for the reply, the write function unblocks it by performing an *up* operation (line 17) on the semaphore in that slot. As mentioned before, if a file operation blocks, once it is unblocked it will copy the data it needs out of the `user_to_kernel_t` structure in the slot it used—such as the filter for the directory or a new filename. Once this is done, it places the slot it used onto the free queue and continues with its execution.

5.6.4 Out-of-band Communication

Occasionally, the daemon will wish to send certain control messages to the Scruf module that are not part of regular operation of the Scruf file system. This is known as out-of-band communication. These messages are similar to control messages in TCP/IP networks which may change the rate at which packets are sent, but do not actually transfer data. For this sort of communication, Scruf uses `ioctl()` commands, so as not to interfere with the regular Scruf communication. `ioctl()` messages do not use the ring or any other data structures in Scruf. The messages are usually a notice to change some state in the Scruf module, and do not require in-order processing; so, for this purpose, `ioctl()` calls work well. Currently, we use this method for communicating initialization and shutdown information to the module when the daemon is started and stopped.

5.7 The User-Level Daemon

Up until now, this chapter has described the function of the Scruf framework that runs in the kernel and makes up the Scruf-VFS loadable module, i.e., the Scruf file operations, data structures, and the `/proc/scruf` pseudo-file. What remains is to describe the user-level part of the framework (see Figure 5.4), which includes the user-level daemon, the API between the daemon and the scripts and the scripts themselves.

The daemon that runs at user-level is conceptually simple. It does not process the operations, but acts as a bi-directional multiplexer in a one-to-many environment. On the *one* side is the kernel which the daemon communicates with via `/proc/scruf`. On the *many* side are the Scruf path and VFS trigger scripts. As mentioned before, the daemon communicates with the kernel using stream data IPC through `/proc/scruf`. It opens the channel with an `open()` call on `/proc/scruf` when the daemon is started, communicates with the kernel via `read()` and `write()` system calls, and when the daemon is terminated, closes the channel with a `close()` call. As with all Unix stream I/O, the handle is simply a file descriptor. The daemon communicates with the trigger scripts using IPC pipes which are also managed and accessed through `open()`, `read()`, `write()`, and `close()` system calls.

If a trigger script is a VFS script, the daemon first checks an internal data structure to see if a process is already running that script, based on the inode number of the script (since inode numbers are unique). Using the inode number ensures that the daemon does not start copies of the same script when multiple scripts are symbolic links to a single script. The inode of a file is easily retrievable from a file's pathname. If the script is running, the daemon sends a message to it describing the file operation. If the script is not running, the daemon first check that it exists; if it does, it invokes it, sends the data and waits for the response, or for the script to exit. The daemon may also determine the filter for the CWD, if necessary, and pass that back to the kernel as well. The daemon knows whether Scruf-VFS needs the filter, based on a flag (line 11, Figure 5.7) passed from the kernel. Determining the filter is as simple as checking the trigger scripts that exist in the CWD.

If the message from the kernel is for a path script, the daemon has a much more straightforward execution pattern. Path scripts do not support persistence or filtering and so the daemon always starts the script, passes the data on the command line, and then waits for a reply. When the script exits it may return a path to the daemon which is returned to the kernel. If the path script does not return a path, the daemon will return an empty path to the

kernel, which will result in an access error as if an invalid path were provided originally by the application.

When a user wishes to turn off the Scruf framework or unmount a Scruf partition, the daemon must be terminated. The daemon can be killed by simply accessing the name *mountdir*`/kill`. So, if Scruf is mounted on `/usr/scovil2/`, any operation on the file `/usr/scovil2/kill`, such as `cat` or `touch`, will kill the daemon. `kill` is a special name that Scruf-VFS is looking for, and because of this, the kill command comes from Scruf-VFS in a message through the `/proc/scruf` pseudo-file. The actual file `/usr/scovil2/kill` does not have to exist since the check is performed before the lookup. Therefore, even if the lookup fails the kill message will still be sent. On receiving the kill command, the daemon will kill any running scripts, close `/proc/scruf` and exit.

5.7.1 Daemon-Script API

To allow users to write their own scripts, the scripts must have a way to pass control to and from the daemon. To do this, an API has been designed which gives the user as much discretion as possible in the operation of the scripts to tailor their extensions in terms of complexity, efficiency, and general operation. The API is based on text messages, so it is language independent and free of any libraries. Using ASCII messages allows the daemon and scripts to communicate using Unix pipes over standard in and standard out. The API is summarized in Table 5.1.

The API for path scripts is a stripped-down version of that used for VFS scripts, so the VFS scripts will be described first. VFS scripts have three categories of messages to send to the daemon. The first category describes a script's mode of operation. When a VFS script is started, it must describe its behaviour. The two behaviours it must describe are (1) whether it is persistent and (2) whether it is synchronous or asynchronous. To tell the daemon it is persistent, the script sends the string `SCRUF_API_PERSIST`. To tell the daemon it is non-persistent, it sends `SCRUF_API_NONPERSIST`. The script indicates it is synchronous by sending `SCRUF_API_SYNC` or, if it is asynchronous, it sends `SCRUF_API_ASYNC`. Persistence information must be sent first. If no behaviour information is sent, or if the values are invalid, the daemon assumes the script is non-persistent and synchronous. Therefore, scripts which do not send information should still work, they will just run slower.

The second category of information consists of the return values for the individual file operations. Once a script has provided its behaviour information, the daemon knows what interaction is required. At this stage in their development, Scruf scripts (both VFS and

path) have only two relevant actions: interposed action and redirection action (Section 4.2). For interposed action, a script returns the string `SCRUF_API_NOOP`, which allows the file operation to go ahead unaffected. In the case of path scripts, this results in an error, since Scruf-VFS will have no pathname to redirect to. For redirection, the script returns `SCRUF_API_RELOAD` and the new filename to be loaded. Redirection is currently only relevant on `open()` calls and so is only accepted from `.on.open` scripts.

Path scripts do not require behaviour information; they are assumed to be non-persistent and synchronous. Persistence is something that will be developed in future work. The second assumption, that path scripts should be synchronous, seems logical at this point. In our experiments, path scripts are utilized only to determine a pathname to be used by an application; so, for now, it is acceptable that the daemon should wait to receive a response. As new path script extensions evolve, which may have consistent return values, Scruf may be made to support asynchronous path scripts.

Debugging of scripts will be a reality as users develop their extensions. Above, we mentioned that the daemon waits for a response from synchronous VFS scripts and from all path scripts; however, it will not wait indefinitely for a response. If no response comes after 1 second[1], a timeout occurs, the daemon will return a `SCRUF_API_NOOP` code to the kernel and execution will continue.

The first two categories of messages describe communication that is passed from the script to the daemon. The third category consists of control messages for persistent scripts; we will now describe their API. For all scripts, path and VFS, all data describing the operation is passed as command line arguments. However, to support persistent VFS scripts, scripts also receive data via standard in IPC. They will receive information as an ASCII string, for example:

```
SCRUF_API_ARGS /usr/scovil2/alberta/myfile.c,read,0,4096,1665,16:41:30.716
```

While this information is redundant for non-persistent scripts and on the initial invocation of persistent scripts, it is necessary for subsequent messages to persistent scripts, which obviously cannot be passed as command line arguments as they are already running.

The final API command is the command for a persistent script to exit. To kill a running script, either when the daemon exits or when it needs to make room in the file descriptor hash table, the daemon simply sends the string `SCRUF_API_KILL` to the script via standard in and the script should exit, at which time the file descriptors are closed.

[1]We realize this timeout period is short. In future, this timeout period may be made configurable.

Category	Command	Sender	Receiver	Summary
Behaviour Information	SCRUF_API_PERSIST	VFS script	Daemon	Tells the daemon this script is persistent.
	SCRUF_API_NONPERSIST	VFS script	Daemon	Tells the daemon this script is non-persistent.
	SCRUF_API_SYNC	VFS script	Daemon	Tells the daemon this script is synchronous.
	SCRUF_API_ASYNC	VFS script	Daemon	Tells the daemon this script is asynchronous.
Script Return Values	SCRUF_API_NOOP	Path or VFS Script	Daemon	Interposed action, file operation continues unchanged.
	SCRUF_API_RELOAD	Path or VFS Script	Daemon	Redirection action, must be followed by a filename, redirects VFS to new file. Can only come from an `.on.open` VFS script.
Persistent Script Controls	SCRUF_API_ARGS	Daemon	Persistent VFS Script	Data string for persistent VFS script describing a file operation.
	SCRUF_API_KILL	Daemon	Persistent VFS Script	Tells the persistent script to exit.

Table 5.1: Summary of Trigger Script API

5.7.2 Daemon-Script Communication

In the previous section, Scruf's API for trigger scripts was explained. Now we will describe how the daemon invokes trigger scripts, sets up the communication channel between itself and the scripts, and how the daemon and a script interact.

The daemon invokes the scripts in the standard method by which a parent process creates a child process in Unix—via `fork()` commands and pipe IPC. Pipes are created by opening two file descriptors, with the `pipe()` command, forking the child process and setting up the pipes to create a channel between the parent and child processes. Since the pipes we use are uni-directional, parent and child both have two pipe ends: one pipe from parent to child and the other from child to parent. Therefore, the daemon has a pair of descriptors for each running script: an *out* file descriptor to write to the script and an *in* file descriptor to read from the script. Once the pipes are set up, the child then performs an `exec()` call to replace itself with the running script. To allow the newly created script (the child) to communicate with the daemon (the parent), the child first sets its ends of the pipes to its standard in and standard out streams; this must be done before the `exec()` call. When

this is all in place, the script and daemon can communicate. In summary, to send data to a script, the daemon writes to that script's *out* file descriptor, and to read from the script it reads from the *in* file descriptor. The script's communication is even simpler: to write to the daemon it writes to standard out, and to read from the daemon, it reads from standard in.

If the script is persistent, the daemon keeps track of its file descriptors in a hash table where the key is the script's inode number, which is retrieved from the `stat()` function. When the daemon needs to communicate with a script, it simply looks it up in the hash table. If the script is in the table, the daemon uses the *in* and *out* file descriptors to communicate with it. If the script is not in the table, the script is not running and so the daemon starts the script executing.

Using standard in and standard out as the communication mechanism between the scripts and the daemon also simplifies trigger script development, since users who are creating their scripts can easily test them by simply running and interacting with them via a command line. A user can invoke a script explicitly, and by mimicking the behaviour of the daemon, by typing the `SCRUF_API_ARGS` and `SCRUF_API_KILL` commands, the user can verify the values that are returned before putting the trigger scripts in place.

5.8 Improving the Performance of VFS scripts

Five features have been mentioned previously that streamline Scruf for both improved performance and quick development of file system extensions. The first two features, symbolic links and inheritance, make Scruf scripts easy to set up. The remaining three allow users to write efficient scripts and eliminate unnecessary overheads of the Scruf framework; they are: filtering, persistent scripts, and asynchronous scripts. This last section of this chapter will describe them in greater detail. The next chapter will test some of these features and measure their impact on performance.

5.8.1 Symbolic Links

A symbolic link can be used as though it were a regular file but, in reality, it is actually a pointer to another file or directory. A file and a symbolic link to that file, resolve to the same inode. Symbolic links are useful when the scripts for different operations are identical, such as in Scruf-Trace.

Creating one actual script, `.on.open`, and linking `.on.read`, `.on.write`, `.on.lseek` and `.on.release` to `.on.open` has two advantages. First, it saves on the space occupied by VFS trigger scripts, as links take up less storage, especially when the trigger

scripts are large binary executables. Second, symbolic links can reduce the number of running processes when persistent scripts are used. To explain, if a directory contains five copies of the same script, it will require that all five scripts be started up the first time their operation occurs. Consider a `cat` operation on a file, where `.on.open` is an actual file and the other four trigger scripts are symbolic links to the `.on.open` script. `cat` causes a file to be opened, read and released. The open is the first operation, and so the `.on.open` script will be started. When the read operation occurs, the daemon will examine the `.on.read` script. If it is a link to the `.on.open` script, the inode number will be the same as the `.on.open` script found in the hash table. The daemon will not start up a new process for the `.on.read` script, but will send the `.on.read` data to the running `.on.open` script. The same is true for all successive file operations in that directory. Recall, that one of the fields of information passed to a VFS script is the type of operation (i.e., open, read), so a single script can differentiate between operations if required to do so. If `.on.read` were its own file, it would have a different inode number than the `.on.open` script and the daemon would have to start a new process. Therefore, if all five scripts were copies, this directory would take up five entries in the hash table of running scripts, as opposed to taking up only one entry if symbolic links were used.

Finally, a process running in Unix has a limit to the number of file descriptors it can have open (i.e., 1024 in Linux). Since each running scripts requires two file descriptors, the *in* and *out* file descriptors described in Section 5.8.1, only (approximately) 500 scripts can be running at any time before hitting the 1024 limit (most likely fewer, due to hash table collisions).

Of course, using symbolic links only applies if the trigger scripts are identical, but it is still a simple modification that can save programming effort and improve performance.

5.8.2 Inheritance

Setting up VFS scripts in every single directory is a nuisance—especially over a large directory tree—which may make Scruf unappealing to users. Inheritance is a feature that was added to ease the installation of Scruf, and allow users to experiment without a great deal of setup time. Figure 5.24 illustrates how the Scruf inheritance mechanism works. The directory **/usr/home/cam/** contains no trigger scripts, **/usr/home/** contains only an `.on.open` script, and **/usr/** contains `.on.write` and `.on.read` scripts. **/usr/home/cam/** will inherit the `.on.open` script from **/usr/home/**, since it is a closer ancestor than **/usr/**. Since **/usr/home/** contains a trigger script, **/usr/home/cam** only inherits that one script and its

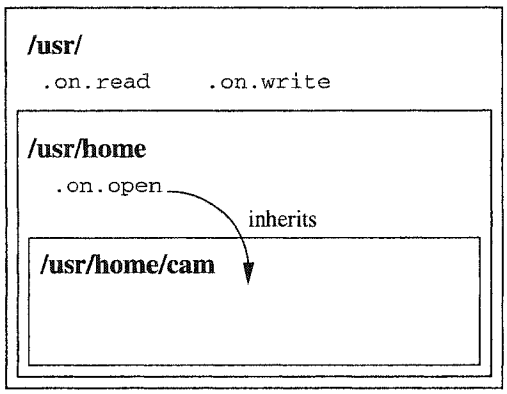

Figure 5.24: An Example of Scruf's Inheritance Mechanism

filter in Scruf-VFS is set accordingly. If a read operation occurs on a file in **/usr/home/cam** it will be filtered out. Inheritance will not search for the `.on.read` script in **/usr/** since it has found VFS scripts to inherit in **/usr/home/**.

/usr/home/ will not inherit the `.on.write` and `.on.read` scripts from **/usr/** since it already contains a trigger script, `.on.open`; therefore, inheritance is not applied.

Inheritance allows a user to place a set of scripts in the root of a sub-tree and have Scruf act as if those files were in all sub-directories below that root. This saves the user from both setup effort and clean-up effort, if the user chooses to remove an extension. Inheritance applies only if the CWD contains no VFS scripts and is applied on a directory basis, not on individual scripts. When a directory inherits scripts from an ancestor directory in the hierarchy, it inherits all of that directory's scripts. It does not try to fill in the missing scripts by looking further up the hierarchy; it stops as soon as it finds any scripts in a directory above it. To say it another way, a directory can only inherit from one ancestor and inheritance comes from the closest ancestor up the hierarchy.

When a directory inherits an ancestor's scripts, the scripts are not copied nor are symbolic links set up; the inheritance is handled entirely within the user-level daemon. The scripts are not copied or linked to the CWD from the ancestor, as we are hesitant to have the system modify files. When a directory inherits scripts, its inheritance information is stored in a hash table in the daemon. The directory is hashed by its inode number and the value stored with it is the path, not the inode, of the directory it inherits from. The daemon only stores directories which inherit scripts from an ancestor; if a directory has scripts in it, it is not stored in the table.

When a message comes up from the Scruf module (see Figure 5.4), the daemon checks

for an entry in the table; if none is found, it works from the CWD of the file and looks for scripts there. This may seem like an overhead, as directories with no scripts or inheritance could still be searched every time, but it is not. Once a directory has been found not to contain nor inherit scripts, its filter will be set in the kernel and no subsequent messages for that directory will be sent. A message may be sent if the filter entry for the directory is removed in the kernel, but that is a pathological case and it is as likely that directory's entry in the inheritance table would be removed. The inheritance table has limited space and when it runs out of space it must remove an entry to make room for a new one. It uses a least recently used (LRU) policy to select a victim to be removed from the table.

5.8.3 Filtering

Having every file operation result in a message being sent to the daemon can be costly. Each message involves two expensive operations: an IPC from the Scruf-VFS module to the daemon and a context switch. These two mechanisms alone have proved to be large hurdles to efficiency in microkernel operating systems. Each message also requires a search for, and possible invocation of, a script at the user-level. If the particular VFS script for an operation does not exist in the CWD, all this communication is wasted.

To eliminate this inefficiency in Scruf, we have implemented message filtering in the Scruf-VFS module. Filtering aims to improve performance by eliminating unnecessary and expensive IPC between Scruf-VFS and the daemon from file operations whose trigger scripts do not exist in the CWD. To filter superfluous messages, the Scruf kernel module, Scruf-VFS, maintains a hash table of directories that files have been accessed in. Along with the directory, which is hashed by inode number, it stores a bit vector which indicates the trigger scripts that exist in that directory. Currently, only VFS trigger scripts are filtered. Figure 5.25 illustrates how filtering works in Scruf-VFS.

When a file operation occurs, the module checks that the CWD has a script for that operation, by looking it up in the hash table. If the CWD does have that script, a message is sent up; if the script does not exist, no message is sent. In Figure 5.25, since `.on.open` and `.on.read` scripts exist in the CWD, messages for open and read operations will be sent. Messages for write operations will be filtered, since there is no `.on.write` script.

Due to limited storage, the hash table behaves like a cache. When it runs out of room, it must remove an entry in order to make room for a new one. The table uses a random replacement strategy to select a victim.

The first time a directory has one of its files accessed, no filter will be present. This will

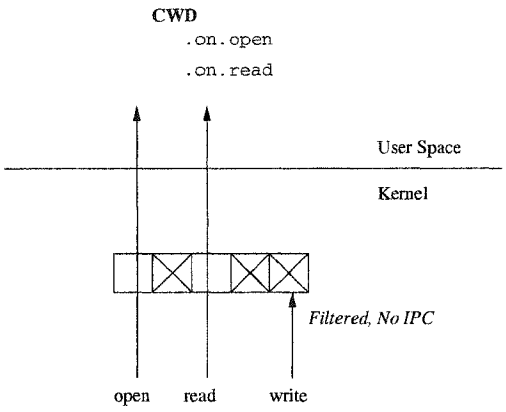

Figure 5.25: An Example of Scruf's Filtering Mechanism

also occur if its entry has been removed by the replacement policy. When the Scruf module does not have an entry for the CWD, along with the message describing the file operation that occurred, a request for a filter for that directory will be sent to the daemon, and the file operation will be forced to block. When the reply comes back, a filter will be included which will be entered in the hash table. It is the daemon that actually searches the directory for the presence of trigger scripts, and attaches that information to messages returned to the kernel.

Filtering is very important in eliminating the inherent overhead of the Scruf framework when scripts are not present. If a new user wishes to experiment with Scruf, he or she can mount an `ext2` partition as a Scruf partition and see negligible overhead if no scripts are used, because the messages are filtered.

5.8.4 Persistent Scripts

VFS scripts have the potential for heavy usage. In our experiments with Scruf-Trace building the Linux kernel (Section 6.2), Scruf-Trace logs, on average, 2,300 file accesses per second. Each log entry corresponds to a communication between the daemon and a script. If scripts must be started each time, that is 2,300 `fork()` and `exec()` calls per second, which is a massive overhead. What would be desirable is to be able to start a script once and leave it running to avoid the overhead of starting the script every time. What is needed are so-called persistent scripts.

In Figure 5.26 is an example of a template of a persistent script written in C. The script begins by defining its behaviour, by writing the relevant API codes to the daemon (line 8). It tells the daemon it is a persistent script by passing the `SCRUF_API_PERSIST` string,

```
 1  int main(void){
 2
 3      char header[32];
 4      char args[128];
 5      int kill,fd;
 6
 7      /* send behaviour information to daemon*/
 8      printf("SCRUF_API_PERSIST SCRUF_API_ASYNC\n");
 9      fflush(stdout);
10
11      kill=0;
12      /* loop until kill command */
13      while (!kill){
14          scanf("%s",header);
15              if (strcmp(header,"SCRUF_API_ARGS")==0){
16
17                  scanf("%s",args);     /* read arguments */
18                  do_processing(args)   /* process argument */
19
20              } else if (strcmp(header,"SCRUF_API_KILL")==0){
21                  kill=1;
22              }
23      }
24  }
25
```

Figure 5.26: A persistent script

and that it is asynchronous by passing the SCRUF_API_ASYNC string (a persistent script does not have to be asynchronous). The heart of the script is the while loop (lines 13 to 23), which processes the messages from the daemon, identified by the SCRUF_API_ARGS header token. The loop processes the arguments describing the file operation and then waits for the next message from the daemon. When the daemon wants the script to exit, it sends it the SCRUF_API_KILL string which causes the script to exit. The daemon then closes the file descriptors for the IPC pipe to the script.

As mentioned in Section 5.8.1, Scruf can have hundreds of scripts running at once. The file descriptors the daemon uses to communicate with the script are stored in a hash table. When the hash table is full and a new script needs to be started, the daemon has to kill one of the running scripts to make room. The daemon selects a victim using an LRU policy. Once the victim is selected the daemon sends the SCRUF_API_KILL string to the script and closes its file descriptors to that script to free them up to use for the new script about to be started. Users should try to use as few scripts as possible by taking advantage of symbolic links and inheritance, as killing and restarting scripts can have a noticeable negative impact on performance.

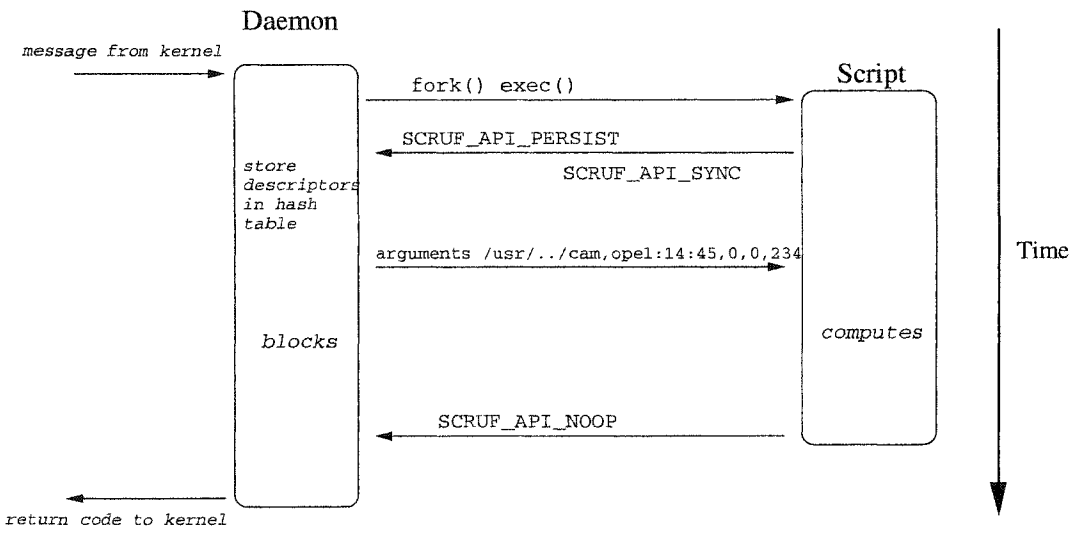

Figure 5.27: IPC between the Daemon and a Synchronous VFS Script

5.8.5 Synchronous v. Asynchronous Scripts

A significant overhead can occur when the daemon must wait for the reply from a VFS script before returning and unblocking the file operation in the Scruf-VFS. This overhead may be unnecessary, especially when the script's operation is an interposed action, as in the case of Scruf-Trace. Scruf-Trace is a simple logger and will never modify the VFS operations it processes. This type of file system extension would benefit from being able to tell the daemon not to wait for a response, because it will always be `SCRUF_API_NOOP` (which will not effect the VFS operation). To handle cases such as this, Scruf supports both synchronous and asynchronous scripts. Diagrams of the basic operations of synchronous scripts and asynchronous scripts are in Figures 5.27 and 5.28, respectively.

Synchronous scripts always return a value, therefore the daemon must wait for a return code, as it may modify the VFS operation, and return the code to Scruf-VFS. Since these scripts block the daemon to wait for their response, they can also be called blocking scripts. This synchronous behaviour involves a waiting period. Notice in Figure 5.27, the daemon waits while the script computes. This waiting period may be small for a single file operation, but may add up to significant overhead for applications with intensive file I/O.

Asynchronous (or non-blocking) scripts can avoid this waiting period by telling the daemon, when they begin execution, that it should not wait for a response and can return `SCRUF_API_NOOP` and unblock the VFS operation immediately. Then, when the daemon sends a message to an asynchronous script, it does not wait for a response; instead, it

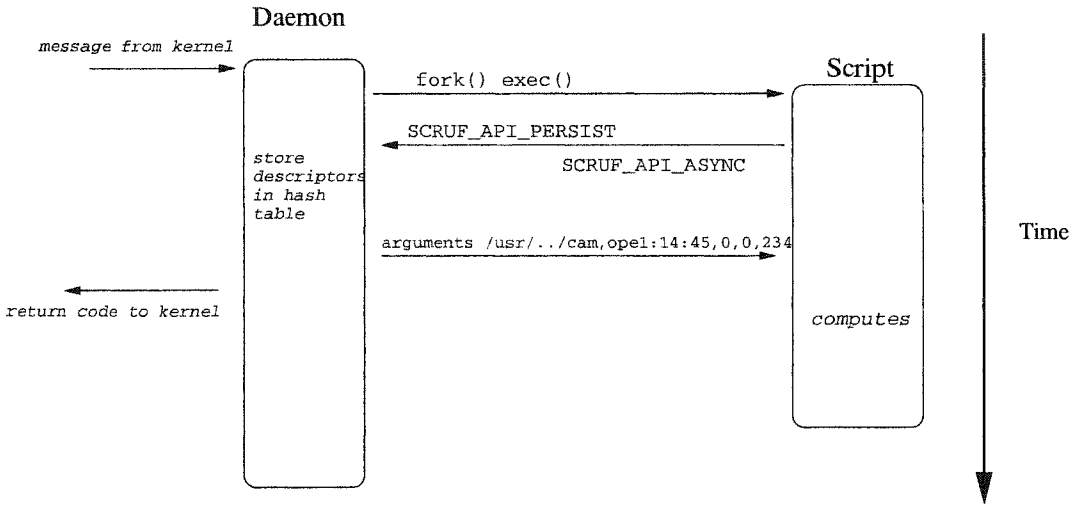

Figure 5.28: IPC between the Daemon and an Asynchronous VFS Script

immediately assumes a return value of SCRUF_API_NOOP and returns to Scruf-VFS. By avoiding this unnecessary waiting period that synchronous scripts require, asynchronous scripts may provide a significant performance improvement.

5.9 Concluding Remarks

This chapter has described, in detail, the implementation of the Scruf framework. We began by describing the Scruf-VFS module which resides at the kernel level. We then described the communication mechanism between kernel and user-level which uses the /proc file system. Then, the implementation of the user-level daemon, the general structure of both VFS and path trigger scripts, and the daemon-script API were described. The chapter concluded by describing five mechanisms for improving the performance of trigger scripts. Now that the framework has been fully described, what remains is to test our implementation with some of the extensions from Chapter 4—in order to demonstrate that Scruf not only provides useful file system extensions, but does so without having a prohibitive negative effect on system performance.

Chapter 6

Experimentation and Performance

In previous chapters, we have demonstrated how the Scruf framework and trigger scripts provide a new approach for extending file system functionality. What remains is to show that the overhead of running the Scruf framework, with and without extensions in place, is kept to a level which is acceptable.

Our experimental platform for running Scruf is a dual 700 MHz Pentium III machine with 512 MB RAM, running Linux 2.4.18. An existing `ext2` platform is mounted using the Scruf loadable module. For the Scruf-Trellis experiments, the remote files are stored on a machine with a 700 MHz Pentium III and 512 MB RAM, running Linux 2.4.9.

For our main benchmarks we have selected the build process of the Linux kernel and the well-known modified Andrew benchmark. Both benchmarks are common in OS and file systems research, and provide a well-understood test of Scruf. For both benchmarks, the Scruf inheritance mechanism is used, meaning only one set of scripts was placed at the root of the source trees. These benchmarks demonstrate that the functionality of the Scruf framework and trigger scripts comes at a minimal to acceptable overhead. All scripts used in these experiments are included in the appendixes.

6.1 Microbenchmarks

Microbenchmarks are measures of the overheads of individual pathways in the Scruf framework. Two important measures are the overhead of the IPC between the kernel and the daemon, and the overhead of the IPC between the daemon and the scripts. These IPC pathways are illustrated in Figure 6.1. We test both synchronous and asynchronous scripts, as well as persistent and non-persistent scripts.

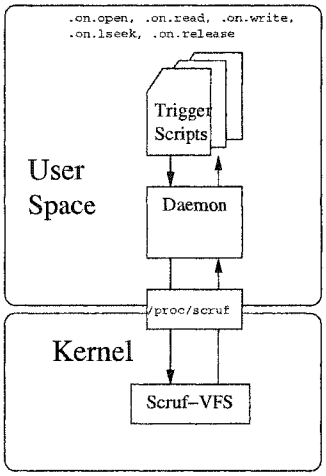

Figure 6.1: The IPC Pathway in Scruf

6.1.1 Synchronous v. Asynchronous Scripts

Scruf's design follows some simple and intuitive principles. First, the installed system should impose minimal overheads when no extensions are used. In Scruf's case, it should be usable without extensions, and performance should be very close to that of an `ext2` partition. Second, when extensions are used, the overheads should be negligible in comparison to the cost of executing the extensions themselves. For example, when using Scruf-Compress, the inherent overhead of decompressing the files should not be overshadowed by the overheads of the extension mechanism. A small relative overhead is tolerable by the user for the functionality provided by the extension.

As discussed earlier, there is significant concern about the overhead of running system extensions at the user-level, due to the amount of user-kernel communication and the expected high cost of IPC. In particular, Scruf is further complicated by the fact that its operation involves more interprocess communication than just the IPC between the kernel and user-level—since the user-level daemon must also communicate with the scripts. This double IPC pathway is worrisome to researchers and so it is important to know how much overhead it causes. IPC overheads have 3 parts:

1. Moving data between sender and receiver.

2. Context switching between and user and kernel level.

3. Synchronization between senders and receivers.

We mentioned earlier that a user can improve the performance of their Scruf extension by taking advantage of asynchronous, or non-blocking, scripts. Asynchronous scripts aim to reduce the overheads from synchronization as the daemon and file operations, blocked in the kernel, wait for return values from VFS trigger scripts. Asynchronous scripts are typically persistent and perform interposed actions. When they are invoked they inform the daemon, via the API, that it should assume a return code of `SCRUF_API_NOOP` and return that code immediately to the kernel. Synchronous (or blocking) scripts block the daemon to wait for its execution to complete and an API command to be returned. Only when the script has completed executing does control return to the daemon.

To test the IPC overhead of the Scruf framework, we test the latency for Scruf-VFS to send 100,000 messages up to the user-level daemon, which processes the messages and sends the data to a persistent script. We test both synchronous and asynchronous trigger scripts. This test is designed to stress the IPC pathway from the kernel to the scripts, which is illustrated in Figure 6.1. The test does not test the full OS pathway (Figure 5.4), and it does not involve file operations or disk activity other than for accessing the trigger script. The messages from Scruf-VFS do not come from any VFS file operations, but instead are created by code in an `ioctl()` function, written for this experiment, which only has to be invoked once, and can be invoked more directly from user-level than file operations can. This is done to get as true a measure of the IPC channel as possible. All the messages are the same operation. They mimic an `open` operation on the same file, so only one script is run to eliminate the overheads of script handling. The daemon is unmodified for the test, since the messages are exactly the same as messages from an actual Scruf-VFS open file operation. The daemon handles the `.on.open` script in the CWD as it would under normal operation. The on-open script used does nothing but count the messages, so the only overhead is the standard API parsing which any persistent script would need to do.

In the first test, the script is synchronous. When the daemon sends a message to a script, it must wait for a response from the script to return to the Scruf-VFS module. The response is always `SCRUF_API_NOOP`. In the second test, the script is asynchronous. When the daemon sends a message to the script, it does not wait and immediately returns a `SCRUF_API_NOOP` code to Scruf-VFS. The results are shown in Figure 6.2.

The sending and reply of 100,000 messages takes the synchronous script an average of 116.8 μs per message. Communicating with the asynchronous script took only 47.5 μs per message.

An overhead over 47.5 μs per message demonstrates that the IPC pathways are efficient

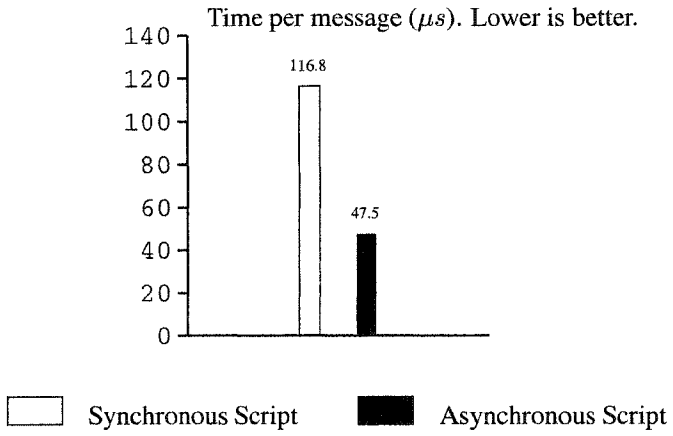

Figure 6.2: Synchronous v. Asynchronous Scripts: Sending 100,000 Messages

and are not a point of concern if poor performance is noted. A difference of 69.3 μs per message between synchronous and asynchronous scripts is an overhead of almost 150%. This result demonstrates that asynchronous scripts can be of significant benefit for applications with a large amount of file I/O. Our benchmark is rather synthetic, since realistic extensions perform non-trivial computations, which will add to the wait time for the daemon while it is blocked on a synchronous script. These extensions will see an even greater benefit from using asynchronous scripts, when possible.

6.1.2 Persistent Scripts

Another feature that improves performance is persistent scripts, which stay alive after their initial invocation to avoid the overhead of restarting the scripts every time that script's associated file operation occurs. To measure the performance improvement that persistent scripts can provide, we create a mock daemon that sends 100,000 messages to a script, which writes each access out to a log file. The purpose of the log file is to ensure that the scripts do simple, useful work. We use two scripts—one is persistent, the other is not. The non-persistent script must be re-started for each message and exits after it writes each message to the log file. The persistent message is invoked only once and remains running to avoid startup overheads. The average time for a non-persistent script to process an access is 1.835 ms. The persistent script averages 16.7 μs per access—a savings of two orders of magnitude.

While this demonstrates the utility of persistent scripts, two points must be made clear.

Stage	ext2	Scruf-Trace	Difference	Number of Log Entries	Main Source of Overheads
make dep	85.658	104.078	18.42 (21.5%)	241,650	Logging 2,300 entries / second
make bzImage	398.528	411.086	12.558 (3.2%)	391,938	Logging 950 entries / second

Table 6.1: Scruf-Trace vs. ext2 building the Linux 2.4.18 Kernel. Times given are in seconds.

First, process start up is very expensive. Second, for more computationally intensive scripts, persistent scripts will not be as great a benefit since the total runtime will be longer. Still, our conclusion is that persistent scripts are of significant benefit for trigger scripts associated with frequent file operations in programs with intensive file I/O.

6.2 Building the Linux Kernel

As previously mentioned, we have implemented Scruf-Trace to log file system operations. To test the overhead of Scruf-Trace we use it to log all opens, reads, writes, lseeks, and releases of files in a build of the Linux 2.4.18 kernel. We feel that Scruf-Trace is the best stress test of Scruf because of the amount of file I/O a large compilation such as the Linux kernel creates. Of course, Scruf-Trace is not representative of the kind of extended file system that most users would run on a regular basis. Systems like Scruf-CVS and Scruf-Trellis are closer to the day-to-day needs of users. We expect the Scruf-related overheads of those file systems to be less since they use fewer trigger scripts (Table 4.2), and they are less I/O intensive.

The Linux kernel compilation consists of two commands: **make dep** and **make bzImage**. **make dep** is run to derive dependencies for the build and does not compile any files, but creates files used in the actual build based on the configuration specified by the user. **make bzImage** begins the actual compilation of the kernel and culminates in the building of the boot image. For these experiments, the kernel is built with the default settings; no switches are touched during the configuration.

Table 6.1 contains the times for builds under both Scruf and unmodified ext2, the differences in time between the runs and the number of file accesses logged by Scruf-Trace. Each timing is the average of five runs. The **make dep** command reports a rather high overhead; however, during its execution Scruf-Trace is logging over 2,300 accesses per second. **make dep** does not contain a great deal of computation, but does involve a large

Phase	ext2	Scruf	Difference	Number of Log Entries	Main Source of Overheads
1	0.328	0.323	(-0.005) (-1.5%)	6	N/A
2	19.173	23.340	4.167 (21.7%)	60500	Checking for triggers in 400 new directories
3	51.585	51.430	-0.155 (-0.3%)	0	N/A
4	38.128	43.098	4.97 (13%)	55908	logging
5	344.437	365.922	21.485 (6.2%)	263100	logging

Table 6.2: Scruf-Trace vs. `ext2` under a modified Andrew Benchmark. Times given are in seconds.

amount of file I/O and this contributes to the higher percentage overhead. In contrast, **make bzImage** reports a much lower percentage overhead. This is because it is the compilation stage of the Linux kernel and so is less I/O intensive due to the computation from compiling; a much lower overhead is reported. Logging an average of 950 entries/second results in a much lower overhead of 3.2%. Overall, the overhead is quite reasonable given the high rate of event logging.

6.3 A Modified Andrew Benchmark

Our second benchmark is a modified Andrew benchmark, called Andrew100 [22]. Andrew100 is based upon the well-known modified Andrew benchmark [11, 20].

The modified Andrew benchmark is a five stage benchmark. Phase 1 creates a source directory tree. Phase 2 copies all the source files from an original source tree to the newly created tree. Phase 3 performs an `ls -l` command on the source tree. Phase 4 is a `grep` and `wc` on each file. Phase 5 is the computationally intensive build of the source tree. The modified Andrew benchmark is too small a benchmark on today's systems. To increase the load, Andrew100 makes 100 copies of the source tree in phases 1 and 2, and phases 3, 4, and 5 perform their tasks on all 100 copies.

For our experiment, only one set of trigger scripts is used for all 100 source trees. It is placed at the root of all 100 trees, i.e., two directories above the source trees themselves, to take advantage of Scruf's inheritance mechanism. The results are presented in Table 6.2.

Phase 1 is the creation of 110 directories for the 100 source trees. Phase 1 is short, and overhead is negligible since `mkdir` operations are not caught by Scruf and therefore operate as they would on `ext2`. Phase 2 is the copy phase which corresponds to copying

68 source files and header files for each instance of the Andrew source; 6800 copies are done in total. This stage is I/O intensive and also short, and these two factors contribute to a high percentage overhead for the logging. Phase 3 performs a `ls -l` on each file; the files themselves are not touched and so no trigger scripts are fired. This is why phase 3 sees negligible overhead. Phase 4 is a `grep` and `wc` on each file in the source tree. Phase 5 is the computationally intensive compilation of all 100 source trees.

The only phases which report significant logging and any overhead are phases 2 (the copy of the source tree), 4 (`grep` and `wc`) and 5 (the build of all 100 source trees). Phase 2 logs 2,600 accesses per second and, while the percentage overhead is high at 21.7%, it is tolerable due to the nature of the phase. To explain, copying generates a great deal of file I/O since each source file is opened, read, and released, and each destination file is opened, written, and released. The files are copied into 400 new directories which must have filters set up and trigger scripts located by inheritance. The runtime of the phase is very short and I/O intensive, and so the proportion of computation to I/O is low. However, the real time overhead of slightly more than 4 seconds is reasonable. Phase 4 has essentially the same file I/O as phase 2, as files are again opened, read, and released during the `grep` and `wc` operations; approximately 1,300 entries are logged each second. Again, the overhead is high, 13%, due to a relatively short phase time, but the percentage overhead is lower than phase 2 due to the computation of `grep` and `wc`. But as with phase 2, the real time overhead is short (under 5 seconds). Phase 5 is a large compilation and contains a more balanced mix of file I/O and computation, so the overhead is low at 6.2%. Phase 5 logs just over 700 accesses per second. This lower logging amount demonstrates the large amount of computation that is being performed.

Scruf-Trace is used in these benchmarks as a stress test of Scruf. The Scruf-Trace scripts are not as computationally intensive as compression or encryption. Scruf-Trace is therefore as close to an isolated measure of the overhead of the Scruf Framework as possible. The overhead shown on common operations is acceptable to obtain such a valuable resource as understanding the file I/O involved in any program. It also demonstrates that Scruf-Trace is usable for tracking in programs which have intensive file I/0. In both computationally intensive and short interactive usage Scruf-Trace provides great utility while keeping overheads low. Both benchmarks also use Scruf's inheritance mechanism, which does not have a noticeable impact on performance.

The Scruf-Trace script used in these two experiments is included in Appendix A.

	ext2	Scruf	Difference	Main Source of Overhead
make dep	85.658	86.01	0.352 (0.4%)	Filtering in Scruf-VFS
make bzImage	398.528	401.648	3.12 (0.8%)	Filtering in Scruf-VFS

Table 6.3: Scruf (no trigger scripts found) vs. ext2 building the Linux 2.4.18 Kernel. Times given are in seconds.

6.4 Running Scruf without Scripts

Scruf is designed to have overheads proportional to the complexity of the extended file system and the frequency of which triggers scripts are invoked. For example, if no extensions or trigger scripts are used, then the overhead should be minimal.

This benchmarks demonstrates that users can expect a negligible overhead when working in directories without scripts. For these benchmarks, the user-level daemon is still running, so the system is still checking for scripts, trying to inherit scripts, and applying filters in the kernel. We run the two benchmarks above without any scripts in place. The results of the Linux kernel are in Table 6.3 and those for the modified Andrew Benchmark are in Table 6.4.

Scruf never imposes an overhead of more than 1% when scripts are not present. From this result, we can conclude that applying filters and searching for VFS scripts to inherit has a negligible impact on performance.

This benchmark demonstrates that a user can install Scruf, use it simply as an ext2 replacement, and see virtually no overhead. Moreover, when an extension is in place, the overheads from the IPC and daemon are shown to be at a minimum (Section 6.1). Any substantial overheads will be the result of overheads from the scripts and the utilities they use (e.g., remote data copying (scp), CVS or encryption), and not that of the underlying system framework. Low framework overhead is an important property in that it lets users know that they do not pay a penalty for their extensions without seeing any real benefit in exchange for the overhead.

6.5 File Systems Which Use Redirection

Scruf-Trace is an example of an extension with interposed actions; its trigger scripts always return the SCRUF_API_NOOP code, and the file operations in Scruf-VFS proceed without any changes. The other operation trigger scripts can currently perform is redirection. Redirection causes Scruf-VFS to close the current file and open a different one. Two file system

Phase	ext2	Scruf	Difference	Main Source of Overhead
1	0.328	0.331	0.003 (0.91%)	Checking for triggers in Make directory
2	19.173	19.202	0.029 (0.15%)	Checking for triggers in 400 new directories
3	51.585	51.622	0.037 (0.07%)	N/A
4	38.138	38.241	0.103 (0.27%)	N/A
5	344.437	344.491	0.054 (0.01%)	N/A

Table 6.4: Scruf (no trigger scripts found) vs. ext2 under a modified Andrew Benchmark. Times given are in seconds.

extensions we have built using redirection are Scruf-Compress and Scruf-Crypt.

Unlike Scruf-Trace, these systems are less likely to be used in I/O intensive environments. Therefore, our benchmark to measure the overhead of redirection performs a typical user task within the compressed file system, Scruf-Compress. For this experiment, we have chosen the GhostScript utility [9] which is standard to any Unix platform. With this experiment we hope to demonstrate that the redirection mechanism will work with minimal overhead; it is important that it does not burden any applications, or it will be unusable in any practical sense.

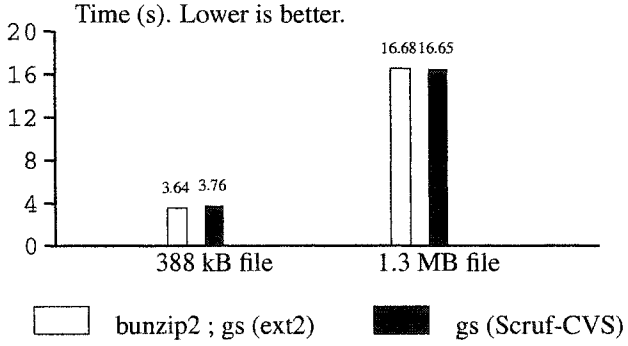

Figure 6.3: Scruf-Compress vs ext2: Running GhostScript on Compressed Files

The command that was used to test Scruf-Compress is

```
[cam@black]$ gs -dNODISPLAY -dNOPAUSE base.ps.bz2 -c quit
```

which simply processes the postscript files (which Scruf-Compress will uncompress first) and exits. The -dNODISPLAY option is added so that the translated postscript is not displayed, which it will do by default. The regular command line implementation that a user

	scp & bzip2	bzip2 & trellis	difference
50 MB file	95.24	95.35	0.11 (0.08%)
100 MB file	179.76	180.24	0.48 (0.26%)

Table 6.5: Remote Data Access using Trellis scripts. Times given are in seconds.

would be required to perform to get equivalent functionality on a regular `ext2` partition would be,

```
[cam@black]$ bunzip2 base.ps.bz2 -o basetemp.ps
[cam@black]$ gs -dNODISPLAY -dNOPAUSE basetemp.ps -c quit
[cam@black]$ rm basetemp.ps
```

The results are shown in Figure 6.3. Each test is the average of five runs with a low range of times. Scruf-Compress performs extremely well in both the short and medium length operations. On the small file, the overhead is only 0.12 seconds, or 3.3% of the total execution time. With the larger file, Scruf-Compress runs faster by 0.03 seconds, which is likely within noise limits. This difference is negligible and demonstrates that the overhead of Scruf is amortized across the larger runtime. These results demonstrate that the overhead of the redirection action of trigger scripts is almost negligible, and would not be worrisome to a user.

The scripts that are required for this experiment are `.on.open` and `.on.release`. An `.on.write` would have been required had the application modified the files, but GhostScript does not. The scripts are written in Python; `.on.open` is 38 lines long and `.on.release` is 36 lines long. The scripts for the more complex write-back version of Scruf-Compress (includes `.on.write`) are included in Appendix B.

6.6 Scruf-Trellis

To test the overhead of Scruf path scripts, we implemented a remote data access system called Scruf-Trellis. Our benchmark for this extension is to copy a remote file locally using `scp` and compress it. Primarily, it is a simple test of functionality. Note that `bzip2` is an unmodified binary application. We compare compressing a file with a path trigger script to a standard `scp` and compress operation. We use `bzip2` as our compression utility.

The straightforward Unix command line is:

```
scp sunset:/usr/sunset1/cam/mymail tmp ; bzip2 -c tmp
                                                > /dev/null
```

Using the Trellis path trigger script, the command is:

```
bzip2 -c /usr/scovil2/trellis:scp:sunset:/usr/sunset1/cam/mymail
                                                        > /dev/null
```

The results are presented in Table 6.5. The differences between Scruf-Trellis and its command-line equivalent are negligible. This demonstrates there is essentially no overhead from using the Scruf framework with path scripts, including IPC and script invocation, to implement simple remote data access. We hope to improve Scruf-Trellis to out-perform its command-line equivalent, with support for prefetching and overlapping of the remote copying and compression.

It should be mentioned that the Scruf-Trellis path script performs a check to ensure it does not redundantly copy a file which is already present on the local machine. It must do this specifically for `bzip2` because `bzip2` opens files it is compressing, twice. If the check is not done, the Trellis script will copy the file twice since each `open()` performs a lookup of the file and each lookup executes the path script. A similar check is necessary for applications which perform a `stat()` operation on a file's pathname before opening it, as the `stat()` function also causes a lookup to be performed.

6.7 Discussion

Our experimental results show that the basic idea of trigger scripts can be implemented in a real system. We also have shown that the inherent overheads of Scruf from IPC are low in our microbenchmarks and by running the system with no trigger scripts in place. We have also demonstrated that trigger scripts can be made very efficient by the use of asynchronous scripts, as well as persistent scripts, when possible.

Using Scruf-Trace as a stress test, we showed that the overheads associated with Scruf do not go above 22% of the real time. Moreover, for extensions which implement redirection of `open()` system calls such as Scruf-Compress, the overheads of the redirection are shown to be negligible and any performance degradation is a result of computations used in the extension—in the case of Scruf-Compress, the `bzip2` utility.

Finally, these experiments show that Scruf extensions are fully compatible with unmodified binary applications, such as `bzip2`, `gcc` and GhostScript.

Chapter 7

Concluding Remarks

Extending file system functionality has been and continues to be an active area of research in academia and industry. Users continue to look for a way to implement behaviours unique to their needs and are often faced with frameworks that require extensive understanding of low-level file system details, and long development time or installation. A solution to this problem must provide fast, flexible implementation and installation, that is simple and convenient to use. As well, the extensions must run efficiently with low system overhead.

The Scruf framework allows users to write extensions quickly, in a language of their choosing, to meet their file system needs.

7.1 Summary of the Scruf Framework

The Scruf framework creates a pathway from the kernel-level file system to the user-level. This pathway allows users to create file system extensions by writing user-level scripts, called *trigger scripts*. Trigger scripts run as normal, user-level programs that can monitor and modify the file system's operation to achieve new functionality. Since Scruf's extensions are run at the user-level, they have additional abilities, such as network access, which may not be possible or easy for regular kernel-level file systems in some operating systems. Scruf supports two kinds of trigger scripts: *VFS scripts* and *path scripts*, which have different semantics. VFS trigger scripts are invoked when associated file operations occur. Path trigger scripts are invoked when given in a pathname. Trigger scripts can be written in almost any programming language and are easily installed and removed so as not to require administrator or super-user privileges. Scruf extensions do not have to be installed system-wide, so only the individual user is affected by his or her extensions.

The performance of the Scruf framework is also encouraging. Overheads are kept to a minimum and any performance degradation of file system operation is a direct result of the

computation within the scripts, and not an overhead of the Scruf framework itself. When no trigger scripts are used, the overheads seen are less than 1% of execution time.

As part of this work, five example file system extensions—Scruf-Trace, Scruf-Compress, Scruf-CVS, Scruf-Crypt, and Scruf-Trellis—have been implemented using trigger scripts, and these extensions work with unmodified binary executables to extend the functionality of any user environment. They demonstrate that Scruf shows promise of becoming an extension framework for building many kinds of extensions which users have long sought in conventional file systems.

7.2 Future Work

Scruf is still in a prototype stage of development. While it has shown promise and usefulness as a framework for file system extension, a good deal of work remains to move it towards being a system ready for daily use. The most important issue at this point is making Scruf support multiple users. Related to this, the issue of security in using trigger scripts must be addressed. Currently, the Scruf framework could make a system prone to attacks. Also, the Scruf file system is not yet fully stable. Recovery from some types of failures has not been addressed. As indicated earlier, in some situations file operations may block indefinitely.

For Scruf to become a more powerful file system extension mechanism, the types of operations it can modify will need to be increased. VFS trigger scripts will need to be expanded to include more file operations—such as file creation, deletion and renaming—as well as directory operations such as `mkdir`. Finally, issues of user-friendliness will also need to be addressed in the future.

Appendix A

Scruf-Trace Script

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>

#define MAXARGS 1500

void printout(char ** args,const int count);

int main(void){

    char * args[MAXARGS];
    char header[32], output[128];
    int flag,count,numread;

    flag=0;
    count=0;

    printf("SCRUF_API_PERSIST SCRUF_API_ASYNC\n");
    fflush(stdout);

    while (1){
        scanf("%s",header);
        if (strcmp(header,"SCRUF_API_ARGS")==0){

            if (!flag){
                args[count]=(char *)malloc(128);
            }
            scanf("%s",args[count]);
            count=count+1;

            if (count==MAXARGS){
                flag=1; /* we have allocated the arrays once */
                printout(args,count);
                count=0;
            }

        } else if (strcmp(header,"SCRUF_API_KILL")==0){
            if (count>0)
                printout(args,count);
            break;
        }
    }
}

void printout(char ** args,const int count){

    int fd, wrote, i;
```

```
    char output[128*MAXARGS];

    fd=open("/usr/scovil3/cam/tapisLog",O_WRONLY|O_CREAT|O_APPEND);
    for (i=0;i<count;i++){
        args[i][strlen(args[i])+1]='\0';
        args[i][strlen(args[i])]='\n';
        wrote=write(fd,args[i],strlen(args[i]));
    }
    close(fd);
}
```

Appendix B

Scruf-Compress Scripts

B.1 The Scruf-Compress .on.open script

```
#!/usr/bin/python

import os
import sys
import string

# behaviour info: not persistent, wait for response
print 'SCRUF_API_PERSIST SCRUF_API_SYNC'
sys.stdout.flush()

while 1:
    line=sys.stdin.readline();
    args=string.split(line)
    if args[0]=='SCRUF_API_ARGS':
        indx=string.find(args[1],'.bz2')
        if (indx==-1):
            print '0'
            sys.stdout.flush()
            continue
        comma=string.find(args[1],',')
        if (comma!=-1):
            filename=args[1][:comma]
        else:
            filename=args[1]

        lastslash=string.rfind(filename,'/')
        if (lastslash==-1):
            thefile='.' + filename[:indx] + '.scruf.uncompress'
            os.system('bunzip2 -c ' + filename + ' > ' + thefile );
        else:
            thefile= '.' + filename[lastslash+1:indx] + '.scruf.uncompress'
            os.system('bunzip2 -c ' + filename[lastslash+1:] + ' > ' + thefile );
        print 'SCRUF_API_RELOAD ' + thefile
        sys.stdout.flush()
    else:
        if args[0]=='SCRUF_API_KILL':
            break
        print 'SCRUF_API_NOOP'
```

B.2 The Scruf-Compress .on.release script

```
#!/usr/bin/python

import os
import sys
```

```
import string

print 'SCRUF_API_PERSIST SCRUF_API_ASYNC'
sys.stdout.flush()

while 1:
    line=sys.stdin.readline();
    args=string.split(line)
    if args[0]=='SCRUF_API_ARGS':
        comma=string.find(args[1],',')
        if (comma!=-1):
            filename=args[1][:comma]
        else:
            filename=args[1]

        slash=string.rfind(filename,'/');
        if (slash!=-1):
            thefile=filename[slash+1:]
        else:
            thefile=filename

        suffix=string.find(thefile,'.scruf.uncompress')
        if (suffix!=-1):
            myx=os.access(filename + '.wb',os.F_OK);
            if (myx==1):
                os.system('rm -f ' + filename[1:suffix] + '.bz2')
                os.system('rm -f ' + filename + '.wb')
                os.system('bzip2 -c '+ filename  + ' >> ' +
                                    filename[1:suffix] + '.bz2' )
            os.system('rm -f ' + filename)
    else:
        if args[0]=='SCRUF_API_KILL':
            break
```

B.3 The Scruf-Compress .on.write script

```
#!/usr/bin/python

import os
import sys
import string

print 'SCRUF_API_PERSIST SCRUF_API_ASYNC'
sys.stdout.flush()

while 1:
    line=sys.stdin.readline();
    args=string.split(line)
    if args[0]=='SCRUF_API_ARGS':
        comma=string.find(args[1],',')
        slash=string.rfind(args[1],'/')
        if comma!=-1 and slash!=-1:
            filename=args[1][slash+1:comma]
        else:
            filename=args[1]

        if (string.find(filename,'.scruf.uncompress')!=-1):
            os.system('touch ' + filename + '.wb')
    else:
        if args[0]=='SCRUF_API_KILL':
            break
```

Appendix C

Scruf-Crypt Scripts

C.1 The Scruf-Crypt .on.open script

```
#!/usr/bin/python

import os
import sys
import string

#behaviour: non-persistent, don't wait

print 'SCRUF_API_NONPERSIST SCRUF_API_SYNC'
sys.stdout.flush()

line=sys.stdin.readline();
args=string.split(line)
if args[0]=='SCRUF_API_ARGS':
    indx=string.find(args[1],'.gpg')
    if (indx==-1):
        print '0'
        sys.exit(0)

    comma=string.find(args[1],',')
    if (comma!=-1):
        filename=args[1][:comma]
    else:
        filename=args[1]

    lastslash=string.rfind(filename,'/')
    if (lastslash==-1):
        returnfile='.' + filename[:indx] + '.scruf.decrypt'
    else:
        returnfile='.' + filename[lastslash+1:indx] + '.scruf.decrypt'

    os.system('echo thepassword | gpg --decrypt --output ' + returnfile +
                                  ' --passphrase-fd 0 ' + filename[lastslash+1:]);

    os.system('echo decrypted ' + filename + ' to ' + returnfile +
                                        ' >> /usr/scovil3/cam/tapisLog')
    print 'SCRUF_API_RELOAD ' + returnfile
```

C.2 The Scruf-Crypt .on.release script

```
#!/usr/bin/python

import os
import sys
```

```
import string

print 'SCRUF_API_NONPERSIST SCRUF_API_ASYNC'
sys.stdout.flush()

line=sys.stdin.readline();
args=string.split(line)
if args[0]=='SCRUF_API_ARGS':
    comma=string.find(args[1],',')
    slash=string.rfind(args[1],'/');
    if comma!=-1:
        filename=args[1][slash+1:comma]
    else:
        filename=args[1]

    suffix=string.find(filename,'.scruf.decrypt')
    if (suffix!=-1):
        myx=os.access(filename + '.wb',os.F_OK);
        if (myx==1):
            os.system('echo deleting ' + filename[1:suffix] +
                                                '.gpg for writeback')
            os.system('rm -f ' + filename[1:suffix] + '.gpg')
            os.system('rm -f ' + filename + '.wb')
            os.system('echo thepassword | gpg --output ' + filename[1:suffix]
                            + '.gpg --symmetric --passphrase-fd 0 ' + filename)
        os.system('rm -f ' + filename)
        os.system('echo deleting ' + filename + ' >> /usr/scovil3/cam/tapisLog')
```

C.3 The Scruf-Crypt .on.write script

```
#!/usr/bin/python

import os
import sys
import string

print 'SCRUF_API_NONPERSIST SCRUF_API_ASYNC'
sys.stdout.flush()

line=sys.stdin.readline();
args=string.split(line)
if args[0]=='SCRUF_API_ARGS':
    comma=string.find(args[1],',')
    slash=string.rfind(args[1],'/')
    if comma!=-1 and slash!=-1:
        filename=args[1][slash+1:comma]
        if (string.find(filename,'.scruf.decrypt')!=-1):
            os.system('touch ' + filename + '.wb')
            os.system('echo marking ' + filename + '
                        modified >> /usr/scovil3/cam/tapisLog')
```

Appendix D

Scruf-CVS Script

D.1 The Scruf-CVS .on.open script

```
#!/usr/bin/env python
# A sample .on.open script for using cvs

import os,sys,shutil,string


# behaviour info: non persistent, wait for response
print "SCRUF_API_NON_PERSIST SCRUF_API_SYNC"
sys.stdout.flush()

# execution
myline=sys.stdin.readline()
args=string.split(myline);
#if args[0]=='arguments':
#    print 'args[1] is ' + args[1]

cvsdir = os.getcwd() + "/CVS"
parse=string.split(args[1],',')
indx=string.rfind(parse[0],'/');
file=parse[0][indx+1:]
#file = sys.argv[len(sys.argv)-1]

# See if we're in a CVS controlled directory
if os.path.isdir(cvsdir) and os.path.isfile(cvsdir + "/Entries") and
    os.path.isfile(cvsdir + "/Repository") and os.path.isfile(cvsdir + "/Root"):

# Make a copy of the file to cvs commit
os.system("mkdir -p .cvs-tmp")
shutil.copyfile(file, ".cvs-tmp/" + file)
        print 'SCRUF_API_NOOP'
        sys.stdout.flush()

# cvs add the copy (add .cvs-tmp, even if it is already in cvs)
        # then commit the copy
        os.system("cvs -Q add .cvs-tmp 2> /dev/null")
os.system("cvs -Q add -m \"trigger script adding file\" .cvs-tmp/" + file)
os.system("cvs -Q commit -m \"trigger script committing changes\"" +
  " .cvs-tmp/" + file + " > /dev/null")

# Remove the copy
os.system("rm -f .cvs-tmp/" + file)
```

Appendix E

Scruf-Trellis Path Script

```
#!/usr/bin/python

import sys
import os
import string

if len(sys.argv)!=2:
    print 'USAGE: trellis <srcfile>'
    sys.exit(0)

comma=string.find(sys.argv[1],',')
if comma==-1:
    filename=sys.argv[1]
else:
    filename=sys.argv[1][:comma]

# create destination filename
dest=filename
for i in range(len(dest)):
    if dest[i]==':' or dest[i]=='/':
        dest=dest[:i] + '.' + dest[i+1:]

# check if file is already local
myx=os.access(dest,os.F_OK);
if myx==0:
    colon=string.find(filename,':')
    if colon!=-1:
        filename=filename[:colon] + ' ' + filename[colon+1:]
        # copy of file
        retval=os.system(filename + ' ' + dest)
        if retval==0:
            print 'SCRUF_API_RELOAD ' + dest
        else:
            print 'SCRUF_API_NOOP'
    else:
        print 'SCRUF_API_NOOP'
else:
    print 'SCRUF_API_RELOAD ' + dest
```

Bibliography

[1] A.D. Alexandrov, M. Ibel, K.E. Schauser, and C.J. Scheiman. Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems*, 16(3):207–233, 1998.

[2] AVFS: A Virtual File System. `http://sourceforge.net/projects/avf/`.

[3] B. N. Bershad and C. B. Pinkerton. Watchdogs - extending the unix file system. *Computing Systems*, 1(2):169–188, Spring 1988.

[4] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G.-R. Malan, and D. Bohman. Microkernel operating system architecture and Mach. *Journal of Information Processing*, 14(4):442–453, 1991.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, first edition, 2001.

[6] M. I. Bushnell. the HURD: Towards a new strategy of OS design. GNU's Bulletin, Free Software Foundation, 1994.

[7] Concurrent Versions System. `http://www.cvshome.org/`.

[8] The /dev File System. `http://www.atnf.csiro.au/people/rgooch/linux/docs/devfs.html`.

[9] Ghostscript, ghostview and GSview. `www.cs.wic.edu/ghost`.

[10] GnuPG (The GNU Privacy Guard). `www.gnupg.org`.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6(1)*, February 1988.

[12] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Symposium on Operating Systems Principles*, pages 52–65, 1997.

[13] Yousef A. Khalidi and Michael N. Nelson. Extensible file systems in spring. Technical report, Sun Microsystems, September 1993.

[14] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In USENIX Association, editor, *Summer conference proceedings, Atlanta 1986: June 9–13, 1986, Atlanta, Georgia, USA*, pages 238–247. USENIX, Summer 1986.

[15] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors.* PhD thesis, University of Toronto, 1994.

[16] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[17] Mac OS X. `www.apple.com/macosx`.

[18] S. J. Mullender. *The Amoeba distributed operating system: Selected papers 1984-1987*. Centrum voor Wiskunde en Informatica, Amsterdam, 1987.

[19] Ernesto Novillo, Paul Lu, and Cameron Macdonell. Program Debugging, Tuning, and Understanding Across Software and Hardware Layers . 2002.

[20] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, pages 247–256, 1990.

[21] PostreSQL. `www.postgresql.org`.

[22] Rodrigo Rodrigues, Barbara Liskov, and Miguel Castro. BASE: Using abstraction to improve fault tolerance. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 15–28, New York, October 21–24 2001. ACM Press.

[23] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 110–123. Kiawah Island Resort, near Charleston, South Carolina, December 1999.

[24] Jeff Siegel and Paul Lu. User-Level Remote Data Access in Overlay Metacomputers. In *4th IEEE International Conference on Cluster Computing (Cluster 2002)*, 2002. Poster paper.

[25] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of USENIX Annual Technical Conference*, 1999.

[26] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *Proceedings of USENIX Annual Technical Conference*, 2000.