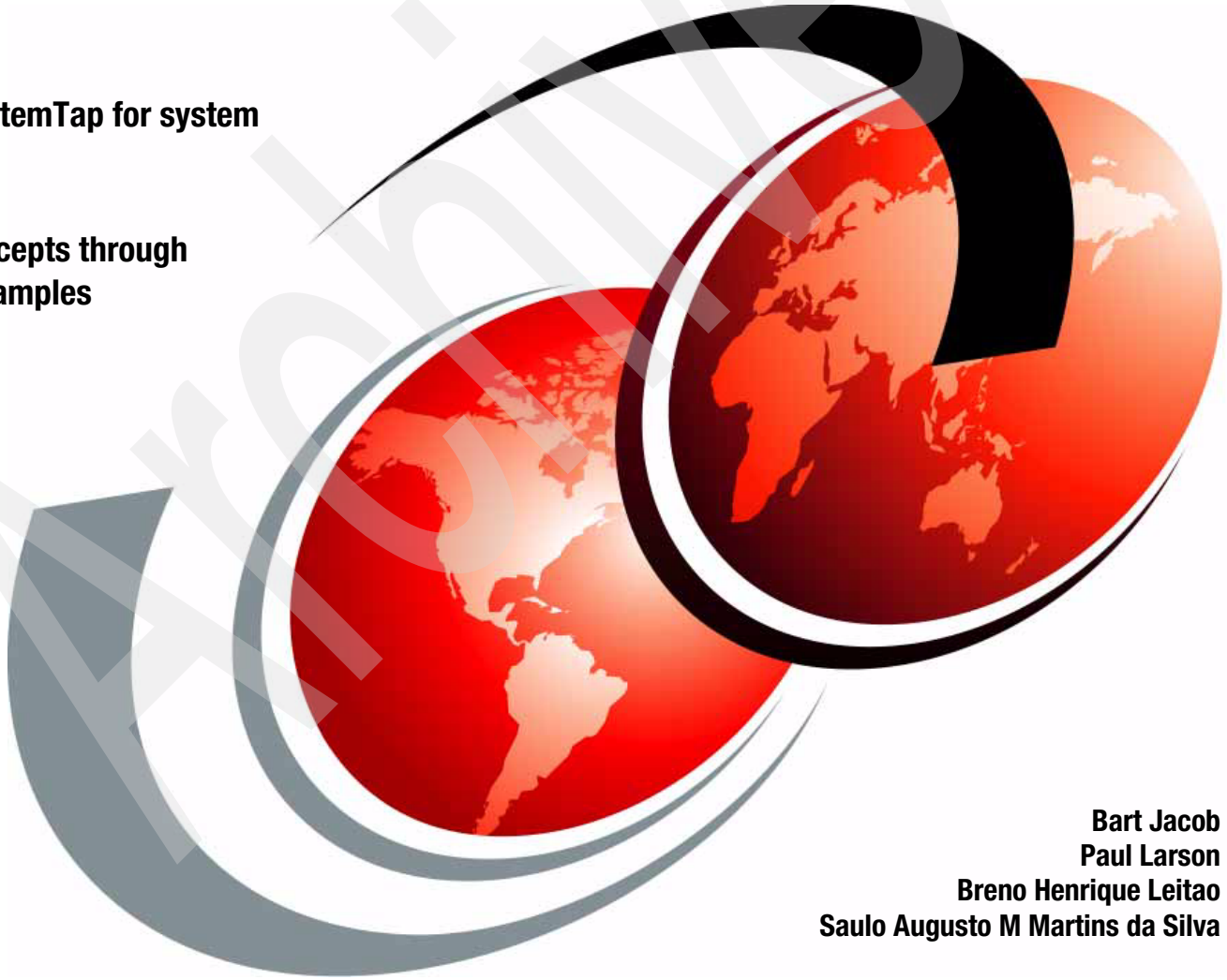
IBM

SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems

- Learn about SystemTap basics
- Utilize SystemTap for system analysis
- Learn concepts through simple examples



Bart Jacob
Paul Larson
Breno Henrique Leitao
Saulo Augusto M Martins da Silva

Redpaper

ibm.com/redbooks

International Technical Support Organization

SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems

January 2009

REDP-4469-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (January 2009)

This edition applies to SystemTap included with Red Hat Enterprise Linux V5.2 and SUSE Linux Enterprise Server Version 10.2.

This document created or updated on January 20, 2009.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

IBM®	Redbooks®	System z®
Power PC®	Redbooks (logo)  ®	

The following terms are trademarks of other companies:

Novell, SUSE, the Novell logo, and the N logo are registered trademarks of Novell, Inc. in the United States and other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

VMware, the VMware "boxes" logo and design are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions.

Intel, Intel logo, Intel Inside logo, and Intel Centrino logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® paper provides information related to the Linux®-based SystemTap tool. This tool can be used by developers, analysts and specialists to better understand and analyze Linux systems and applications. Until now, its primary use has been to look into the workings of the Linux kernel and various system calls. Its function is expanding to provide information related to user space as well, but that will not be the focus of this paper.

This paper describes the basic mechanisms used by SystemTap to gather data and offers guidance on getting started using the tool. Using examples, we show how to use the tool to capture and present useful information that may not be readily available by the myriad of other tools generally available on the Linux platform. Though our examples are relatively simple, they provide the basics to build on to develop more robust scripts that meet the reader's specific needs.

The reader will appreciate the power and simple elegance of SystemTap and how it can be used to help analyze Linux systems to identify functional and performance-related problems, which in turn can help ensure that system and application design will minimize the chances for problems in the future.

This paper is intended for individuals who have programming experience and familiarity with Linux and existing system facilities.

The team that wrote this paper

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Austin Center.

Bart Jacob is a Senior Consulting IT Specialist at IBM - International Technical Support Organization, Austin Center. He has over 25 years of experience providing technical support across a variety of IBM products and technologies, including communications, object-oriented software development, and systems management. Since joining the ITSO in 1989, he writes IBM Redbooks and creates and teaches workshops around the world on a variety of topics. He holds a master's degree in Numerical Analysis from Syracuse University.

Paul Larson is an Advisory Software Engineer at IBM in Austin, TX. He has been using Linux since 1993 and working in the Linux Technology Center at IBM since 2000. He holds a degree in Computer Science from Texas A&M Corpus Christi. His areas of expertise include Linux, testing, networking, and clustering.

Breno Henrique Leitao is a Software Engineer in Brazil. He has 11 years of experience in Linux and three years of experience working in kernel space. He has worked on LTC at IBM for two years. His areas of expertise include Linux, network and device drivers. He has a computer science degree from Universidade de Sao Paulo, and is pursuing his masters at Universidade de Campinas.

Saulo Augusto M Martins da Silva is an Infrastructure Specialist at IBM in Brazil. He has 12 years of experience in Linux Architecture and five years dedicated to Linux/390. Since joining IBM two years ago he has worked with Linux on System z performance and monitoring. He has a Computer Information System Degree from Faculdade de Ciencias Gerencias - UNA. He has RHCE certifications and LPI certification.

Thanks to the following people for their contributions to this project:

Jeffrey Borek
IBM Seattle

William Cohen
Red Hat, Inc.

Jim Keniston
IBM Beaverton

Ananth N Mavinakayanahalli
IBM India

Emily Ratliff
IBM Austin

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

1

Introduction

This chapter provides an introduction to SystemTap. We briefly describe what it is, how and where it is typically used, and a few basic concepts including safeguards related to its usage in a production environment. More technical details and examples can be found in later chapters.

1.1 What SystemTap is

SystemTap is a tool for the Linux Operating System that allows developers and system administrators to deeply investigate the behavior of the kernel and even user space applications in order to discover error conditions, performance issues, or just to understand how the system works. SystemTap is licensed under GPL and runs on a lot of architectures, such as x86, x86_64, Power PC®, and System 390. It also runs in virtual machines emulating any of these architectures. SystemTap was created by a consortium that includes Hitachi, IBM, Intel®, Oracle®, and Red Hat.

With SystemTap it is easy to summarize and visualize information related to the workings of your system, allowing you to easily get to the information required to address simple as well as complex issues.

SystemTap's main focus is to make it easy to capture and see the required data without requiring you to do a tedious kernel backup, modification, compile, install and reboot, which is often required when working with and debugging or analyzing kernel space.

SystemTap is a command line application that utilizes a plain text file (a script) as input and generates plain text output. The input file is a script created with the specific SystemTap language, which is similar to the C language. It is described in the next chapter. It basically follows an event-action structure. The script will describe what the tool should do in specific cases, for example, printing the process name every time a context switch happens. The script is mainly a set of subroutines and probe points. Probes will be described in more detail later, but think of them as break points, such as entry or exit from system calls. The script can gather and accumulate data and print formatted information to standard out.

Basically, SystemTap normally performs the following steps:

1. Translates the .stp script into the C source code (.c file) for a kernel module. This module implements the instrumentation specified by the script.
2. Compiles the C file to create the module (.ko file).
3. Loads the module into the kernel.
4. The module runs, reporting events of interest as specified by the script. Stap collects this information from the kernel and reports it to stdout. This is typically (almost always) plain text, but your script can generate binary records (using the printf %b format specifier) if that is what you want. There are even examples of passing binary code to a graphing tool.
5. The session ends and the module is unloaded when you send stap a CTRL-C, or the module decides it is done.

If you want, you can ask stap to save the .c file (from step 1) or the .ko file (from step 2), but we typically portray the output of stap as what is produced by step 4.

Basically SystemTap is a command line tool, but a graphical user interface is also available as a front end to work with SystemTap; it can be found at:

`http://stapgui.sourceforge.net`

SystemTap can be considered a high-level layer over kprobes when running on kernel space. However, SystemTap allows use of symbolic names and locations rather than the raw addresses of kprobes. Hence it has all the advantages provided by kprobes over an easy-to-use language to describe the handlers. See the description of kprobes in 2.1.1, "Kprobes" on page 6.

1.2 Purpose of SystemTap

Tracing events is a technique that developers use to understand an application's behavior. It is used because it is simple and easy, and provides a powerful way for understanding what logic is occurring and what variables and memory locations are changing. It is usually used when you do not have or cannot use an interactive debugger.

The main purpose of SystemTap is to trace events that happen on your system during a defined period of time. Trace, in this context, means calling a handler (which is basically a subroutine) as soon as an event occurs.

An event can be identified as any point at which your kernel can stop and call a handler. Examples of common events used with SystemTap include:

- Function enter and exit
- A timer expiring
- A network packet reception

A handler is typically used to collect and display critical data when the event occurs—function parameters, variable values, contents of memory locations, and so on. Data aggregation is also a very common use of SystemTap.

In short, this is a tool that can be used to better understand how an application works, and to identify the possible causes and locations of errors.

A typical use of SystemTap includes extracting data from event contexts, accumulating it in global variables and displaying it in a way that is most helpful to individuals performing the analysis.

1.3 Where SystemTap is used

An application that runs on Linux normally executes code in two sections called kernel space and user space. SystemTap is also unique for providing a unified view from user space and kernel space in the same script.

Historically, SystemTap provided little support for user space tracing. However, recent versions have added more robust support for user spaces. We do not say much about user space tracing in this paper, mainly because SLES 10 SP2 and RHEL5.2 do not yet support it. Just to provide a hint at how useful user space tracing will be as it is enabled for environments in the future, here are some situations where SystemTap can provide more help than other user application tracers and debuggers such as strace, ltrace, and gdb:

- Giving an integrated view of events in the kernel and selected user applications
- Probing of multi-threaded applications
- Probing multi-process applications—including client/server applications where the client and server processes execute independently
- Probing while running the application at or near full speed
- Surgical probing, where a hand-written SystemTap script can provide the required precision, in terms of events and functions probed and data reported, that strace and other similar tools cannot.

In current distributions, and as described in this paper, SystemTap is mostly used to debug the kernel, and is one of the most popular tracing tools used by kernel developers, mainly

because it is not intrusive and is easily configured. It does not require that the kernel be reconfigured, recompiled and reinstalled in order to capture data such as the value of specific variables.

Because of this characteristic, it is also used in production servers to help find errors that are not easily reproducible, in a transparent and non-disruptive manner. It does not require that the server be stopped or reinitialized.

1.4 Probing with SystemTap

In the context of SystemTap, probing means printing (or aggregating) debug information at specific points in executable code. For example, every time a function is called, you can display its arguments, or its return value. You can also limit the display of this data to when it is called in a specific context. When a function of interest is called, your script might not print anything at all, but simply update one or more counters or statistics. This aggregated information may be reported later (for example, when a timer probe expires or the session ends) or not at all.

Probing is the most common and easy way to discover a system's behavior and what is happening during the trace period. Most kernel functions can be instrumented, so that every time a function is called or returns, it calls your code to display or track the information you want.

Also, SystemTap provides a lot of useful built-in functions that help display the information in an easy-to-interpret way. This is very useful when working with statistics and performance issues.

1.5 Safeguards

Clearly, because SystemTap operates inside the Linux kernel through the use of loadable modules, safety is a concern. Because one of the goals of SystemTap is to help debug problems and create utilities for production systems, SystemTap was designed with several safety mechanisms. Common problems such as division by zero errors, and referencing bad memory locations, would normally be fatal to the system if they occurred in the kernel. SystemTap, however, catches these errors when the script is compiled, generates an error, and stops before the code is allowed to execute.

Memory access is carefully controlled by SystemTap, and explicit allocation is not allowed within a script. Other potential problems, such as infinite loops, are prevented by SystemTap. Recursion is limited to a predefined depth.

The safety mechanisms provided by SystemTap are there to help prevent SystemTap scripts from causing problems (for example, hangs or crashes) in production systems. For non-production systems, though, there is a way to remove many of the restrictions and write embedded C code, or even modify data in the kernel while it is running. This "guru" mode, which is discussed later, significantly enhances the power and flexibility of SystemTap at the cost of increased risk (if there is an error in the design or implementation of the script). Writing and the execution of scripts that run in guru mode should only be done by individuals who are already comfortable writing code that runs in the Linux kernel.

2

SystemTap technical overview

This chapter provides a more detailed technical overview of SystemTap, including:

- Architecture
- Using SystemTap
- SystemTap language

After a brief discussion of SystemTap architecture and usage, an introduction to the SystemTap language is provided. This chapter provides enough information to get started programming SystemTap scripts.

2.1 Architectural overview

SystemTap makes kernel debugging in Linux relatively easy, but relies on technology that preceded it to do the work. This section discusses the other technologies that are at the heart of SystemTap, and how SystemTap gets from a script written by a developer to a kernel module, to producing useful output.

The most common and well known facility used by SystemTap is Kprobes, which is described next. Another facility, which is growing in use, are kernel markers.

2.1.1 Kprobes

Kprobes is an API that allows developers to write Linux kernel modules to insert probes into a Linux kernel. Since these modules are dynamically loadable, Kprobes allows these probes to be inserted, and information to be gathered, without the need for traditional methods that required instrumenting code, recompiling the kernel, and rebooting the entire system. Kprobes is a powerful mechanism that allows a probe to be inserted at almost any address in the kernel. Handler code is specified that executes when the probe is hit.

Two types of Kprobes are utilized by SystemTap Kprobes and return probes.

Kprobe

A Kprobe is a general purpose hook that can be inserted almost anywhere in the kernel code. To allow it to probe an instruction, the first byte of the instruction is replaced with the breakpoint instruction for the architecture being used. When this breakpoint is hit, Kprobe takes over execution, executes its handler code for the probe, and then continues execution at the next instruction.

Return probes

A return probe, also called a *kretprobe*, attaches to the entry point of a function like a regular Kprobe. When the function is called, the return probe gets the return address and replaces it with a *trampoline address*. When the function exits, it returns to the trampoline address instead of where it was originally set to return, and the handler code for the probe is called. Return probes have access to the return values from functions. Figure 2-1 on page 7 provides a graphical representation of how a kretprobe works.

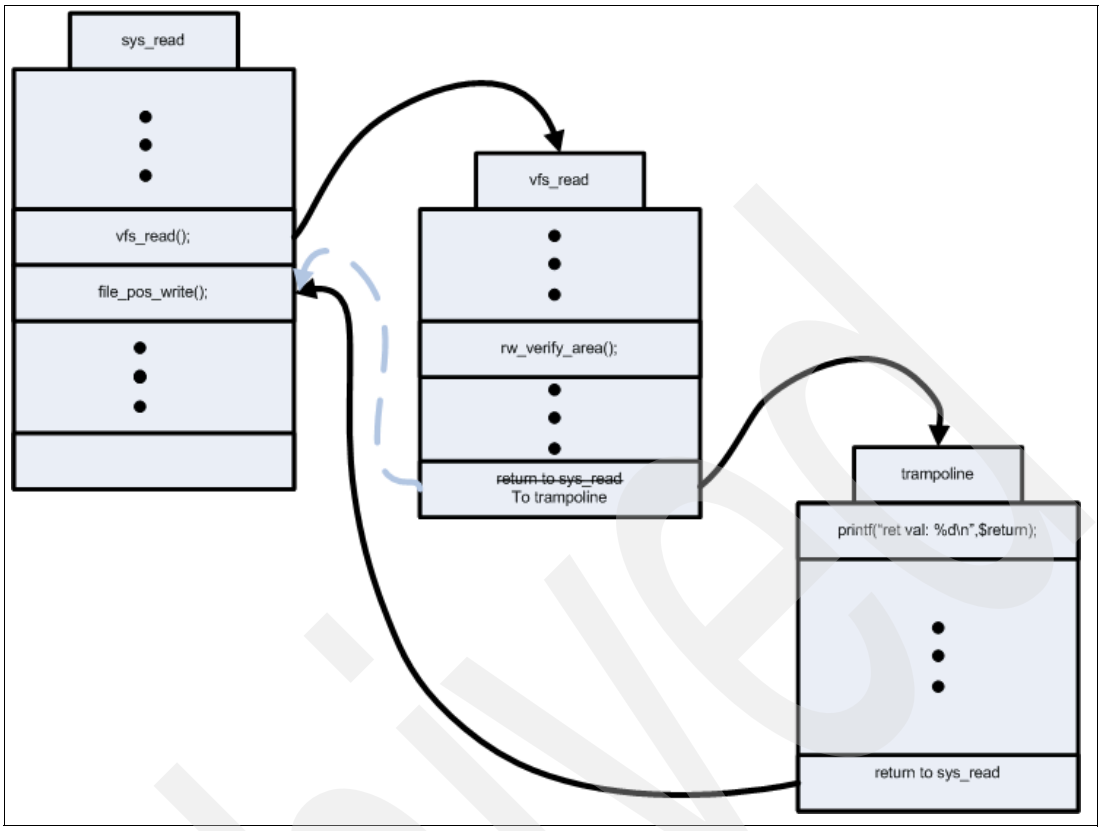


Figure 2-1 How a kretprobe works

2.1.2 Kernel markers

Kernel markers are special points at an important location inside the kernel that call hook functions every time the instruction pointer reaches that point/mark. A probe function tied to the marker is assigned at runtime and can have as many parameters as desired. Since it is easy to dynamically modify a kernel at any time, this provides a powerful method for debugging and analyzing system information from the kernel.

A marker, when not used, is automatically turned off, which virtually eliminates any negative effect on performance, so it is common to find systems running with markers enabled even when performance of the kernel is an important consideration.

2.2 SystemTap processing steps

In 2.1, “Architectural overview” on page 6, we described the underlying mechanisms that SystemTap utilizes to insert probes and gain control to execute user-defined logic based on a defined event.

The implementation of SystemTap is through the command **`stap`**. This command takes as input a file (script) that defines to SystemTap where to insert probes and what subroutines to execute when those probes are reached.

Although executing a script with SystemTap is as simple as calling the **`stap`** command with the script name to run, there are a great number of steps that SystemTap goes through in

order to check the script for errors, convert the script to a form that can be executed in the kernel, run the program, and generate output. The flow of these steps is shown in Figure 2-2.

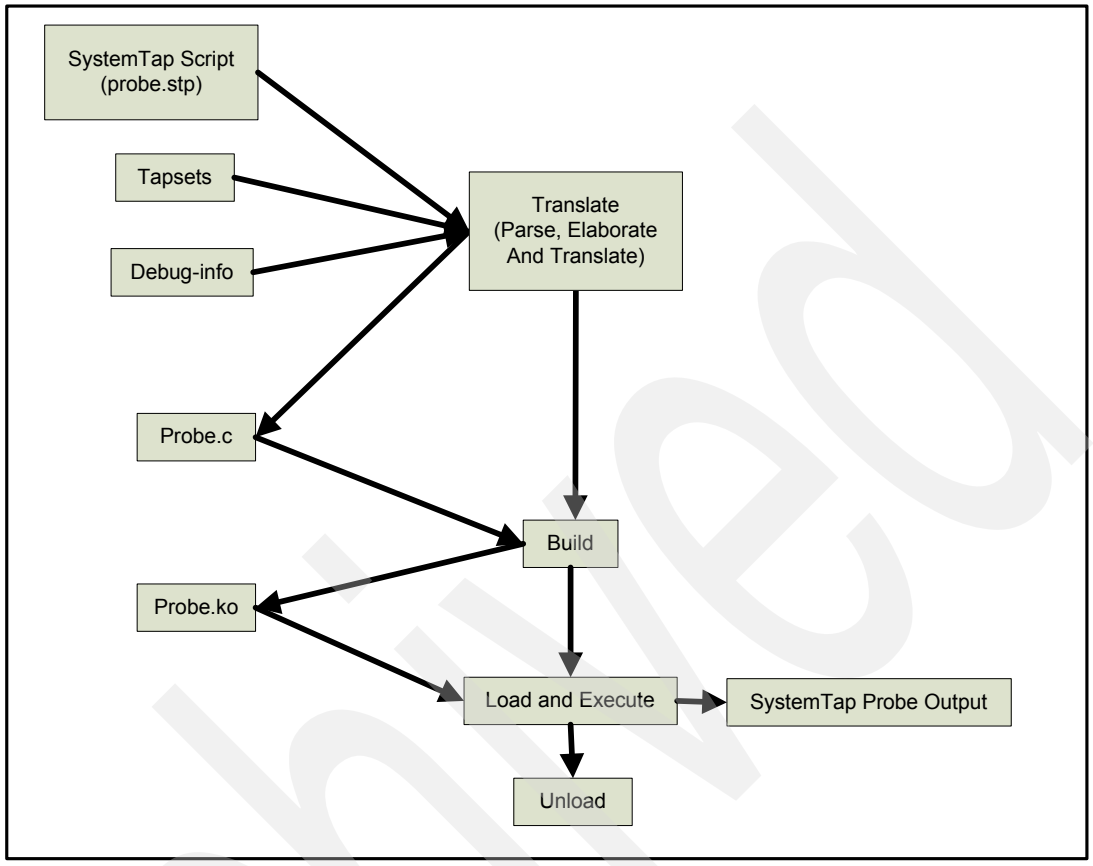


Figure 2-2 Logical flow of SystemTap processing

The following sections describe some of the key phases of this flow.

2.2.1 Translate

The translate step is really a three-part process: Parse, elaboration, and translation.

The first step when running a SystemTap script is parsing the input file (script). SystemTap code is similar in many ways to awk and C code, and is very simple to write. During the parsing of the code, it is represented internally in a parse tree. Preprocessing is performed during this step, and the code is checked for semantic and syntax errors.

During the elaboration step, the symbols and references in the SystemTap script are resolved. Also, any tapsets (see 2.6, “Tapset” on page 17) that are referenced in the SystemTap script are imported. Debug data that is read from the DWARF (a widely used, standardized debugging data format) information, which is produced during kernel compilation, is used to find the addresses for functions and variables referenced in the script, and allows probes to be placed inside functions. If the kernel is compiled without the debug data, SystemTap’s ability to properly probe the code will be limited.

The next step of the process takes the output from the elaboration phase and converts (translates) it into C source code. Variables used by multiple probes are protected by locks. Safety checks, and any necessary locking, are handled during the translation. During the

translation step, the code is also converted to use the Kprobes API for inserting probe points into the kernel.

2.2.2 Build

Once the SystemTap script has been translated into a C source file, the code is compiled into a module that can be dynamically loaded and executed in the kernel.

2.2.3 Execution

Once the module is built, SystemTap loads the module into the kernel. When the module loads, an init routine in the module starts running and begins inserting probes into their proper locations. Hitting a probe causes execution to stop while the handler for that probe is called. When the handler exits, normal execution continues. The module continues waiting for probes and executing handler code until the script exits, or until the user presses Ctrl-c, at which time SystemTap removes the probes, unloads the module, and exits.

Output from SystemTap is transferred from the kernel through a mechanism called relayfs, and sent to STDOUT.

2.3 Using SystemTap

The following sections describe how to run and utilize SystemTap scripts.

2.3.1 Running SystemTap scripts

Most of the time, a typical user of SystemTap simply calls the **stap** command, which is a front end for basically two commands (**stap** itself and **staprun**). **stap** itself is responsible for creating the kernel module with the functions described in the scripts. The other application is **staprun**, which is an application that loads the module generated by stap and runs it. Because a kernel module must be loaded, **stap** should be called from the root user, or using the **sudo** command. One could also add people to the *stapdev* and or *stapusr* group and not have them use the **sudo** command.

Executing a SystemTap script would normally look as shown in Example 2-1.

Example 2-1 Executing a SystemTap script

```
# stap helloworld.stp
Hello World.
```

Note: Typically, SystemTap scripts are given an .stp extension, but it is not necessary for them to have any particular extension.

Additionally, SystemTap scripts can be created so that they will automatically run when executed from the command line. To do this, the script should be made executable, and the following line should be added to the beginning of the script:

```
#!/usr/bin/stap
```

When a line like this is added to the beginning of the script, and the execute bit is enabled on the script, it can be invoked directly by executing the script. When this is called as a non-root

user, SystemTap will automatically try to perform the privileged operations as root through the **sudo** command. If that happens, the user may be prompted for his/her password, and the account must be enabled to allow **sudo** access.

2.3.2 Dissection of a SystemTap script

A SystemTap script is divided into two parts, the probe functions and functions. Probe functions are called whenever an event occurs, which implies that routines are event oriented. On the other side there are functions that are executed when explicitly called by some other function such as a probe subroutine.

Example 2-2 shows a simple example (note that the line numbers are not part of the script file; they are there to make the description of this script easier to follow).

Example 2-2 Simple example of an stap script

```
1  function is_open_creating:long (flag:long){
2  CREAT_FLAG = 4 // 0x4 = 00000100b
3  if (flag & CREAT_FLAG){
4     return 1
5  }
6  return 0
7  }
8
9   probe kernel.function("sys_open"){
10 creating = is_open_creating($mode)
11 if (creating)
12    printf("Creating file %s\n", user_string($filename))
13 else
14    printf("Opening file %s\n",
15                                 user_string($filename))}
```

Line 1 is declaring a function called `is_open_creating`, which will return an integer value. This function has one argument, which must be an integer also, and is called flag.

On line 2, a variable called CREAT_FLAG is created and is assigned the value 0000100b, which is the binary representation of decimal number 4.

On line 3, we check to see whether the parameter passed to the function has the CREAT_FLAG bit set, that is, 0000100b.

If so, it returns 1, on line 4. Otherwise, it returns 0 on line 6.

On line 9, we have defined a probe point at the system call sys_open (you could also use the syscall.probe probe point in this case).

On line 10, we create a variable called “creating” and assign the return of the function is_open_creating using the flag argument (from the sys_open call). The $mode variable contains the value of the mode argument passed to sys_open. Because we are probing the entry point of sys_open, all of the values passed to sys_open are available for examination in this way.

On line 11, the script checks to see whether `creating` is equal to 1. If so, it prints “Creating file” and the filename, also retrieved from the sys_open’s filename argument.

Otherwise, on line 14, it prints “Opening file” and the file name.

In summary, the script file shown in Example 2-2 defines a probe to be inserted whenever the sys_open function is called. The probe will call the simple function (is_open_creating) defined in the script and use its return value to determine if the sys_open call is opening an existing file or creating a new file and print this information to stdout.

2.4 SystemTap language

A programmer familiar with awk, C, or any other procedural language will have little trouble understanding the SystemTap programming language. It is not designed to be an extensive, general purpose programming language. However, its simplicity and familiarity make it well suited for the tasks that SystemTap is designed for. This section introduces the language and some of its key features, but does not try to be an exhaustive reference. See the complete language reference at:

http://sourceware.org/systemtap/langref/

2.4.1 Basics

SystemTap has the same basic capabilities of most modern programming languages, such as variables, arrays, conditionals, and loops.

Variables

Variable names in SystemTap must be alphanumeric, but they cannot start with a numeral. The names may also include dollar sign and underscore characters. The type of the variable is inferred by its first use. Variables may be declared at any point in a function or probe simply by using them (for example, myvar = 256). By default, variables are local to the function or probe in which they are defined. To declare a global variable, simply add ‘global’ to the beginning of the variable declaration, as follows:

```
global myvar=0
```

Global variables may be defined outside of probes and functions at any point in the script. They do not have to be defined at the beginning of the script.

Arrays

Unlike scalar variables, arrays may not be defined locally. All arrays must be defined as global variables. See Example 2-3.

Example 2-3 Example of declaring an array

```
probe begin {
   myarr[0]=1
   myarr[1]=2
   myarr[3]=4
   foreach(x in myarr) {
      printf("%d\n",myarr[x])
   }
}
global myarr
```

Notice that it is alright to skip indices in the array, but all elements must be of the same type.

If not specified, array capacity is typically limited to 2048. You may override this limit by specifying a new value for MAXMAPENTRIES on the command line. For a single array declaration, a new size may be specified as follows:

```
global mybigarr[20000]
```

For more information regarding changing the MAXMAPENTRIES value, see 6.3, “Dealing with skipped probes” on page 64.

Associative arrays, or hash tables, are also possible in the SystemTap scripting language. The index, or key, in an associative array is composed of one or more string or integer values.

Example:

```
arr1[“foo”] = 14
arr2[“coords”,3,42,7] = “test”
```

Array elements and entire arrays can be removed using the `delete` statement. This is useful for resetting arrays, elements, and even individual scalar variables back to zero or NULL before another iteration, such as for clearing statistics. Contrary to what the name implies, the variable is not actually deleted completely. Therefore, type information is still preserved, and the variable must continue to hold the same type of value (string or integer) that it was originally assigned.

Deleting an array:

```
delete myarr
```

Deleting an element

```
delete myarr[tid()]
```

Conditionals

Conditional expressions in SystemTap are very similar to C. The syntax is:

```
if(expression) statement [else statement]
```

The statements can be individual statements, or code blocks surrounded by braces.

Loops

Looping constructs in SystemTap are also very similar to C and other languages.

For loops

For loops are handled much as in the C language. The first statement in parentheses is executed one time at the beginning, the second is tested and the loop continues while it is non-zero, and the third is executed each time through the loop.

```
for(i=0;i<10;i++) { ... }
```

While

While loops also use a similar syntax to C:

```
while (i<10) { ... }
```

The expression for a while loop is tested at the beginning of each iteration. If it resolves to a non-zero value, then the statement, or statement block, is executed.

Foreach

The `foreach` loop statement is useful for creating a loop iteration for every element of an array. For example:

```
foreach (item in myarr) { myarr[item]++ }
```

In this example, `item` takes on the value of each key in the array, and increments to the next key for each iteration through the loop. Multiple variables can be used in associative arrays with multiple key values for each entry. See Example 2-4.

Example 2-4 Example of a foreach loop

```
foreach ([pid,procname] in processlist)
   printf("%s[%d]=%d\n", procname, pid, processlist[pid,procname])
```

Example 2-4 shows how more data can be stored in an array to distinguish elements from one another. For instance, there might be more than one process named ssh, but they would have a different process id. The `foreach` loop can iterate over the entire array, and use multiple key values.

2.4.2 Passing arguments from the command line

Arguments to SystemTap scripts can be passed on the command line and into the script when the **`stap`** command is invoked. However, the type of data being passed on the command line must be known.

Integer values passed to a SystemTap script can be referenced using `$N` where `N` is the argument number starting at 1.

String values passed to a SystemTap script use the slightly different notation `@N` where `N` is the argument number. Strings passed on the command line should be enclosed in double quotes if they contain spaces.

The following example shows a command line that passes a string and an integer on the command line:

```
stap script.stp sometext 42
```

The following line in a script would print these values on the screen:

```
printf("arg1: %s, arg2: %d\n", @1, $2)
```

Note: If an integer is passed and is read incorrectly as `@N`, then the `@N` variable will contain that integer represented as a string type. If a string is passed to the script and read incorrectly as `$N`, the `$N` variable will contain 0.

2.4.3 Statistics

SystemTap supports a feature that allows for the accumulation of data in aggregate variables. These aggregate variables are similar to arrays in the SystemTap language in that they must be declared as global variables. To add values to an aggregate variable, the aggregation operator <<< is used.

```
agg <<< 1
agg <<< 5
agg <<< 8
```

```
agg <<< 9
```

It is also possible to create arrays of aggregation variables in SystemTap. These arrays are created as global variables just like normal arrays. Instead of assigning values to elements of the array, though, the aggregation operator must be used instead.

```
aggarr[pid()] <<< 1
```

Data can later be extracted from the aggregation variable using the following data extraction functions:

- @count(var) - Return the number of values stored in the aggregation variable.
- @min(var) - Return the smallest value stored in the variable.
- @max(var) - Return the largest value stored in the variable.
- @sum(var) - Return the sum of all values stored in the variable.
- @avg(var) - Return the average (mean) of the values stored in the variable.

SystemTap also provides functions for generating ASCII graphical representations of the data stored in an aggregate variable.

The first of these histogram functions is called `@hist_linear` which takes as arguments the aggregate variable, the lower limit, the upper limit, and the spacing between each bar. For example, to print a graph starting at 0 and going to 10, with every number between represented (spacing of 1), the following instruction could be used:

```
print(@hist_linear(agg, 0, 10, 1))
```

Since the `@hist_linear` function returns a string, it must be printed manually. The output would look similar to Example 2-5.

Example 2-5 @hist_linear output

```
value |-------------------------------------------------- count
    0 |                                                    0
    1 |@@@@                                                4
    2 |@@@@                                                4
    3 |@@@@@@@@                                            8
    4 |                                                    0
    5 |@@@@@@@@@@@@@@@@                                   16
    6 |@@@@                                                4
    7 |@@                                                  2
    8 |@@@@@@@@                                            8
    9 |@@@@                                                4
   10 |@                                                   1
```

The other function provided by SystemTap for creating graphical output is called `@hist_log`, which is similar to `@hist_linear` in output, but only takes the array to be processed as an argument. The minimum and maximum values are calculated based on the data being printed, and the spacing between lines is automatically based on powers of 2. This can be especially useful for things such as memory, which already fit well into this model. Example 2-6 on page 15 shows an example.

Example 2-6 @hist_log output

```
value |-------------------------------------------------- count
    0 |                                                    0
    1 |@@@@                                                4
    2 |@@@@@@@@@@@@                                       12
    4 |@@@@@@@@@@@@@@@@@@@@@@@@                           22
    8 |@@@@@@@@@@@@@@                                     13
   16 |                                                    0
   32 |                                                    0
```

2.4.4 Predefined functions

SystemTap provides many functions useful for creating scripts for analyzing functional and performance problems. A few of those functions are introduced briefly here. Some are discussed later in this paper. For a full listing, see the SystemTap Language Reference.

- print(str) - Print the value of str.
- printf(fmt) - Print a string with formatting similar to the printf function in C.
- sprintf(fmt) - Return a string with formatting similar to the sprintf function in C.
- strlen(str) - Return the number of characters in str.
- isinstr(str1,str2) - Return 1 if the str1 contains the str2.
- strtol(str, base) - Convert a numerical value with the specified base stored in a string to an integer.
- exit() - Exit the running script.
- probefunc() - Return the name of the function being probed.
- execname() - Return the name of the current running process.
- pid() - Return the process id of the current process.
- uid() - Return the user ID of the current process.
- cpu() - Return the CPU number the current process is executing on.

2.5 Writing probes

Writing probes for SystemTap is fairly straightforward, but does require enough knowledge of the kernel to know which functions to trace. However, once you know the name of the function you want to trace, SystemTap allows the insertion of probes at various locations in that function in order to capture the desired data. SystemTap is also flexible enough to allow wild cards for probing all functions matching a certain pattern. In this section we discuss the types of probes available, and how to use SystemTap to probe specific functions.

2.5.1 Probing kernel and module functions

To probe a function in the kernel, the following syntax is used:

```
probe kernel.function("function_name"){ ... }
```

This will insert a probe that calls the code contained in the braces every time the kernel function `function_name()` is called. If that function is not found, then an error is returned at

compile time. This can be extended by adding `.call` at the end to specifically probe when the function is called, or `.return` for inserting a probe when the function exits. The `.call` and `.return` extensions are not valid for in-line functions.

To probe a function contained in a kernel loadable module, a slightly different syntax is used:

```
probe module("ext3").function("ext3_get*") { ... }
```

This example insert a probe for every function in the ext3 module beginning with "ext3_get" because of the wild card.

Probes may also be stacked so that all matching probe points will execute the same handler code, as shown by Example 2-7.

Example 2-7 Example of stacking multiple probe points

```
probe module("ext3").function("ext3_get*"),
      module("ext3").function("ext3_set*") {
   print("getting or setting something here\n")
}
```

This example executes the same handler code for any function in the ext3 module beginning with "ext3_get" or "ext3_set".

2.5.2 Optional probes

The previous probe examples generate an error if the specified function is not found. Sometimes, however, it is desirable to write a probe for a function that may or may not exist. This is particularly useful if the exact version of the kernel being analyzed is not known at the time the script is written, and must be written to accommodate multiple kernel versions since function names within the kernel can change from time to time. To indicate that a probe is optional, the probe definition should be followed by a question mark (?), as follows:

```
kernel.function("may_not_exist") ? { ... }
```

It is also possible to create an optional probe that stops processing the list of probe points as soon as one is satisfied. To indicate this type of optional probe, an exclamation point (!) is used, as follows:

```
kernel.function("this_might_exist") !,
kernel.function("if_not_then_this_should") !,
kernel.function("if_all_else_fails") { ... }
```

In this example, if the first probe point exists, then the probe is attached only to that point. If it does not exist, the second is tried. Only if the first two probe points do not exist, does it fall through and probe the final function in the list. This is because of the exclamation point character. If that character had not been used, it would try to attach a probe to all three functions, if they existed.

2.5.3 Conditional probes

A conditional probe will only be activated if a certain condition is met while the script is running. This conditional attached to the probe may be something that depends on a runtime value, such as an argument passed to the script. It may also be something that changes during the course of running the script, and the probe will activate once the condition becomes true.

Example:

```
probe kernel.function("some_func") if ( someval > 10)
```

2.5.4 Special probe points (begin, end, never)

SystemTap defines the following special probe points:

Begin

This probe point is executed at the beginning of the script. Any one-time setup functions should be called here. Other common uses are for printing header information, and printing a line to let the user know when the script is actually starting to run.

End

The end probe point happens at the very end of the script, when the script exits either by a call to exit() or by the user pressing Ctrl-c. The end probe point is often used for cleanup, and aggregation of data to report final values, or calculations based on data collected while the script was running, such as totals and averages.

Never

The never probe point will never be executed, but the code contained in it will still be checked by the compiler. This probe point may be useful when dealing with lists of optional probe points as a fail-safe.

2.6 Tapset

Tapsets in SystemTap are libraries of useful functions, and predefined probes that may be useful in writing scripts. They can serve to simplify scripts when the implementation details of gathering useful data from a particular kernel function are complicated. They can also be useful in situations where determining the correct probe point for a type of probe depends on things such as kernel version, architecture, or modules loaded. This complexity can be abstracted in a tapset, and used seamlessly in a SystemTap script.

Some tapsets are simply useful utility functions to help SystemTap developers. Many of the predefined functions listed in 2.4.4, “Predefined functions” are implemented in tapsets.

2.6.1 Including tapsets

Tapsets do not have to be explicitly included. The tapsets installed on a system can be found in the /usr/share/systemtap/tapset directory. This directory is scanned during the elaboration phase to locate tapsets for inclusion when **stap** is called in a script. If there are other directories that should be scanned, they may be specified using the **-I** option on **stap** when it is called.

2.6.2 Creating tapsets

Experts on a given subsection of the kernel may wish to write tapsets that expose useful data from the functions in that area. Writing tapsets is very similar to writing SystemTap scripts, except that they are intended to be used by other scripts, rather than encompassing the entire work of the script itself. Tapset files should usually be placed in /usr/share/systemtap/tapset/.

Tapsets can be used to create new probe points that are tied to one or more kernel functions. The code in the tapset can be used to create new variables containing values extracted from the kernel function. These new variables can be accessed by scripts using the tapset; see Example 2-8.

Example 2-8 Example of a tapset function

```
probe usb.submit_urb = kernel.function("usb_submit_urb")
{
    urb = $urb
    dev = $urb->dev
    flags = $mem_flags
}
```

A script calling this probe might look like Example 2-9.

Example 2-9 Example of a script calling this probe

```
probe usb.submit_urb
{
    printf("usb_submit_urb called on device at %x\n", dev)
}
```

A common thing for tapsets that include return functions is to convert the return value to a string. To do this, the function returnstr() is used. The returnstr() function will automatically find the return value when it is called from a return function probe, but needs to be told whether it should create a string with a decimal or hexadecimal value.

To use decimal, call it with an argument of 1, or call it with 2 for hexadecimal.

For example:

```
retstr = returnstr(1)
```

3

Installation

This chapter describes the installation of SystemTap. We cover three scenarios:

- RHEL5.2
- SLES10 SP2
- Installation from source

After the installation methods are discussed, we describe steps to verify that SystemTap installation was successful and that it is working properly. We also provide information on troubleshooting installation problems.

3.1 Installation on RedHat Enterprise Linux 5

To install SystemTap in a RedHat Enterprise Linux 5 environment, you need three primary packages in addition to the entire development packages group:

- SystemTap
- kernel-devel
- kernel-debuginfo
- Development packages

The easiest way to install it is with the yum utility. Here is an example:

```
# yum --enablerepo=rhel-debuginfo install systemtap kernel-devel
kernel-debuginfo
```

Example 3-1 shows the output of a SystemTap installation on RHEL5.2.

Example 3-1 Output of a successful SystemTap installation on RHEL5.2

```
Loading "rhnplugin" plugin
Loading "security" plugin
This system is not registered with RHN.
RHN support will be disabled.
rhel-debuginfo            100% |=========================|  951 B    00:00
Setting up Install Process
Parsing package install arguments
Resolving Dependencies
--> Running transaction check
---> Package systemtap.i386 0:0.6.2-1.el5 set to be updated
---> Package kernel-devel.i686 0:2.6.18-92.1.13.el5 set to be installed
---> Package kernel-debuginfo.i686 0:2.6.18-92.1.13.el5 set to be updated
--> Finished Dependency Resolution

Dependencies Resolved
...
Installed: kernel-debuginfo.i686 0:2.6.18-92.1.13.el5 kernel-devel.i686
0:2.6.18-92.1.13.el5 systemtap.i386 0:0.6.2-1.el5
Complete!
```

3.2 Installation on Novell SUSE Linux Enterprise Server 10 SP2

To install SystemTap on SLES 10 Service Pack 2, you need to install some packages that exist in Novell® repositories. In addition to the three basic packages, we need the development packages group:

- kernel-source
- kernel-debuginfo
- SystemTap
- Development packages

To install, you must have an account on some repository in order to download the debuginfo package.

Maybe your system is already configured to access an online repository. In this case, you can perform the installation using the zypper utility, as follows:

```
# zypper install kernel-source SystemTap kernel-debuginfo
```

Example 3-2 shows the output of a SystemTap installation on SLES10 SP2.

Example 3-2 A successful Installation on SLES10 SP2

```
Restoring system sources...
Parsing metadata for -20081020-162106...
Parsing metadata for SUSE Linux Enterprise Server 10 SP2...
Parsing RPM database...
Summary:
<install>   [S1:0][package]kernel-source-2.6.16.60-0.31.i586
<install>   [S2:1][package]systemtap-0.5.8-0.19.i586

Downloading: [S2:1][package]systemtap-0.5.8-0.19.i586, 640.3 K(2.0 M unpacked)
Installing: [S2:1][package]systemtap-0.5.8-0.19.i586
Downloading: [S1:0][package]kernel-source-2.6.16.60-0.31.i586, 45.1 M(228.6 M
unpacked)
Installing: [S1:0][package]kernel-source-2.6.16.60-0.31.i586
```

3.3 Installing without a repository

If you do not have an online repository configured, you can get the rpm files—probably from the DVD that you used to installed the system—and install the packages from there. For example:

```
rpm -i kernel-default-debuginfo-2.6.16.60-0.31.i586.rpm \
       kernel-source-2.6.16.60-0.21.i586.rpm  \
       systemtap-0.5.8-0.19.i586.rpm
```

If you do not see any error message, then the packages were installed correctly. No output is expected saying that the installation was successful.

3.4 Installation from Source

If you are running an upstream kernel, or a kernel that you compiled by yourself, then you will normally need to install SystemTap from the source code.

Installing SystemTap from source basically means two types of installation: From a source tarball, or from the git repository.

3.4.1 Compiling kernel to support SystemTap

If you are compiling a kernel by yourself and want this kernel to support SystemTap, you need to enable some features in the kernel. First, you will want to have debug information available, which implies compiling the kernel with the -g flag. You will also need to have Kprobe enabled and debugfs enabled. Enabling these capabilities is done through the config options in the kernel .config file, as shown in Example 3-3 on page 22.

Example 3-3 Options in .config file to enable kernel to support SystemTap

```
CONFIG_DEBUG_INFO
CONFIG_KPROBES
CONFIG_DEBUG_FS
CONFIG_RELAY
```

After enabling these options, just compile your kernel and install your kernel as usual. You can then run the SystemTap test suite to assure that everything is running fine.

3.4.2 Obtaining the SystemTap source code

If you want the latest (cutting-edge) version, then you will need to get the source from the git repository. This requires you to install the git application and clone the repository from the project page. The official project page is found at:

`http://sources.redhat.com/systemtap`

Using tarball

You can get the SystemTap source code from a tarball file that is available weekly at:

`http://sources.redhat.com/pub/systemtap/ftp/snapshots/`

Using the git repository

Git is the tool used by the SystemTap developers to manage the source code. So, using git you can get any SystemTap version, even the latest one, called upstream.

Cloning the source code (upstream version)

Once you have git installed on your system, you just need to clone the developers' repository to your machine. To do that, use the following command:

```
git clone git://sources.redhat.com/git/systemtap.git
```

Example 3-4 shows the expected behavior of a successful clone.

Example 3-4 Output when cloning the git repository for SystemTap source code

```
Initialized empty Git repository in /tmp/systemtap/.git/
remote: Generating pack...
remote: Done counting 21149 objects.
remote: Deltifying 21149 objects...
remote:  100% (21149/21149) done
remote: Total 21149 (delta 15659), reused 18266 (delta 13544)
Receiving objects: 100% (21149/21149), 5.16 MiB | 18 KiB/s, done.
Resolving deltas: 100% (15659/15659), done.
```

Cloning a specific SystemTap version

If you want a newer SystemTap version, but not the latest one, which can be less stable, you can get the last stable release.

To do so, just clone the repository as described above, then run:

```
# git tag -l
```

You will get the SystemTap versions/tags shown in Example 3-5.

Example 3-5 Output from git tag command

```
fc7-emerg-snapshot
release-0.6.2
release-0.7
release-0.7-rc1
release-0.7-rc2
rhel4u2b
rhel4u2b2
rhel4u2b3
rhel4u3
rhel4u4
rhel51b1
rhel52-062-1
rhel52-062-2
rhel5b2
rhel5rc
setuid-branchpoint
snapshot-20070903
systemtap-snapshot-20070908
systemtap-snapshot-20070915
```

Once you have decided what version best fits your requirements, just change your repository to that version, using the following command:

```
git checkout <tag_name>
```

After selecting the version, just compile the source code, creating the binaries for the specified version.

3.4.3 Compiling SystemTap

In order to compile the latest SystemTap, you need a recent version of the elfutils, which can be found at:

http://fedorahosted.org/releases/e/l/elfutils

Download the tar file and expand it into an appropriate directory. You won't need to perform a specific installation, just specify its directory during the SystemTap compilation process as shown in Example 3-6.

Example 3-6 Compiling SystemTap with elfutils

```
[root@boston SystemTap]# ./configure --with-elfutils=/root/elfutils-0.127
```

Once you have compiled SystemTap in this manner, you can install it using **`make install`**.

To uninstall it, use **`make uninstall`**.

3.5 Troubleshooting installation

Because of the manual steps described, and the possibility of errors occurring, you will want to verify that the installation has completed successfully and troubleshoot any problems that

occur. In this section we show how to handle problems in order to have a fully functional installation.

3.5.1 Verifying the installation

After you completed the various installation steps, we recommend that you check that everything is functioning as expected. To accomplish this, there are packages that check whether the installation is complete. Here is how to use them.

On RHEL

On RHEL5 run the commands shown in Example 3-7 in order to check whether the installation was successful.

Example 3-7 Installation check on RedHat Enterprise Linux 5

```
# yum install systemtap-testsuite
# cd /usr/share/systemtap/testsuite
# make installcheck
```

These commands run a series of SystemTap scripts and output a result displaying how many tests ran, how many were successful, and so on.

Example 3-8 shows an example of the output.

Example 3-8 Example of the output

```
Running /usr/share/systemtap/testsuite/systemtap/notest.exp ...
Running /usr/share/systemtap/testsuite/systemtap.base/add.exp ...
Running /usr/share/systemtap/testsuite/systemtap.base/alternatives.exp ...
....
# of expected passes            459
# of expected failures          171
# of unknown successes          2
# of known failures             4
# of untested testcases         19
```

Also, after the installation finishes, a file called systemtap.sum will be generated inside /usr/share/systemtap/testsuite, which will contain the output of the check, and display what failed and what passed.

If everything runs fine, you won't find any cases of unexpected failures or unexpected success. If you see any problems, you can look into systemtap.sum in order to find which test caused the error(s).

3.5.2 Common problems

We now show some of the more common problems found during SystemTap installation.

Missing debuginfo

If you try to compile a script and get a message such as `missing kernel 2.6.16.60-0.21-bigsmp i686 debuginfo`, this means that you need to install the kernel debuginfo on your system before running any SystemTap command. Refer to Example 3-9 and Example 3-10.

On RHEL

Example 3-9 shows the output on RHEL.

Example 3-9 Missing kernel error message on RHEL

```
semantic error: libdwfl failure (missing kernel 2.6.16.60-0.21-bigsmp i686
debuginfo): No such file or directory while resolving probe point
kernel.function("sys_open")
semantic error: no probes found
Pass 2: analysis failed. Try again with more '-v' (verbose) options.
```

On SLES

Example 3-10 shows the output on SLES.

Example 3-10 Missing kernel error message on SLES

```
semantic error: libdwfl failure (dwfl_linux_kernel_report_offline): No such file
or directory while resolving probe point kernel.function("sys_open")
Pass 2: analysis failed.  Try again with more '-v' (verbose) options.
```

Recent SystemTap on an old kernel

SystemTap and the kernel have a lot of common interfaces. If a new kernel breaks some interface, or even if SystemTap tries to utilize an API that the old kernel does not have, then you may see some strange results. You will need to match the kernel version with the correct SystemTap version. Example 3-11 shows the result of trying to compile a cutting-edge SystemTap using an old kernel.

Example 3-11 Possible error of a version mismatch between kernel and SystemTap

```
/usr/local/share/SystemTap/runtime/time.c:243: error: .on_each_cpu. undeclared
(first use in this function)
/usr/local/share/SystemTap/runtime/time.c:243: error: (Each undeclared identifier
is reported only once
```

Kernel binary and source version mismatch

After installing all required packages to start debug, when you run your first script you may see an `Invalid module format` message. This means that the binary kernel may be different from the kernel source.

To confirm this, just look at the dmesg log, and check whether it has a message saying that the struct_module disagrees about the version; see Example 3-12. If so, then you need to install the same source kernel version as the binary kernel you are running.

Example 3-12 Kernel binary and source version mismatch

```
Error inserting module
'/tmp/stapsPA2IX/stap_9a7fc312f2f332d24cffd6bd39769c7a_214.ko': Invalid module
format
Pass 5: run failed.  Try again with more '-v' (verbose) options.
[root@boston leitao]# dmesg | tail -n 1
stap_9a7fc312f2f332d24cffd6bd39769c7a_214: disagrees about version of symbol
struct_module
```

Missing libelf

If you try to compile SystemTap using an old elfutils version, you may get the error message shown in Example 3-13.

Example 3-13 Error message when compiling without the libelf

```
configure: error: missing elfutils development headers/libraries (install
elfutils-devel, libebl-dev, libdw-dev and/or libebl-devel)
```

To solve this error, recompile with the correct version of elfutils. See 3.4.3, “Compiling SystemTap” on page 23.

Cache issues

SystemTap uses a cache to save some steps if you run the same script more than once. This is a good thing, since it provides improved performance for your scripts.

A problem can occur when you have reconfigured your environment after having attempted to run a script using a wrong package. For example, if you had the wrong version of kernel-devel or kernel-debuginfo, and after seeing errors, you uninstalled these and installed the correct versions, your cache will not automatically be refreshed or deleted, and you will still see the same problem, even after fixing it.

So, every time you change your kernel binaries or source, you need to delete the SystemTap cache, which is located at $HOME/.systemtap/cache.

For more information, see “Kernel binary and source version mismatch” on page 25.

4

Using SystemTap to analyze functional problems

Functional problems, also known as bugs, are not unknown among developers and some of them are really hard to fix. The first step is to understand what the system is actually doing instead of what it should be doing.

This chapter introduces some new SystemTap functions and shows how to use them to analyze functional problems.

We cover several sample scenarios:

4.1 Generating a call graph

In order to debug functional problems in a section of code, it is often necessary to have an understanding of the flow of execution through the code. An easy way to get a quick understanding of the flow is by creating a call graph. SystemTap makes this very simple. The simple script in Example 4-1 displays a call graph.

Example 4-1 Creating a call graph

```
probe module("ext3").function("*").call {
   printf("%s -> %s\n", thread_indent(1), probefunc())
}
probe module("ext3").function("*").return {
   printf("%s <- %s\n", thread_indent(-1), probefunc())
}
```

This example will create a call graph of the ext3 module. Example 4-2 shows a partial example of the output from executing the script in Example 4-1 while running the Linux **ls** command.

Example 4-2 Partial output from Example 4-1

```
  0 ls(6218): -> ext3_permission
  3 ls(6218): <- ext3_permission
  0 ls(6218): -> ext3_dirty_inode
  3 ls(6218):  -> ext3_journal_start_sb
  8 ls(6218):  <- ext3_journal_start_sb
 12 ls(6218):  -> ext3_mark_inode_dirty
 16 ls(6218):   -> ext3_reserve_inode_write
 20 ls(6218):    -> ext3_get_inode_loc
 23 ls(6218):     -> __ext3_get_inode_loc
 28 ls(6218):     <- __ext3_get_inode_loc
 31 ls(6218):    <- ext3_get_inode_loc
 35 ls(6218):   <- ext3_reserve_inode_write
 39 ls(6218):   -> ext3_mark_iloc_dirty
 43 ls(6218):   <- ext3_mark_iloc_dirty
 46 ls(6218):  <- ext3_mark_inode_dirty
 49 ls(6218):  -> __ext3_journal_stop
 53 ls(6218):  <- __ext3_journal_stop
 55 ls(6218): <- ext3_dirty_inode
```

Notice that the thread_indent() function automatically takes care of adding and removing indentation, depending on the value of the delta passed to it. It also automatically prints the number of microseconds since the last initial level of indentation, the process name, and thread ID.

Another useful example for when you want to examine the interface between a module and the kernel is shown in Example 4-3. For example, what happens when you call the **ethtool** command for a specific driver.

This example shows the major functions of ethtool and tg3 used when you call a specific **ethtool** command for a specific driver.

Example 4-3 Interface between tg3 and ethtool subsystem

```
probe kernel.function("sock_ioctl"){
```

```
        printf("%s -> %s\n", thread_indent(1),probefunc())
}
probe kernel.function("sock_ioctl").return {
        printf("%s -> %s\n", thread_indent(-1),probefunc())
}

probe kernel.function("*@net/core/ethtool.c").call {
        printf("%s -> %s\n", thread_indent(1),probefunc())
}
probe kernel.function("*@net/core/ethtool.c*").return {
        printf("%s <- %s\n", thread_indent(-1), probefunc())
}

probe module("tg3").function("tg3_get*") {
        printf("%s -> %s\n", thread_indent(1),probefunc())
}

probe module("tg3").function("tg3_get*").return {
        printf("%s <- %s\n", thread_indent(-1),probefunc())
}
```

The output of this script is shown in Example 4-4.

Example 4-4 Output after "ethtool eth0"

```
0 ethtool(14590): -> sock_ioctl
     5 ethtool(14590):  -> dev_ethtool
     9 ethtool(14590):   -> tg3_get_wol
    14 ethtool(14590):   <- tg3_get_wol
    18 ethtool(14590):  <- dev_ethtool
    21 ethtool(14590): -> sock_ioctl
     0 ethtool(14590): -> sock_ioctl
     5 ethtool(14590):  -> dev_ethtool
     9 ethtool(14590):   -> tg3_get_msglevel
    13 ethtool(14590):   <- tg3_get_msglevel
    17 ethtool(14590):  <- dev_ethtool
    20 ethtool(14590): -> sock_ioctl
     0 ethtool(14590): -> sock_ioctl
     4 ethtool(14590):  -> dev_ethtool
     9 ethtool(14590):   -> ethtool_op_get_link
    12 ethtool(14590):   <- ethtool_op_get_link
    16 ethtool(14590):  <- dev_ethtool
    19 ethtool(14590): -> sock_ioctl
```

4.2 Retrieving function arguments and return values

Often it is useful to see the values that are passed to a function when it is called, and the return value of the function when it exits. Example 4-5 shows a simple example of how to do this.

Example 4-5 SystemTap script to print function parameters and return values

```
probe module("ext3").function("ext3_block_to_path") {
   printf("block_to_path(inode[%d,uid=%d,gid=%d], %d)\n",
```

```
      $inode->i_ino, $inode->i_uid, $inode->i_gid, $i_block)
}
```

Example 4-6 shows a sample output from that script.

Example 4-6 Sample output

```
block_to_path(inode[8,uid=0,gid=0], 23629)
block_to_path(inode[8,uid=0,gid=0], 23630)
block_to_path(inode[7227693,uid=0,gid=0], 0)
block_to_path(inode[3892451,uid=0,gid=0], 0)
block_to_path(inode[8012485,uid=500,gid=500], 0)
```

Example 4-7 shows another example that probes the mkdir system call (“sys_mkdir”), and prints the file the system is trying to create, the creation mode, and whether it was created or not (return value).

Example 4-7 Sample script for an mkdir probe

```
probe syscall.mkdir{
        printf("Creating directory %s using mode %d...", user_string($pathname),
$mode)
}

probe syscall.mkdir.return {
        if (!$return)
                printf("created\n")
        else
                printf("failed\n")
}
```

Example 4-8 shows the output of Example 4-7.

Example 4-8 Output of mkdir probe

```
Creating file foo using mode 511...failed
Creating file /tmp/foo using mode 511...created
Creating file /tmp/stapgIfyVs using mode 448...created
```

4.3 Generating a stack trace

Stack traces are useful for drilling down to understand the path that led to a certain section of code, or to a known error condition. Using SystemTap, it is possible to insert a probe at a critical point in the kernel, check that the error condition was hit, and generate a stack trace to assist in debugging the problem. The script is shown in Example 4-9.

Example 4-9 Script for generating a stack trace

```
probe kernel.function("path_get").return {
   if($return < 0) {
      printf("In process [%s]\n", execname())
      print_regs()
      print_backtrace()
      print("----------------------------------------\n")
   }
```

```
}
```

To test this example, a simple **ls** command is sufficient. Example 4-10 shows the output from the probe when the test is executed on ppc64.

Example 4-10 Sample output from the stack trace script

```
In process [ls]
NIP: C00000000012FA84 XER: 20000000 LR: C0000000001340B4 CTR: 0000000000000FED
REGS: c0000001f23976f0 TRAP: 0700
MSR: 8000000000029032 CR: 22000248
DAR: 0000000000000000 DSISR: 000000000ffdd058
 CPU: 0
 GPR00: C0000001F20D0E80 C0000001F2397970 C0000000007D0670 C0000001F23979E0
 GPR04: C0000001F2397B50 0000000000000001 C0000001F2397B50 0000000000000000
 GPR08: 000000000FFEF6D0 C0000001F10BB2E0 00000000FFFFFEFF 0000000000000323
 GPR12: 000000000E5E2A0B C000000000843480 000000000FFF059C 0000000000000000
 GPR16: 0000000000000000 0000000000000000 000000000FFF0008 0000000000000000
 GPR20: 000000000FFDD058 000000000FFF0944 0000000000000000 0000000000000000
 GPR24: 00000000FFFFFEFF C0000001DD5F0980 C0000001F23979E0 C0000001DD5F0980
 GPR28: C0000001F2397B50 C0000001F21D2B44 C00000000077E7E0 C0000001F2397970
NIP [c00000000012fa84] LR [c0000000001340b4]
 0xc00000000012fa84 : .path_get+0x0/0x90 [kernel]
[0xc0000001f23976f8] [0xc0000001f23979e0] 0xc0000001f23979e0 :
klist_next+0x1f1bcf388/0x0 [kernel] (unreliable)
[0xc0000001f2397970] [0xc0000001f2397a10] 0xc0000001f2397a10 :
klist_next+0x1f1bcf3b8/0x0 [kernel]
[0xc0000001f2397a20] [0xc000000000134394] 0xc000000000134394 :
.do_path_lookup+0x158/0x20c [kernel]
[0xc0000001f2397ae0] [0xc0000000001351cc] 0xc0000000001351cc :
.user_path_at+0x78/0xd0 [kernel]
[0xc0000001f2397c20] [0xc000000000122b20] 0xc000000000122b20 :
.sys_faccessat+0x100/0x22c [kernel]
[0xc0000001f2397d10] [0xc000000000122c88] 0xc000000000122c88 :
.sys_access+0x3c/0x58 [kernel]
[0xc0000001f2397da0] [0xc00000000001a7a8] 0xc00000000001a7a8 :
.compat_sys_access+0x38/0x58 [kernel]
[0xc0000001f2397e30] [0xc000000000008534] 0xc000000000008534 :
.start_here_common+0x198/0x354 [kernel]
--- Exception: c00 at 0x000000000ffd88c4
    LR =0x000000000ffc41ec

----------------------------------------
```

The probe prints out the process that was running when the error condition was detected, then prints the registers and a back trace of the functions running in kernel space before the error was hit.

4.4 Reading data at specific locations

While debugging a functional problem, it may also be useful to read the value of kernel variables or data structures at a specific location. Historically, this has almost always been accomplished with the printk() function. Now with SystemTap, probes can dynamically be

inserted into the running kernel without the need for a lengthy process of rebuilding the kernel; see Example 4-11. They can also be executed immediately on a running system while the problem is happening, rather than having to reboot into a debug kernel and reproducing the problem again.

Example 4-11 Script to display process, pid, and pid status

```
probe kernel.function("*@kernel/sched.c:686"){
        printf("Active process: %d\n", $array->nr_active)
        printf("Current pid %d in ", $p->pid)
        if (!$p->state)
                printf("runnable state\n")
        else if ($p->state > 0)
                printf("stopped state\n")
        else // -1
                printf("unrunable state\n")
}
```

Example 4-12 shows sample output from this SystemTap script.

Example 4-12 Sample output for the process status script

```
Active process: 0
Current pid 5463 in stopped state
Active process: 1
Current pid 15487 in stopped state
Active process: 2
Current pid 15487 in stopped state
Active process:
Current pid 15487 in stopped state
Active process: 0
Current pid 5353 in stopped state
```

4.5 Probing functions, unknown pointers

SystemTap allows you to explore functions even though you may not know their name. Often, you may have only a pointer to a function.

For example, in the net_dev structure, you have a lot of fields that just point to specific hardware functions. This scheme is used because those functions are hardware dependent and may get dynamically resolved based on the hardware in use. In order to allow a high abstraction level, there is just a common interface that calls the specific function, based on the kind of interface you are using.

So, calling net_device->open() will call the specific open() function for the net_device hardware. Using SystemTap, you could easily discover exactly what function or code segment is being called. You just need to create two simple scripts, as shown in Example 4-13 and Example 4-13 on page 33.

In the first example, we want to discover which function __netif_rx_action() is calling. So the first thing we need to discover is the address of the dev->poll() function. This can be accomplished with the script shown in Example 4-13.

Example 4-13 Script to obtain the address of the dev->poll() function

```
probe kernel.function("__netif_rx_schedule"){
   printf("Function address %x\n", $dev->poll);
   exit()
}
```

This script might produce the following output:

```
   Function address f89b8a1a
```

Next, we want to discover the name of the function that is at that address. To do so, we can probe at that point, and print the function name, as shown in Example 4-14.

Example 4-14 Script to print function name at probe point

```
probe kernel.statement(0xf89b8a1a).absolute {
        printf("%s\n", probefunc())
}
```

This function may print the following output, as an example:

```
   tg3_poll
```

The above examples could alternatively be implemented as shown in Example 4-15.

Example 4-15 Alternate implementation

```
global addr

probe kernel.function("__netif_rx_schedule"){
   addr = sprintf("%p", $dev->poll);
   exit()
}

probe end {
   printf("Function address and name are:\n")
   print_stack(addr)
}
```

We used the above scripts to discover that the dev->poll at __netif_rx_schedule() pointed to the tg3_poll function at the time we ran these scripts.

Using scripts such as these, you can easily check whether a net_device structure was filled correctly.

4.6 Signals

It is easy to handle signals with SystemTap. Many common functions are implemented in the tapset libraries, which enable the creation of a lot of complex scripts simply using library calls.

SystemTap allows you to probe at many signal events using signal.send, signal.handle, and signal.flush probe points.

Normally you will find these variables inside a signal probe point:

sig	The signal number
sig_name	The name of the signal probed
name	The function you're probing
sig_pid	The pid of the process sending the signal
si_type	The signal type
shared	Indicates if the signal is shared
pid_name	Name of the signal recipient process

Example 4-16 shows a simple example that shows all signals sent from any process to another, and whether the check was ignored or accepted by the application that received it.

Example 4-16 Script to display signal information

```
probe signal.send{
        printf("Signal sent %s from %s to %s\n", sig_name, execname(), pid_name);
}

probe signal.check_ignored {
        printf("Signal %s(%d) checked by %s...", sig_name, sig_pid, pid_name)
}

probe signal.check_ignored.return{
        if ($return)
                printf("Ignored\n\n")
        else
                printf("Accepted\n\n")
}
```

The output in Example 4-17 shows all signals sent by an application to another, and whether they were accepted or rejected.

Example 4-17 Sample output of the signal script

```
Signal sent SIGALRM from swapper to syslogd
Signal SIGALRM(4695) checked by syslogd...Accepted

Signal sent SIGINT from sshd to stapio
Signal SIGINT(8668) checked by stapio...Accepted

Signal sent SIGINT from sshd to staprun
Signal SIGINT(8661) checked by staprun...Ignored

Signal sent SIGINT from sshd to stap
Signal SIGINT(8660) checked by stap...Ignored
```

4.7 Probing TCP and UDP

There are a lot of tapset functions that help to work with protocols and sockets. The most used functions are those for when a TCP or UDP packet is sent or received. These probe points are:

- **tcp.recvmsg** - This is a probe point associated whenever you receive a TCP message.
- **tcp.sendmsg** - This probe point is called whenever you send a TCP message.
- **udp.sendmsg** - This probe point is called whenever you send a UDP message.
- **udp.recvmsg** - This probe point is called whenever you receive a UDP message.
- **tcp.disconnect** - When a TCP connection is disconnected.
- **udp.disconnect** - When a UDP connection is disconnected.

Inside these probe points you can usually find the following parameters:

- **sock** - The socket structure that carries the message
- **len** - The length of the message sent or received

Example 4-18 on page 36 shows a simple script for handling messages.

Example 4-18 Simple script for handling events related to messaging

```
probe udp.recvmsg {
        printf("%s: UDP: Receiving message. Socket %d. Size %d\n", execname(),
sock, size)
}

probe tcp.recvmsg {
        printf("%s: TCP: Receiving message. Socket %d. Size %d\n", execname(),
sock, size)
}

probe udp.sendmsg {
        printf("%s: UDP: Sending message. Flags %d. Size %d\n", execname(),
$sk->sk_flags, size)
}

probe tcp.sendmsg {
        printf("%s: TCP: Sending message. Flags %d. Size %d\n", execname(),
$sk->sk_flags, size)
}

probe udp.disconnect {
        printf("%s: UDP: Disconnected %d with flags %d\n", execname(), sock,
flags)
}
probe tcp.disconnect {
        printf("%s: TCP: Disconnected %d with flags %d\n", execname(), sock,
flags)
}
```

The output may display the following, as shown in Example 4-19.

Example 4-19 Sample output of the messaging script

```
sshd: TCP: Receiving message. Socket 3974402176. Size 16384
sshd: TCP: Sending message. Flags 776. Size 52
sshd: TCP: Sending message. Flags 776. Size 148
sshd: TCP: Sending message. Flags 776. Size 100
sshd: TCP: Sending message. Flags 776. Size 100
sshd: TCP: Receiving message. Socket 3974402176. Size 16384
sshd: TCP: Sending message. Flags 776. Size 52
```

4.8 Probing page faults

SystemTap has a tapset library with a set of probe points related to page faults. These can be very useful when digging into memory issues.

A page fault is, in most cases, an exception raised by the memory management unit, when a memory page is accessed, and this page is not available in the main memory. This is a common approach used by operating systems that support virtual memory. The exception is handled by the operating system itself, which has the goal of loading the data from the page file and placing it in memory.

Page faults degrade performance and can result in a condition known as system thrashing. This occurs when the system is using most of its cycles to only handle page faults. Optimization and page fault minimization improve performance, which is good for the entire system, not only for the current application. A very useful way to verify that your system is processing a lot of page faults is described here, and may help you to analyze (and improve) the memory management of software you are writing. It also could be used as a way to choose the best replacement algorithm.

The tapset is located at /usr/share/systemtap/tapset/memory.stp and contains all the probe points described here:

- **vm.pagefault** is a probe point whenever a process triggers a fault.
- **vm.write_shared** is fired when a process tries to write in a shared page.
- **vm.write_shared_copy** is fired when a process tries to write into a shared page that requires a page copy.
- **vm.munmap** is fired when a munmap is requested.
- **vm.brk** is called when the heap will be resized.
- **vm.iomm_kill** is called when a process is chosen by the OOM Killer.

Example 4-20 shows an example of the page faults that occur in the system, the type of page fault, and the process that caused the page fault.

Example 4-20 Sample script for viewing page fault information

```
probe vm.pagefault{
        printf("%s: Page fault on %p during a ", execname(), address)
        if (write_access){
                printf("write..")
        } else {
                printf("read..")
        }
}

probe vm.pagefault.return {
        if (fault_type == 0){
                printf("Out of memory\n")
        } else if (fault_type == 1){
                printf("Bad memory access\n")
        } else if (fault_type == 2) {
                printf("Minor fault\n")
        } else {
                printf("Major fault\n")
        }
}
```

Example 4-21 shows the output of this script.

Example 4-21 Sample output of the page fault script

```
stapio: Page fault on b7f70200 during a write..Minor fault
stapio: Page fault on 804c000 during a read..Minor fault
stapio: Page fault on ca5070 during a read..Minor fault
vim: Page fault on 8120520 during a read..Minor fault
vim: Page fault on 812bde0 during a read..Minor fault
vim: Page fault on 812c025 during a read..Minor fault
```

```
vim: Page fault on 8129650 during a read..Minor fault
automount: Page fault on b7f8b000 during a write..Minor fault
automount: Page fault on b7f8b000 during a write..Minor fault
automount: Page fault on b7f8b000 during a write..Minor fault
automount: Page fault on b7f8b000 during a write..Minor fault
```

Example 4-22 shows a modified version of Example 4-20 that helps ensure that messages are printed atomically. In the case where the process in fault is rescheduled, there can be multiple faults at the same time. Using the associative array makes sure that the output is printed as one unit.

Example 4-22 Modified script for showing page fault information

```
global current_fault

probe vm.pagefault{
  if (write_access){
    type="write.."
  } else {
    type="read.."
  }
  current_fault[tid()] = sprintf("%s: Page fault at %p on a %s",
          execname(), address, type)
}
probe vm.pagefault.return {
  print(current_fault[tid()])
  delete current_fault[tid()]
  if (fault_type == 0){
    printf("Out of memory\n")
  } else if (fault_type == 1){
    printf("Sigbus\n")
  } else if (fault_type == 2) {
    printf("Non-blocking operation\n")
  } else {
    printf("Blocking operation\n")
  }
}
```

4.9 Debugging NFS

SystemTap provides a lot of functions related to the Network File System (NFS), mainly when you are using NFS in a client or server side. The tapsets that contain NFS functions are found in /usr/share/systemtap/tapset with the following names: nfsd.stap, nfs_proc.stp and nfs.stp. You will find most of the NFS server functions in nfsd.stap, and most of the client probes in the nfs_proc.stp and nfs.stp files.

NFS server

Among the functions used to probe NFS servers, the most common are:

- **nfsd.dispatch:** Fired when the NFS server receives an operation from an NFS client.
- **nfsd.proc.lookup**: Fired when an NFS client searches or opens a file in the NFS server.

- **nfsd.proc.read**: Fired when an NFS client reads a file in the NFS server.
- **nfsd.proc.write**: Fired when an NFS client writes data to the NFS partition associated to the probed server.
- **nfsd.proc.commit**: Fired when a client commits an operation.
- **nfsd.proc.create**: Fired when a file is created by a client in the NFS partition associated with the probed server.
- **nfsd.proc.remove**: Fired when a served NFS file is removed by a client.
- **nfsd.proc.rename**: Fired when an NFS file is renamed by the client.
- **nfsd.open**: Fired when an NFS server opens a file.
- **nfsd.close**. Fired when an NFS server closes a file.
- **nfsd.read**. Fired when an NFS server reads a file
- **nfsd.write**: Fired when an NFS server write data to a file
- **nfsd.commit**: Fired when the server commits all pending writes to the disk.

NFS Client

Among the functions used to probe NFS clients, the most common ones are:

- **nfs.proc.lookup**: Fired when a client opens/searches for a file in the NFS server.
- **nfs.proc.read**: Fired when a client reads a file synchronously from the NFS server.
- **nfs.proc.write**: Fired when a client writes a file synchronously to the NFS server.
- **nfs.proc.commit**: Fired when a client writes the buffered data to the disk.
- **nfs.proc.read_done**: Fired when a read reply is received from the NFS server.
- **nfs.proc.write_done**: Fired when a write reply is received from the NFS server.
- **nfs.proc.commit_done**: Fired when a commit reply is received from the NFS server.
- **nfs.proc.open**: Fired when a file is being opened inside an NFS mounted directory.
- **nfs.proc.release**: Fired when a file is being released inside an NFS directory.
- **nfs.proc.create**: Fired when a file is being created in an NFS directory.
- **nfs.proc.remove**: Fired when a file is removed from the NFS directory.
- **nfs.proc.rename**: Fired when a file is renamed inside an NFS mounted directory.

Example 4-23 shows a simple example using some of these NFS client functions.

Example 4-23 Script tracking NFS client function calls

```
probe nfs.proc.open {
        printf("Opening file %s from server %x\n", filename, server_ip)
}

probe nfs.proc.write {
        printf("Writing to file %s on server %x\n", filename, server_ip)
}

probe nfs.proc.write_done {
        printf("Writing reply received from server %x. %d bytes wrote\n",
server_ip, count)
}

probe nfs.proc.read_done {
```

```
        printf("Read done from server %x\n", server_ip)
}

probe nfs.proc.remove {
        printf("Removing file %s\n", filename)
}

probe nfs.proc.create {
        printf("Creating file %s on server %x\n", filename, server_ip)
}
```

The output from this script may look similar to Example 4-24.

Example 4-24 Sample output from an NFS client script

```
Opening file  from server 91040309
Creating file myfile on server 91040309
Opening file myfile from server 91040309
Opening file foobar from server 91040309
Opening file foobar from server 91040309
Opening file foobar from server 91040309
Opening file foobar from server 91040309
Opening file myfile from server 91040309
Creating file .myfile.swp on server 91040309
Opening file .myfile.swp from server 91040309
Creating file .myfile.swpx on server 91040309
Opening file .myfile.swpx from server 91040309
Removing file .myfile.swpx
Removing file .myfile.swp
Creating file .myfile.swp on server 91040309
Opening file .myfile.swp from server 91040309
Writing reply received from server 91040309. 4096 bytes wrote
Writing reply received from server 91040309. 12288 bytes wrote
Opening file myfile from server 91040309
Writing reply received from server 91040309. 6 bytes wrote
Writing reply received from server 91040309. 4096 bytes wrote
Removing file .myfile.swp
Opening file foobar from server 91040309
Opening file foobar from server 91040309
Opening file foobar from server 91040309
Removing file myfile
```

5

Using SystemTap to analyze performance problems

This chapter provides guidance and examples for how to use SystemTap as a performance analysis tool and how to use functions and probes to get more information than many standard performance tools provide.

A performance analysis tool should provide useful information about how the system hardware is being used by the software and how the operating system is helping to manage that.

An IT professional responsible for monitoring and helping to ensure optimal performance should have knowledge and the ability to analyze the information gathered and provided by the performance monitoring tools, and to identify the system or application bottlenecks and propose alternatives to solve them. The professional may also provide detailed reports and information to other IT professionals such as developers, middleware support, system programmers and others to give them sufficient information to make changes in their areas of responsibility to help avoid performance related issues.

SystemTap uses Linux kernel function calls to generate data regarding the CPU, memory, and I/O for an entire system or for a specific process. The individual responsible for performance analysis and tuning can utilize SystemTap as a tremendous resource to gather the required data and generate the reports necessary to perform his role.

Also SystemTap is used as a low level kernel development performance tool that enables the developer and all involved in kernel design to get the best performance from functions, modules, and device drivers.

5.1 Probe points

SystemTap provides an easy way to probe after a specified time period. These probes are called asynchronous probes, which are called independently of the execution point within your code logic. That is, you do not set a break point at a particular call or instruction, but rather the probe is tripped based on time. So, if the timer period designated for your probe expires, then the kernel will stop what it is doing and will call your probe function.

The most common functions used for probe time are based on various standard time divisions as shown in the following function list:

- **timer.ms(N)** - A probe once every N milliseconds
- **timer.s(N)** - A probe once every N seconds
- **timer.us(N) -** A probe once every N microseconds
- **timer.ns(N)** - A probe once every N nanoseconds
- **timer.hz(N)** - N probes per second

Example 5-1 show a sample probe.

Example 5-1 Sample probe using clock times

```
probe timer.ms(500){
        printf("MS: %s\n", ctime(gettimeofday_s()))
}
probe timer.s(1){
        printf("S: %s\n", ctime(gettimeofday_s()))
}
```

Example 5-2 shows sample output from that probe.

Example 5-2 Sample output of time based probe

```
MS: Tue Nov 11 16:20:47 2008
MS: Tue Nov 11 16:20:47 2008
S: Tue Nov 11 16:20:47 2008
MS: Tue Nov 11 16:20:48 2008
MS: Tue Nov 11 16:20:48 2008
S: Tue Nov 11 16:20:48 2008
MS: Tue Nov 11 16:20:49 2008
MS: Tue Nov 11 16:20:49 2008
S: Tue Nov 11 16:20:49 2008
MS: Tue Nov 11 16:20:50 2008
MS: Tue Nov 11 16:20:50 2008
S: Tue Nov 11 16:20:50 2008
MS: Tue Nov 11 16:20:51 2008
```

You can also randomize the time intervals that the kernel will be interrupted to execute your probe function. To do so, just add the suffix .randomize(M) after your timer function name, for example, timer.s(N).randomize(M). The timer probe will then execute every M seconds plus or minus some random number between 0 and N. It is important to note that the random value changes at every cycle.

Example 5-3 shows an example of this.

Example 5-3 A probe executing every 6 seconds, plus or minus a random period of up to 5 seconds

```
global time

probe begin {
        printf("       Current time              Time Diference(ms)\n")
        printf("------------------------      ------------------------\n");
        time = gettimeofday_ms()
}

probe timer.s(6).randomize(5){
        printf("%s %20d\n", ctime(gettimeofday_s()), gettimeofday_ms() - time)
        time = gettimeofday_ms()
}
```

Example 5-4 shows the possible output.

Example 5-4 The output showing random probe times

```
Current time              Time Diference(ms)
------------------------      ------------------------
Fri Nov 14 11:58:07 2008                  1229
Fri Nov 14 11:58:12 2008                  4648
Fri Nov 14 11:58:17 2008                  5310
Fri Nov 14 11:58:26 2008                  9369
Fri Nov 14 11:58:27 2008                  1012
Fri Nov 14 11:58:37 2008                  9280
Fri Nov 14 11:58:45 2008                  7883
Fri Nov 14 11:58:46 2008                  1438
Fri Nov 14 11:58:56 2008                  9593
Fri Nov 14 11:59:05 2008                  9094
```

Jiffies

Sometimes you may want to specify a time period based on the cycle time of the system and not a specific time duration. In this case, you can use something called a jiffy instead of milliseconds or microseconds. A jiffy is a kernel internal value that measures an amount of time inside the kernel. It is defined by a compile-time variable called HZ in the kernel configuration. For example, on ppc64 this variable is set to 250 by default, which results in a jiffy that is equal to 4 milliseconds.

SystemTap provides a probe that interrupts the kernel after a given amount of jiffies and calls your specified routine. The probe is called as timer.jiffies(N), and will call your function after N jiffies.

Example 5-5 shows an example of using jiffies.

Example 5-5 Simple example of a jiffy probe

```
probe timer.jiffies(1000){
        printf("%s\n", ctime(gettimeofday_s()));
}
```

The output of this probe would look something like that shown in Example 5-6.

Example 5-6 Sample output of a jiffy probe

```
Tue Nov 11 15:49:25 2008
Tue Nov 11 15:49:26 2008
Tue Nov 11 15:49:27 2008
Tue Nov 11 15:49:28 2008
Tue Nov 11 15:49:29 2008
Tue Nov 11 15:49:30 2008
Tue Nov 11 15:49:31 2008
Tue Nov 11 15:49:32 2008
Tue Nov 11 15:49:33 2008
Tue Nov 11 15:49:34 2008
Tue Nov 11 15:49:35 2008
```

The randomize capability is also available for jiffy-based probe functions. Example 5-7 shows an example.

Example 5-7 Sample jiffy probe using random timing

```
global time

probe begin {
        printf("       Current time              Time Diference(ms)\n")
        printf("------------------------      ------------------------\n");
        time = gettimeofday_ms()
}

probe timer.jiffies(10000).randomize(8000){
        printf("%s %20d\n", ctime(gettimeofday_s()), gettimeofday_ms() - time)
        time = gettimeofday_ms()
}
```

Example 5-8 shows the output of that probe.

Example 5-8 Random Jiffies probes

```
Current time              Time Diference(ms)
------------------------      ------------------------
Fri Nov 14 11:32:46 2008                12241
Fri Nov 14 11:32:54 2008                 8072
Fri Nov 14 11:33:04 2008                10372
Fri Nov 14 11:33:09 2008                 4606
Fri Nov 14 11:33:22 2008                12970
Fri Nov 14 11:33:28 2008                 6041
Fri Nov 14 11:33:40 2008                12385
Fri Nov 14 11:33:44 2008                 3808
Fri Nov 14 11:33:53 2008                 8951
Fri Nov 14 11:34:00 2008                 6757
```

It is useful to use random times when normal timing does not work well; for example, when you have a predictable pattern of execution on your system, and you want to grab data from time to time without always getting the same result due to the pattern.

Profile probe

There is also a very useful asynchronous probe called timer.profile. Handler routines attached to this probe are executed on each CPU at every timer interrupt. This makes the timer.profile probe useful for writing performance profiling tools in SystemTap, because the handler can look at the process's state on every CPU at a high frequency.Example 5-9 shows a timer.profile probe.

Example 5-9 Example of timer.profile probe

```
probe timer.profile {
   printf("cpu: %d - process: %s\n",cpu(),execname())
}
```

The test system we used has two CPUs, so there will be two processors probed every time the timer.profile is hit, as shown in Example 5-10.

Example 5-10 Result of Example 5-9

```
cpu: 0 - process: sshd
cpu: 1 - process: find
cpu: 1 - process: find
cpu: 0 - process: sshd
cpu: 0 - process: swapper
cpu: 1 - process: find
cpu: 0 - process: sshd
cpu: 1 - process: find
cpu: 0 - process: sshd
cpu: 1 - process: find
cpu: 1 - process: find
cpu: 0 - process: sshd
```

To determine how often the timer.profile probe runs in a second, a simple script like this (Example 5-11) could be used.

Example 5-11 Example probe to print time between timer.profile probes

```
probe timer.profile { ptick++ }
probe timer.s(1) {
        stick++
        printf("%d\n", ptick/stick)
}
global stick, ptick
```

An x86 system with two processors showed the output in Example 5-12.

Example 5-12 Sample output

```
2002
2002
2001
2001
2001
```

In this case, the profile timer was running about 1000 times a second for each CPU. The output from the script must be divided by the number of processors in the system.

5.2 Timing function execution times

When working with kernel performance, you would like to see how long a function takes to execute. This can be very useful for identifying which functions may be the cause of a specific performance problem, or even to compare performance (benchmark) between two machines when focusing on specific functions.

This is easily done with SystemTap by simply getting the time when a function is called, when it exits, and printing the difference.

Example 5-13 is a simple example that measures how much time a system call takes to execute.

Example 5-13 Sample probe to measure duration of a system call

```
global time
probe syscall.open {
        time[tid()] = gettimeofday_ns()
}
probe kernel.function("sys_open").return {
        if(time[tid()]) {
                printf("sys_open took %d ns to execute\n",gettimeofday_ns() -
time[tid()])
                delete time[tid()]
        }
}
```

The output of the above probe is shown in Example 5-14.

Example 5-14 Sample output showing function execution duration

```
sys_open took 6002 ns to execute
sys_open took 5686 ns to execute
sys_open took 5628 ns to execute
sys_open took 5707 ns to execute
sys_open took 5362 ns to execute
sys_open took 6234 ns to execute
sys_open took 5777 ns to execute
sys_open took 11314 ns to execute
sys_open took 1091570 ns to execute
```

Using this example, you could filter the syscall you are most interested in, and print the arguments to help identify what may be causing a specific performance problem.

5.3 Using SystemTap to analyze system performance

As explained in 2.6, “Tapset”, there are functions and probes that can be used in SystemTap scripts to provide a wide range of data, and much of this can be useful when doing system performance analysis.

5.3.1 CPU performance

Data gathered using SystemTap related to the CPU can be used to understand under what conditions you might consider upgrading your server or adding additional CPUs to provide the required capacity.

The CPU performance numbers can be influenced by and also influence all other system resources. Processor usage is the basis of performance analysis.

With SystemTap you can obtain information about which processes were executing on each CPU. The following script will run at each CPU cycle (probe timer.profile) to collect the information about which process is currently running on the CPU, print how many times it was running on the specific CPU, and calculate the percentage of time it was running for the duration of the test. See Example 5-15.

Example 5-15 SystemTap script proc_cpu.stp

```
global one, two
global time

function div:string (a:long, b:long){
        total = a + b
        mod = (100*a)%total
        return sprintf("%d.%d%%", (100*a)/total, mod*100/total);
}

probe timer.profile {
        if (cpu() == 0)
                one[execname(), tid()]++
        if (cpu() == 1)
                two[execname(), tid()]++
}
probe begin {
        time = gettimeofday_s()

}
probe end {
        printf("%20s %5s %19s %19s\n", "process", "TID", "CPU0", "CPU1")
        foreach([p, t] in two){
                printf("%20s %5d %19s %19s\n", p, t, div(one[p, t], two[p, t]),
div(two[p, t], one[p,t]))
                delete one[p,t]
        }
        foreach([p, t] in one){
                printf("%20s %5d %18d%% %18d%%\n", p, t, 100,  0)
        }

        time = gettimeofday_s() - time
        printf("\nTotal.....................: %5d secs.\n",  time);
}
```

The output from this script is shown in Example 5-16.

Example 5-16 Output from proc_cpu.stp script

```
              process   TID                CPU0                CPU1
             swapper     0               49.19%              50.80%
              mysqld 23542                0.0%              100.0%
              mysqld 23551                0.0%              100.0%
              mysqld 23559                0.0%              100.0%
               httpd 23078                100%                0%
               httpd 23083                100%                0%
               httpd 23081                100%                0%
               httpd 23079                100%                0%
               httpd 23082                100%                0%
               httpd 23084                100%                0%
               httpd 23077                100%                0%
               httpd 23080                100%                0%
            pnmscale 23556                100%                0%

Total..........................:    65 secs.
```

An alternate script similar to the above is shown in Example 5-17, which provides more flexibility in the number of CPUs. For example, for a 4 CPU system you could run the script as follows:

```
stap proc_cpu1.stp 3
```

Example 5-17 SystemTap script proc_cpu1.stp

```
#Argument on line ($1) is the number of cpus - 1 in the system
global sample
global time*
probe timer.profile {
sample[execname(), tid()] <<< cpu()
}

probe begin {
   time = gettimeofday_s()
}
probe end {
   foreach([p, t] in sample){
      printf("process: %-20s TID: %5d\ncpu\tsamles\n", p, t)
   print (@hist_linear(sample[p,t], 0, $1, 1))
   }
   time = gettimeofday_s() - time
   printf("\nTotal..........................: %5d secs.\n",
   time);
}
```

As one example of how the above information could be useful, you may determine if any of an application's threads are moving from one processor to another during the execution period. If that occurs, the applications may degrade the overall system due to the processor switch. It is necessary to understand that if the same application is running in two processors that is fine,

but if the same thread from one application is running in both processors, that could be a problem.

Important: The swapper process is the process running in the CPU when it is idle.

5.3.2 Memory performance

Memory data collection can be quite simple but understanding its implications can sometimes be confusing. There are descriptions about used memory, free memory, buffer memory, cache memory, and in cached memory there are active and inactive spaces.

The Linux kernel monitors memory and takes advantage of it by using unused memory to do file system caching, for example. That should provide better response times for file access and overall system performance, but some applications that were not originally designed for Linux may not understand that file cached memory is available memory itself.

Example 5-18 Output of the free command

```
boston:~ # free
             total       used       free     shared    buffers     cached
Mem:       3099212    2007700    1091512          0     195880    1647812
-/+ buffers/cache:     164008    2935204
Swap:      1020116          0    1020116
```

Example 5-18 shows that from 3 GB of available memory, 2 GB is used, but of this approximately 1.6 GB of memory is used for file system cache.

For standalone systems that may be a good situation, but it is certainly not when you are using a virtualized environment such as Linux on IBM System z® or other environments such as Xen Hypervisor or VMware®. The cache memory could be used in other guest machines in the host environment

The Linux kernel has controls for how the cached memory is really used by the system, called active cached memory pages. Inactive memory space is the area where files are cached but is not used and it is the memory area that should be paged out or swapped out.

Example 5-19 Output of the /proc/meminfo file

```
boston:~ # cat /proc/meminfo
MemTotal:      3099212 kB
MemFree:       1091352 kB
Buffers:        196044 kB
Cached:        1647812 kB
SwapCached:          0 kB
Active:         619968 kB
Inactive:      1241104 kB
```

With a simple SystemTap script such as the example in Example 5-20, you may better understand how the memory is mapped. This example shows how much memory was requested by all mmap() function requests during a certain time interval. To better provide additional useful information, the processor ID and CPU that is used by the process are also included.

Example 5-20 SystemTap script proc_memory.stp

```
global process , uprocess

probe kernel.function("sys_mmap2") {
        process[execname(), ppid(), pid(), cpu()] += $len
}

probe kernel.function("sys_munmap") {
        uprocess[execname(), ppid(), pid(), cpu()] += $len
}

probe begin {
        printf("%20s %5s %5s %11s %3s\n","Process","PPID","PID","SZ(Kb)","CPU")
}

probe timer.s(5) {
        foreach ([exec+,pprocid,procid,cps] in process){
                sizeKB = process[exec,pprocid,procid,cps] / 1024
                printf("%20s %5d %5d +%11d %3d \n",
                    exec, pprocid, procid, sizeKB, cps)

        }

        foreach ([exec+,pprocid,procid,cps] in uprocess){
                sizeKB = uprocess[exec,pprocid,procid,cps] / 1024
                printf("%20s %5d %5d -%11d %3d \n",
                      exec, pprocid, procid, sizeKB, cps)
        }

        delete process
        delete uprocess
}
```

As explained, information provided by this simple SystemTap script gives you the exact size of memory requested (+ signal) and released (- signal) by the application (Example 5-21) during a certain interval when the mmap function was used. The **delete** command was used to empty the variable every 5 seconds.

Example 5-21 Sample output

```
             Process  PPID   PID      SZ(Kb) CPU
      httpd2-prefork 14362 12724 +      17454   0
      httpd2-prefork 14362 12720 +        776   0
              mysqld  5465  5509 +       1644   0
             staprun 25964 26063 +          4   0
      httpd2-prefork 14362 12724 -      17970   0
      httpd2-prefork 14362 12720 -        776   0
              mysqld  5465  5509 -       1644   0
      httpd2-prefork 14362 12720 +      24232   0
              mysqld  5465  5509 +       2916   0
      httpd2-prefork 14362 12720 -      24748   0
              mysqld  5465  5509 -       2916   0
```

In analyzing system performance, this information can be useful for determining how much memory is mapped per specific application. If you know the application and process involved in your system, you can filter that using the pid() or ppid () function or execname() to get the real states of your application in a certain period of time. That will help you with background knowledge about how the application and/or middleware works to size the correct memory requirements for your application.

5.3.3 Disk storage device performance

Disk device data collection is quite difficult to obtain because of the diversity of the type of disk devices, access methods, storage equipment, and so on. With SystemTap you can determine how much data is requested for read/write for a certain application; see Example 5-22.

Example 5-22 proc_device.stp

```
global ioreq, allreq, starttime

probe ioblock.request{
        ioreq[gettimeofday_s(),execname(),devname,rw] <<< size
        allreq++
}

probe begin {
        starttime = gettimeofday_s()
        printf ("%24s %20s %6s %10s %10s %10s\n",
            "Time","Process","Device","# Requests", "Read KB","Write KB")
}

probe timer.s(5) {
    foreach ([time,exec,dev,rw] in ioreq) {
        date = ctime(time)
        ionum = @count(ioreq[time,exec,dev,rw])
        if (rw == 0 ) {
                read = @sum(ioreq[time,exec,dev,rw])
        } else {
                write = @sum(ioreq[time,exec,dev,rw])
        }
        printf ("%10s %20s %6s %10d %10d %10d \n",
              date,exec,dev,ionum,read,write)
    }
    delete ioreq
}


probe end {
       printf("Number of IO Block Request in a %d sec. : %d\n",gettimeofday_s() -
starttime,allreq)
}
```

The output from example 5-21 running on a Linux on System z in an Apache Web server results in several read requests and some write requests. This information may help you to determine what is the best elevator algorithm, the best file system type, and options for a specific server and application; see Example 5-23.

Example 5-23 Partial output from proc_device.stp script

```
                      Time              Process Device # Requests    Read KB   Write KB
Thu Nov 13 20:30:18 2008               pdflush dasdb1        127          0     520192
Thu Nov 13 20:30:17 2008             kjournald dasdb1        652          0    2670592
Thu Nov 13 20:30:23 2008                mysqld dasdb1          1       4096          0
Thu Nov 13 20:30:23 2008             kjournald dasdb1          6       4096      24576
Thu Nov 13 20:30:23 2008                mysqld dasdb1          4       4096      16384
Thu Nov 13 20:30:27 2008             kjournald   dm-0         10          0      40960
Thu Nov 13 20:30:27 2008             kjournald dasdf1         10          0      40960
Thu Nov 13 20:30:28 2008             kjournald dasdb1          6          0      24576
Thu Nov 13 20:30:27 2008             kjournald dasdc1          5          0      20480
Thu Nov 13 20:30:27 2008             kjournald dasdd1         16          0      65536
Thu Nov 13 20:30:27 2008             kjournald   dm-1         21          0      86016
Thu Nov 13 20:30:32 2008                mysqld dasdb1         42          0     172032
[...]
Thu Nov 13 20:31:38 2008               pdflush dasdg1          4          0      16384
Thu Nov 13 20:31:43 2008               pdflush dasdb1         26          0     106496
Thu Nov 13 20:31:43 2008             kjournald dasdb1          1          0       4096
Thu Nov 13 20:31:43 2008             kjournald dasde1          1          0       4096
Thu Nov 13 20:31:43 2008               pdflush dasde1          4          0      16384
Thu Nov 13 20:31:43 2008               pdflush dasdd1          9          0      36864
Thu Nov 13 20:31:43 2008               pdflush dasdg1          8          0      32768
Thu Nov 13 20:31:43 2008               pdflush   dm-1         23          0      94208
Thu Nov 13 20:31:48 2008               pdflush dasdc1          5          0      20480
Thu Nov 13 20:31:48 2008               pdflush dasdf1          4          0      16384
Thu Nov 13 20:31:48 2008               pdflush   dm-0         23          0      94208
Thu Nov 13 20:31:48 2008               pdflush dasdg1          3          0      12288
Thu Nov 13 20:31:48 2008               pdflush dasdd1          3          0      12288
Thu Nov 13 20:31:48 2008               pdflush dasde1          8          0      32768
Number of IO Block Request in a 107 sec. : 2943
```

Note: The kjournald process is a thread responsible for managing the journaling of an ext3 file system. The pdfush is the thread that flushes the data in memory to the disk.

You can also work with SystemTap scripts to determine what files are being accessed from the disks or memory.

It is often useful to know which application opens which file in order to debug what is going on, and also to measure system performance, mainly when file lock is a big offender for your performance.

In this example, we probe the open syscall, and save the time a file was opened by some process such as the file descriptor number. Then, when the file is closed, we print the process name that opened that file, the file name, and the time that the file remained open; see Example 5-24.

Example 5-24 A script that shows how long a file was opened by a process

```
probe syscall.open.return{
        open[execname(), $return] = gettimeofday_us()
        names[execname(), $return] = user_string($filename)
}

probe kernel.function("sys_close"){
```

```
        printf("%20s %30s %5d\n", execname(), names[execname(), $fd],
                        (gettimeofday_ns() - open[execname(), $fd]))
}
```

Example 5-25 displays the result of Example 5-24.

Example 5-25 Sample output

```
              find                          irq   2350
              find                          irq    410
     hald-addon-stor                    /dev/hdc  43000
               gpm                     /dev/tty0    960
     hald-addon-stor                    /dev/hdc  42920
               gpm                     /dev/tty0   1200
             crond                          cron   4140
             crond                 /etc/cron.d    2130
     hald-addon-stor                    /dev/hdc  42960
               gpm                     /dev/tty0    880
```

With a few changes to the script, you have the example of Example 5-26, which shows file opens, but in this case there is a condition that only prints the files that are opened for more than one second.

Example 5-26 SystemTap script files_opens.stap

```
global open , names

probe begin {
        printf("%10s %12s %30s\n","Process" , "Open time(s)" , "File Name")

}

probe kernel.function("sys_open").return{
        open[execname(),task_tid(task_current()),$return] = gettimeofday_us()
        names[execname(),task_tid(task_current()),$return] = user_string($filename)
}

probe kernel.function("sys_close"){
        open_time_ms = (gettimeofday_us() - open[execname(),task_tid(task_current()), $fd])
        open_time_s = open_time_ms / 1000000
        if ((open_time_s >= 1) && (names[execname(),task_tid(task_current()), $fd] != "")) {
                printf("%10s %6d.%.5d %30s\n", execname(),
open_time_s,open_time_ms%1000000,names[execname(),task_tid(task_current()), $fd])
        }
}
```

Example 5-27 shows the output from Example 5-26.

Example 5-27 Output from files_opens.stp script

```
   Process     Open time(s)   File Name
     httpd      1.102703 /opt/g2data/cache/derivative/0/1/18.dat
     httpd      1.702867                 /tmp/phpv7HFif
     httpd      1.40255                 /tmp/php1S490a
     httpd      1.226289           /opt/g2data/locks/22
```

```
    httpd      1.389015           /opt/g2data/locks/15
    httpd      1.475688                    /etc/my.cnf
    httpd      1.370549                    /etc/my.cnf
```

Example 5-27 shows the Apache httpd system running in a normal environment. Two files with more than 2 MB were uploaded to the server; see Example 5-28.

Example 5-28 Partial Output from files_open.stp script

```
   Process     Open time(s)                       File Name
     httpd      1.497017                  /tmp/phpZ37pB7
     httpd      1.382449                  /tmp/php4zUFvP
     httpd      1.274013            /opt/g2data/locks/34
     httpd      1.462496            /opt/g2data/locks/15
     httpd      1.576966                     /etc/my.cnf
     httpd      1.439219                     /etc/my.cnf
     httpd      1.740506                  /tmp/phpnhkyki
     httpd      2.637305 /var/www/html/gallery2/main.php
     httpd      1.760644 /var/www/html/gallery2/bootstrap.inc
   httpd    2.179127 /var/www/html/gallery2/modules/core/classes/Gallery.cla
ss
   httpd    3.810221 /var/www/html/gallery2/modules/core/classes/GalleryData
Cache.class
     httpd      9.716467 /var/www/html/gallery2/config.php
     [...]
     httpd      9.491108 /opt/g2data/cache/module/core/0/0/0.inc
   httpd    3.741714 /var/www/html/gallery2/modules/core/classes/GalleryStor
age.class
     httpd     10.453927 /var/www/html/gallery2/lib/adodb/adodb.inc.php
     httpd      1.979278 /var/www/html/gallery2/lib/adodb/adodb-time.inc.php
     httpd     14.182344 /var/www/html/gallery2/lib/adodb/drivers/adodb-mysqli.inc.php
    [...]
        dd     99.545759                    /dev/zero
        dd     99.345874                    swap.file
        dd     99.347393                    swap.file
     httpd      1.833663 /opt/g2data/cache/entity/0/0/6.inc
     httpd      3.851490 /var/www/html/gallery2/modules/core/classes/GalleryEntity.class
     httpd      1.397644 /var/www/html/gallery2/modules/core/classes/GalleryPersistent.class
     httpd      1.216459            /opt/g2data/locks/40
     httpd      1.508141            /opt/g2data/locks/15
     httpd     25.649404                     /etc/my.cnf
     httpd      1.450782                     /etc/my.cnf
```

The second output from files_open.stp shows longer than average times for files staying open when a process, such as dd, is performing a lot of I/O to the same device.

Note: The **dd if=/dev/zero of=swap.file bs=1024 count=1512000** command was issued to simulate the I/O performance at the devices.

5.3.4 Network performance

This is a special script that records how long a poll section takes on a tg3 card. It mainly records the time a poll() function is called, and when it returns; see Example 5-29.

It then calculates the difference and saves it in an aggregator. At the end of the script, it prints a history of the aggregator, displaying how much time tg3_poll() consumed. Note that the time will be spread out. You may care about the long poll() sessions.

Example 5-29 Example script to aggregate tg3_poll() time

```
global value
global netpstats
global cpustats
global init_time

probe begin{
   printf("Press ^C to stop\n")
   init_time = gettimeofday_ms()
}

probe module("tg3").function("tg3_poll"){
   value = gettimeofday_ns()
   cpustats <<< cpu()
}

probe module("tg3").function("tg3_poll").return{
   diff = gettimeofday_ns() - value
   netpstats <<< diff
}

probe end {
   print(@hist_linear(netpstats,0, 15000, 500))
   print(@hist_log(cpustats))
   printf("Total time: %d miliseconds\n", gettimeofday_ms() - init_time)
}
```

Example 5-30 shows the output of Example 5-29 on page 55.

Example 5-30 Sample output

```
[root@mu leitao]# stap tg3.stap
Press ^C to stop
 value |-------------------------------------------------- count
  1500 |                                                    0
  2000 |                                                    0
  2500 |@@                                                  2
  3000 |@@@@@@@@@@@@@@@@                                   15
  3500 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        37
  4000 |@@@@@@@@                                            8
  4500 |@@@@                                                4
  5000 |@@@@@@@@@@                                        10
  5500 |@@@@@                                               5
  6000 |@                                                   1
  6500 |                                                    0
```

```
  7000 |@@                                                  2
  7500 |@@@@@                                               5
  8000 |                                                    0
  8500 |                                                    0
  9000 |@@@@                                                4
  9500 |                                                    0
 10000 |@                                                   1
 10500 |                                                    0
 11000 |@@@@                                                4
 11500 |@@@                                                 3
 12000 |@                                                   1
 12500 |@@                                                  2
 13000 |@@                                                  2
 13500 |                                                    0
 14000 |@@                                                  2
 14500 |@                                                   1
 15000 |@                                                   1
>15000 |@                                                   1

value |-------------------------------------------------- count
    0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 111
    1 |                                                    0
    2 |                                                    0

Total time: 19018 miliseconds
```

6

Advanced topics and recommendations

In this chapter we cover more advanced topics such as queues, guru mode, embedded C, strategies and recommendations for a good SystemTap script. We also include some general usage tips at the end of the chapter.

6.1 SystemTap queues

SystemTap provides a framework that allows it to work with queues in an especially easy manner.

A queue is a special SystemTap structure that provides a way to display statistical information for operations that happen in two phases. For instance, a prepare phase and an execution phase, or a waiting phase and an execute phase.

Queues are very useful when working with I/O operations that run in two phases; a request is made and queued, then it is scheduled to run and runs to completion. Another area that has the same behavior is the transmission of network packets. SystemTap can provide many benefits when using SystemTap queues to analyze network related performance.

Queue functions are a set of functions that are defined in a tapset called queue_stats.stp and that might be used when dealing with simple states such as “wait” and “running”. This feature is also used to get statistics from sys_calls, I/O operations such as disk access, and network, among others.

When using a queue, you basically specify when an event is waiting, running, or done. Based on this, the framework will provide a percentage of time that the event was waiting or executing.

Here are the basic steps for utilizing the queue framework:

1. Create a queue.
2. Specify where the event will be waiting.
3. Specify where the event will be running.
4. Specify where the event will be finished.
5. Display the output statistics.

Example 6-1 shows a simple example that handles syscalls. The syscall will be instrumented so that when the code is not executing a system call, we will define this state as “waiting”. When a system call is executing, then it will be in the running state, and when a system call is returned, the call is complete, and the waiting state will begin, in another cycle.

Using this mechanism, we can see how much of the CPU is being utilized inside a system call.

Example 6-1 shows the code.

Example 6-1 Sample script using queues to measure system call timings

```
probe begin{
        qsq_start("syscall")
        qs_wait("syscall")
}
probe syscall.* {
        qs_run("syscall")
}

probe syscall.*.return {
        qs_done("syscall")
        qs_wait("syscall")
}
```

```
probe end{
        qsq_print("syscall")
}
```

Example 6-2 shows the output.

Example 6-2 Sample output

```
syscall: 19757.002 ops/s, 0.510 qlen, 138 await, 112 svctm, 46% wait, 99% util
```

The next example (Example 6-3) will track a process and display how much time, in percent, was waiting for I/O, which means that the process task was promoted to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE. In this specific example, it looks for the process named db2sysc, from time to time, and in the end, prints the time that this task wasted waiting for I/O.

Example 6-3 A simple queue example to display I/O

```
probe begin{
        printf(“Press ^c when done\n”)
        qsq_start("state")
}

probe timer.profile {
        if (isinstr(task_execname(task_current()), "db2sysc")){
                state = task_state(task_current())
                if (state = 0){
                        qs_run("state")
                        qs_done("state")
                } else if (state = 1){
                        qs_wait("state")
                } else if (state = 2) {
                        qs_wait("state")
                }
        }
}

probe end{
        qsq_print("state")
        exit()
}
```

Example 6-4 shows the output.

Example 6-4

```
state: 0.000 ops/s, 413.454 qlen, 0 await, 0 svctm, 60% wait, 0% util
```

6.2 Guru mode

Important: Guru mode provides for some additional capabilities, but also removes various system protection facilities. Therefore, using guru mode in a production environment should only be done by experts who thoroughly understand the consequences.

SystemTap supports some advanced features that are enabled in the guru mode. In this mode, which is enabled when you run SystemTap using the -g argument, you do not have the memory and data protection normally provided by the SystemTap access restrictions.

Without these data access restrictions, you can modify any data inside a function, or even memory spaces. This is very useful for some interesting cases, such as testing a patch (fix) prototype, or even injecting errors somewhere in the code.

So, in guru mode you can change the behavior of the kernel, altering how it executes a function.

Example 6-5 demonstrates how powerful SystemTap is. In this example, the write syscall is probed and the output is changed, so that the first six characters are not displayed. To do so, the buf pointer is shifted 6 characters.

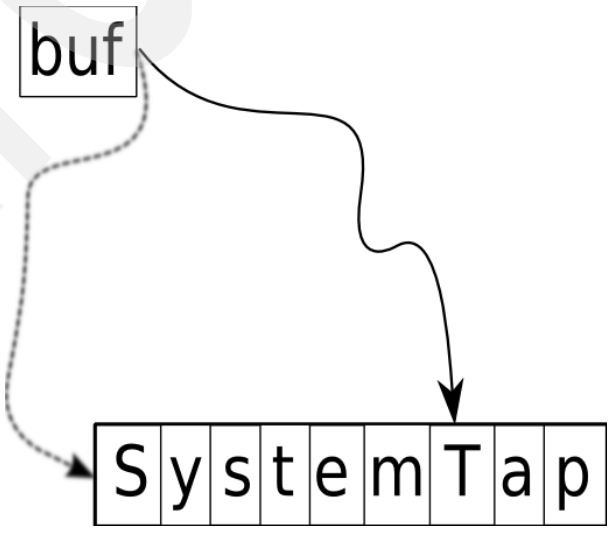
Example 6-5 A script to trap sys_write

```
probe syscall.write {
        if (isinstr(user_string($buf),"SystemTap")){
                $buf = $buf + 0x06 // Shift 6 characters
        }

}
```

Example 6-6 displays a graphic showing how the script changes the buf pointer in the write syscall.

Example 6-6 6-character shifting of buf pointer



Example 6-7 shows what you will get when you print SystemTap.

Example 6-7 Sample output

```
# echo SystemTap
Tap
#
```

6.2.1 Writing embedded C code

Once you are in guru mode, you may want to use more advanced features. For example, writing your own C code. This is a feature that SystemTap provides to users who want to create more complex scripts.

When you write embedded C code, SystemTap includes this block of code and puts it in the C file generated during the elaboration phase. It does this without even parsing and checking for any kind of errors that might be included.

Embedded C code is usually used inside a function that starts with %{ and ends with %}.

Note that if you have errors inside your embedded C code, this will only appear during the compilation phase. It is often difficult to understand what is wrong and to fix it, because errors happen in the compilation phase. Line numbers are different from your script, and you will only see the error when compiling using verbose options. See 6.2.2, “Finding embedded C errors” on page 63.

Example 6-8 shows an embedded C code example.

Example 6-8 An embedded C example

```
%{
        #include<linux/netdevice.h>
%}

function change_mtu:long (dev:long, val:long) %{
        int ret;
        struct net_device *netdevice;

        netdevice = (struct net_device *)(long)THIS->dev;
        printk("%s has MTU changed from %d to %d\n",
            (char *) netdevice->name, (int) netdevice->mtu, (int)THIS->val);
        dev_set_mtu(netdevice, THIS->val);
%}

probe module("tg3").function("tg3_get_stats") {
        change_mtu($dev, 2000)
        printf("Changing tg3 MTU\n")
}
```

Calling functions

From inside your embedded C code you can call external functions. External functions are any routine that exists in the kernel source that is not defined as **`inline`**. You need to be careful when calling an external routine, since it may change the state of your machine. For example, it could change a variable value, or even lock or release a lock, which will put the system in an inconsistent state.

It is important to include the header file that defines the function you are using, otherwise you will get the error shown in Example 6-9.

Example 6-9 Example of an error

```
implicit declaration of function ‘function_name'
```

In Example 6-8, we call the external function “dev_set_mtu”.

The THIS structure

When you create an embedded C function, you will usually want to call it, passing arguments to it, and getting the return value from it.

The way SystemTap implements this communication is through the THIS structure. This structure is dynamic and contains basically the arguments you specified as variables, and a field called __retvalue, which is the value that will be returned to the call site.

Example 6-10 shows such a function.

Example 6-10 Sample foo function

```
   function foo:long (bar:string, baz:long) %{
             // Code here
}%
```

When SystemTap compiles this function, the THIS structure will contain the following fields:

- long __retvalue
- char *bar
- long baz

So, if you want to access any argument, for example bar, you will need to reference THIS->bar. And the return value should be saved in THIS->__retvalue.

Example 6-11shows how this is used.

Example 6-11 Using a THIS structure

```
function foo:long (bar:long) %{
        printk(KERN_EMERG "Bar: %d\n", (int) THIS->bar);
        THIS->__retvalue = 0;
%}

probe syscall.write {
        printf("%d\n", foo($fd))
        exit()
}
```

Example 6-12 shows the result of Example 6-11.

Example 6-12 Output of the above example

```
[root@mu leitao]# stap this.stp  -g
0
Message from syslogd@ at Wed Nov 12 07:26:10 2008 ...
mu kernel: Bar: 1
```

Kread

Whenever you want to dereference a pointer, it is recommended to use the kread() macro. This macro is used to protect against pointers that might crash your system due to misreferencing. So, you would use `x = kread(foo->bar)` instead of `x = foo->bar`.

6.2.2 Finding embedded C errors

If your embedded C code contains errors, you will receive the message shown in Example 6-13 during compilation.

Example 6-13 Example of embedded C code compilation error

```
# stap embedded.stap  -g
Pass 4: compilation failed.  Try again with more '-v' (verbose) options.
```

The best way to determine what is wrong is to compile using “-gkuvv”, which generates verbose output and does not delete the source file, so that you could see what is wrong in the source code. After you attempt to compile the code, the line number that is causing the error will appear in the error message as shown in Example 6-14.

Example 6-14 Compile time error

```
# stap embedded.stap  -gkvv 2>&1 | grep  "error:"
/tmp/stapOq2GRb/stap_5601f0546ef5f80afcc1ba32a1a6f3a0_874.c:147: error:
dereferencing pointer to incomplete type
```

You can then find the generated source code and look for the failing line number in the temporary directory.

6.2.3 Common error when using Embedded C

The following are the most common errors when using Embedded C.

Message implicit declaration of function 'xxxx'

When you get this error, it can have two possible causes:

- You forgot to include the header in your script. Just insert a section that includes it, as shown in Example 6-15.

Example 6-15 Including a header in your script

```
%{
   include<linux/netdevice.h>
%}
```

- You are trying to call an inline function, which is not allowed by definition.

error: expected ';' before '}' token

If you get this message during the script compilation, you probably forgot to put a semicolon (;) on the end of a specific line.

Attention: On Embedded C, each line must contain a semicolon, otherwise the compiler will complain.

6.3 Dealing with skipped probes

When running a SystemTap script, you may occasionally encounter a warning that there were “skipped probes.”

```
Warning: Number of errors: 0, skipped probes: 10
```

This can happen for various reasons. For instance, the system may be very busy, and have trouble keeping up with the number of probes that are hit. Or, the variables used in the script may be required by too many probe handlers, and lock contention for access to those variables may be causing too much of a delay. Another common reason is that the probe handler could be too complex and require too much processing time. Having a long-running probe handler is not such a big deal if the probe is hit only every few seconds, but if it is a frequently hit probe then this can be a big problem.

The resolution to this problem in most cases is to simply reconsider the amount of work happening in the probe handler. One strategy for doing this is to filter the probes on which you execute the full handler to only the bare minimum. For instance, if you only care to gather information from a probe when a certain parameter is passed to the probe handler, create an if statement that checks for this condition and only executes the code in the handler if the condition is true.

Another method is to limit the processing and output in the probe handler. It is likely not necessary nor desirable to recalculate and update statistics, or print very much information every time a probe handler is executed. Instead of updating multiple global variables, update as few as possible. Leave any accumulation and analysis of results, and printing of long output, until a later time. This kind of processing and output can take place when the script exits, or in a timer probe that only runs every few seconds.

For more details on this subject, see

http://sourceware.org/systemtap/wiki/TipSkippedProbesOptimization

6.4 Modifying SystemTap global limits

SystemTap has a set of global limits that help protect the system from scripts that might go too far and adversely affect the system. For example, there are limits in place to prevent a script from recursing too deep, creating enormous strings, or creating arrays that are too large. The following configuration options can be used:

MAXNESTING - The maximum number of levels of recursion allowed. The current default is 10 levels.

MAXSTRINGLEN - The maximum length of any single string used in the script. The current default is 128 characters.

MAXACTION - The maximum number of actions that can be executed in a probe handler. The current default is 1000

MAXMAPENTRIES - The maximum array size used during a probe. The current default is 2048 entries.

MAXTRYLOCK - The number of times to attempt to acquire a lock on a global variable before giving up and skipping the probe. The current default is 1000.

MAXERRORS - The number of allowable errors during the execution of the script before forcing an exit. The current default is 0, meaning that even a single error will force an exit.

MAXSKIPPED - The number of probes that may be skipped before forcing an exit. The current default is 100.

MINSTACKSPACE - The minimum amount of free stack space in the kernel. The current default is 1024 bytes.

Note that you can change the values of these default limits if you wish, but remember that they are there for a reason. Often the script can be written in a different way so that it is not necessary to change a limit to make it work.

To change one of these defaults, use the -D argument when running SystemTap:

```
# stap -D MAXACTION=1500 longprobe.stp
```

6.5 Fault injection

Since you can probe functions everywhere, SystemTap can be used as a way to inject errors/faults someplace into the kernel in order to see how the kernel behaves. This process can be used on test suites and can be very useful.

Since you could stop a function anywhere, and even interact with the system state, such as changing a variable, it is easy to reproduce errors not easily reproducible, or even test some special cases that you could not do easily, even if you were to compile the kernel.

Another useful case for fault injection is changing the value of a return function. You can return a different value from almost every function in the system, even in system calls.

Example 6-16 shows an example that shows the mkdir system call being probed during its return, and if the file name being created is something like “notcreate”, then it changes the return value to 0, which means the function was not successful, failing to create the directory.

Example 6-16 Preventing the creation of a directory using the name “notcreate”

```
probe syscall.mkdir.return {
        if (isinstr(user_string($pathname), "notcreate"))
                $return = 1
}
```

Example 6-17 shows the result.

Example 6-17 Trying to create a notcreate directory while the example above is running

```
# mkdir /tmp/foobar
# mkdir /tmp/notcreate
mkdir: cannot create directory `/tmp/notcreate'
```

Another useful script is used to simulate a system in some specific state, for example latency in the network. This is very useful when working in device driver development, user space development, and performance analysis.

This application could easily be done by hooking the transmission routine and creating a delay in the hook, which will delay the transmission of the packet. If you want to delay the reception, just add the same hook in the reception routine (tg3_rx in the tg3 device).

Example 6-18 shows an example where the tg3_tx() function is probed, and before it is executed, we take a long time executing a simple loop, whose only purpose is to delay the packet transmission.

Example 6-18 Delaying tg3 transmission

```
function wait_time() %{
        int i = 2;
        while (i < 10000){
                i = i*3/2;
        }
%}

probe module("tg3").function("tg3_tx"){
        wait_time()
}
```

Another useful case, shown in Example 6-19, is to simulate low memory, which causes the kernel to stop allocating more memory. It could be accomplished in a variety of ways. The easiest way to do so may be changing the kmalloc return value, which could be done as shown in Example 6-19.

Example 6-19 Trapping kmalloc to not “reject “some allocations

```
global i = 0

probe kernel.function("__kmalloc").return {
        $return = 0
        if (i < 10){
                printf("Hooked\n")
        } else {
                exit()
        }
        i = i + 1
}
```

Note that his example does not deny the memory allocation. It does allocate the memory, but the return value is changed to 0, which simulates a failed memory allocation. Hence, the use of this script causes memory leakage.

6.6 Tips

This final section provides some useful tips that may help make your use of SystemTap a success.

6.6.1 Inline functions

SystemTap allows you to probe inline functions, but on the enterprise version, you do not have access to its arguments or return point. So, before getting an argument or return point from a function, check to see whether it is not inline, otherwise you will get the following error shown in Example 6-20 on page 67.

Example 6-20 Error while trying to get arguments from an inline function

```
semantic error: failed to retrieve location attribute for local 'size' (dieoffset:
0x2108bdb): identifier '$size' at kmalloc.stp:5:17
```

6.6.2 Functions that could be probed

In order to discover which functions could be instrumented, you could run the command shown in Example 6-21.

Example 6-21 Listing functions that are not blacklisted.

```
#stap -p2 -e 'probe kernel.function("*") {}' 2>&1 | grep ^kernel.fun
```

Example 6-22 shows part of the output.

Example 6-22 Part of the output

```
kernel.function("run_init_process@init/main.c:746") /* pc=0x0 */ /* <-
kernel.function("*") */
kernel.function("init_post@init/main.c:755") /* pc=0x3c */ /* <-
kernel.function("*") */
kernel.function("rest_init@init/main.c:422") /* pc=0xf7 */ /* <-
kernel.function("*") */
kernel.function("execve@include/asm/unistd.h:477") /* pc=0x1 */ /* <-
kernel.function("*") */
kernel.function("try_name@init/do_mounts.c:55") /* pc=0x118 */ /* <-
kernel.function("*") */
kernel.function("name_to_dev_t@init/do_mounts.c:137") /* pc=0x295 */ /* <-
kernel.function("*") */
kernel.function("new_decode_dev@include/linux/kdev_t.h:49") /* pc=0x1f0 */ /* <-
kernel.function("*") */
kernel.function("strncmp@include/asm/string.h:125") /* pc=0x2d6 */ /* <-
kernel.function("*") */
kernel.function("new_decode_dev@include/linux/kdev_t.h:49") /* pc=0x36b */ /* <-
kernel.function("*") */
kernel.function("strcmp@include/asm/string.h:103") /* pc=0x38e */ /* <-
kernel.function("*") */
......
```

6.6.3 Access user space pointers

When working with system calls, they normally point to user space addresses, for example a pointer that says the name of a file when trying to open or create it.

The pointer to the string address containing this file points to a user space address, so in order to access it, you will need a special function called **user_string()**, as in Example 6-23.

Example 6-23 Accessing user space string pointer

```
probe kernel.function("sys_mkdir"){
        printf("Creating directory %s using mode %d...", user_string($pathname),
$mode)
}
```

6.6.4 Printing time

It is easy to print time during your traces. Most libc functions are available in SystemTap, for example **ctime()**, which will format the number of milliseconds since the UNIX® epoch; see Example 6-24.

Example 6-24 Printing time during a probe

```
probe begin {
        printf("%s\n", ctime(gettimeofday_ms()))
}
```

Example 6-25 shows the output.

Example 6-25 Output of the above script

```
Tue Sep 10 10:18:47 2013
```

6.6.5 Executing system applications

SystemTap allows you to call any external application from inside your SystemTap script. This is a powerful feature that enables you to set up the environment before initializing the test. To do so, use the function **system()**; see Example 6-26.

Example 6-26 Executing command in user space

```
probe begin{
        system("ping 9.3.4.161 -c 1")
}
```

6.6.6 Unprivileged script

If you get the message "`embedded code in unprivileged script`" when running a stap script, it means that the script is trying to execute some code that has tried to access protected data.

If this is something you really want to do, then you need to call SystemTap in guru mode, by passing the -g parameter. Note that this could corrupt your kernel space data.

6.6.7 Vim and SystemTap

There is a syntax highlighter for VIM, and it can be very useful if you are writing many scripts and like syntax highlighters.

The installation method is well described in the SystemTap's wiki page at:

`http://sourceware.org/SystemTap/wiki/VIMSyntaxHighlightingForSystemTap`

6.6.8 Dump all variables

If you are using a newer SystemTap version (version 0.8 and newer) , then you have the variables $$locals and $$vars, which print all function local variables and all arguments, and this is very useful in cases where you do not know the parameters' names. To use it, create something like the following:

```
printf("local variables %s and arguments%s\n", $$locals, $$vars)
```

6.6.9 Cached SystemTap programs

The process of converting a SystemTap script to something that can be executed in the kernel normally only takes a few seconds. In the case of long complex scripts, however, the time added to the beginning of the execution can be noticeable. Because of this, SystemTap generates a hash value for each script after the elaboration phase, and copies the resulting .c and .ko files to the user's home directory under the .systemtap/cache directory. If nothing in the script or the tapsets has changed, **staprun** executes the module directly from the cached copy. Otherwise, it is converted to C all over again, and recompiled.

If you can determine the cache entry associated with a script you wish to run, you can easily just call **staprun** on the .ko file in that directory, and execute the probe without having to go through the normal process. This .ko file may be moved to another machine and executed using **staprun**, provided the other machine has the same architecture, kernel, and so on. An easy way to determine which cached .ko file corresponds to a script is to use the -p4 option on stap; see Example 6-27.

Example 6-27 Showing the cached .ko file

```
# stap -p4 iosch.stp
/root/.systemtap/cache/62/stap_62c2d8b0364480708110fe0cadbd43c4_3058.ko
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

Online resources

These Web sites are also relevant as further information sources:

- SystemTap Graphical User Interface

  `http://stapgui.sourceforge.net`
- SystemTap Language Reference

  `http://sourceware.org/systemtap/langref/`
- SystemTap Source Code

  `http://sources.redhat.com/systemtap`
- SystemTap Tar Ball

  `ftp://sources.redhat.com/pub/systemtap/snapshots/`
- Skipped probres reference

  `http://sourceware.org/systemtap/wiki/TipSkippedProbesOptimization.`
- VIM Syntx Highlighter

  `http://sourceware.org/SystemTap/wiki/VIMSyntaxHighlightingForSystemTap`
- elfutils information

  `http://fedorahosted.org/releases/e/l/elfutils.`

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems

Learn about SystemTap basics

Utilize SystemTap for system analysis

Learn concepts through simple examples

This IBM Redbooks paper provides information related to the Linux-based SystemTap tool. This tool can be used by developers, analysts and specialists to better understand and analyze Linux systems and applications. Until now, its primary use has been to look into the workings of the Linux kernel and various system calls. Its function is expanding to provide information related to user space as well, but that will not be the focus of this paper.

This paper describes the basic mechanisms used by SystemTap to gather data and offers guidance on getting started using the tool. Using examples, we show how to use the tool to capture and present useful information that may not be readily available by the myriad of other tools generally available on the Linux platform. Though our examples are relatively simple, they provide the basics to build on to develop more robust scripts that meet the reader's specific needs.

The reader will appreciate the power and simple elegance of SystemTap and how it can be used to help analyze Linux systems to identify functional and performance-related problems, which in turn can help ensure that system and application design will minimize the chances for problems in the future.

This paper is intended for individuals who have programming experience and familiarity with Linux and existing system facilities.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

REDP-4469-00