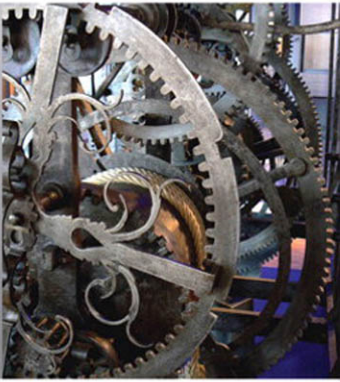
DTrace

Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD



Brendan Gregg • Jim Mauro

Foreword by Bryan Cantrill

DTrace

DTrace

Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD

Brendan Gregg
Jim Mauro

PRENTICE
HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Gregg, Brendan.
Dynamic tracing in Oracle Solaris, Mac OS X, and FreeBSD / Brendan
Gregg, Jim Mauro.
p. cm.
Includes index.
ISBN-13: 978-0-13-209151-0 (alk. paper)
ISBN-10: 0-13-209151-8 (alk. paper)
1. Debugging in computer science. 2. Solaris (Computer file) 3. Mac
OS. 4. FreeBSD. I. Mauro, Jim. II. Title.
QA76.9.D43G74 2011
005.1'4—dc22
2010047609

Copyright © 2011 Oracle and/or its affiliates. All rights reserved.
500 Oracle Parkway, Redwood Shores, CA 94065

Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-13-209151-0
ISBN-10: 0-13-209151-8
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, March 2011

Contents

Part I Introduction

Part II Using DTrace

Part III Additional User Topics

Chapter 13 Tools 947

Foreword

In early 2004, DTrace remained nascent; while Mike Shapiro, Adam Leventhal, and I had completed our initial implementation in late 2003, it still had substantial gaps (for example, we had not yet completed user-level instrumentation on x86), many missing providers, and many features yet to be discovered. In part because we were still finishing it, we had only just started to publicly describe what we had done—and DTrace remained almost entirely unknown outside of Sun. Around this time, I stumbled on an obscure little Solaris-based tool called `psio` that used the operating system's awkward pre-DTrace instrumentation facility, TNF, to determine the top I/O-inducing processes. It must be noted that TNF—which arcanely stands for Trace Normal Form—is a baroque, brittle, pedantic framework notable only for painfully yielding a modicum of system observability where there was previously none; writing a tool to interpret TNF in this way is a task of Herculean proportions. Seeing this TNF-based tool, I knew that its author—an Australian named Brendan Gregg—must be a kindred spirit: gritty, persistent, and hell-bent on shining a light into the inky black of the system's depths. Given that his TNF contortionist act would be reduced to nearly a one-liner in DTrace, it was a Promethean pleasure to introduce Brendan to DTrace:

```
From: Bryan Cantrill <bmc@eng.sun.com>
To: Brendan Gregg <brendan.gregg@tpg.com.au>
Subject: psio and DTrace
Date: Fri, 9 Jan 2004 13:35:41 -0800 (PST)
```

```
Brendan,

A colleague brought your "psio" to my attention -- very interesting.
Have you heard about DTrace, a new facility for dynamic instrumentation
in Solaris 10? As you will quickly see, there's a _lot_ you can do with
DTrace -- much more than one could ever do with TNF.
...
```

With Brendan's cordial reply, it was clear that although he was very interested in exploring DTrace, he (naturally) hadn't had much of an opportunity to really use it. And perhaps, dear reader, this describes you, too: someone who has seen DTrace demonstrated or perhaps used it a bit and, while understanding its potential value, has perhaps never actually used it to solve a real problem. It should come as no surprise that one's disposition changes when DTrace is used not to make some academic point about the system but rather to save one's own bacon. After this watershed moment—which we came to (rather inarticulately) call the DTrace-just-saved-my-butt moment—DTrace is viewed not as merely interesting but as essential, and one starts to reach for it ever earlier in the diagnostic process.

Given his aptitude and desire for understanding the system, it should come as no surprise that when I heard back from Brendan again some two months later, he was long past his moment, having already developed a DTrace dependency:

```
From: Brendan Gregg <brendan.gregg@tpg.com.au>
To: Bryan Cantrill <bmc@eng.sun.com>
Subject: Re: psio and DTrace
Date: Mon, 29 Mar 2004 00:43:27 +1000 (EST)

G'Day Bryan,

DTrace is a superb tool. I'm already somewhat dependent on using it.
So far I've rewritten my "psio" tool to use DTrace (now it is more
robust and can access more details) and an iosnoop.d tool.
...
```

Brendan went on to an exhaustive list of what he liked and didn't like in DTrace. As one of our first major users outside of Sun, this feedback was tremendously valuable to us and very much shaped the evolution of DTrace.

And Brendan became not only one of the earliest users and foremost experts on DTrace but also a key contributor: Brendan's collection of scripts—the DTraceToolkit—became an essential factor in DTrace's adoption (and may well be how you yourself came to learn about DTrace). Indeed, one of the DTraceToolkit scripts, `shellsnoop`, remains a personal favorite of mine: It uses the syscall provider to display the contents of every read and write executed by a shell. In the early days of DTrace, whenever anyone asked whether there were security implications to running DTrace, I used to love to demo this bad boy; there's nothing like seeing someone else's password come across in clear text to wake up an audience!

Given not only Brendan's essential role in DTrace but also his gift for clearly explaining complicated systems, it is entirely fitting that he is the author of the volume now in your hands. And given the degree to which proficient use of DTrace requires mastery not only of DTrace itself but of the larger system around it, it is further appropriate that Brendan teamed up with Jim Mauro of *Solaris Internals* (McDougall and Mauro, 2006) fame. Together, Brendan and Jim are bringing you not just a book about DTrace but a book about using it in the wild, on real problems and actual systems. That is, this book isn't about dazzling you with what DTrace can do; it is about getting you closer to that moment when it is *your* butt that DTrace saves. So, enjoy the book, and remember: DTrace is a workhorse, not a show horse. Don't just read this book; put it to work and *use* it!

—Bryan Cantrill
Piedmont, California

This page intentionally left blank

Preface

> "[expletive deleted] it's like they saw inside my head and gave me The One True Tool."
>
> —A Slashdotter, in a post referring to DTrace

> "With DTrace, I can walk into a room of hardened technologists and get them giggling."
>
> —Bryan Cantrill, father of DTrace

Welcome to Oracle Solaris Dynamic Tracing—DTrace! It's been more than five years since DTrace made its first appearance in Solaris 10 3/05, and it has been just amazing to see how it has completely changed the rules of understanding systems and the applications they run. The DTrace technical community continues to grow, embracing the technology, pushing DTrace in every possible direction, and sharing new and innovative methods for using DTrace to diagnose myriad system and application problems. Our personal experience with DTrace has been an adventure in learning, helping customers solve problems faster, and improving our internal engineering efforts to analyze systems and find ways to make our technology better and faster.

The opening quotes illustrate just some of the reactions we have seen when users experience how DTrace empowers them to observe, analyze, debug, and understand their systems and workloads. The community acceptance and adoption of DTrace has been enormously gratifying to watch and participate in. We have seen DTrace ported to other operating systems: Mac OS X and FreeBSD both ship with DTrace. We see tools emerging that leverage the power of DTrace, most of which are being developed by community members. And of course feedback and comments from users over the years have driven continued refinements and new features in DTrace.

About This Book

This book is all about DTrace, with the emphasis on *using* DTrace to understand, observe, and diagnose systems and applications. A deep understanding of the details of how DTrace works is not necessary to using DTrace to diagnose and solve problems; thus, the book covers using DTrace on systems and applications, with command-line examples and a great many D scripts. Depending on your level of experience, we intend the book's organization to facilitate its use as a reference guide, allowing you to refer to specific chapters when diagnosing a particular area of the system or application.

This is not a generic performance and tools book. That is, many tools are available for doing performance analysis, observing the system and applications, debugging, and tuning. These tools exist in various places—bundled with the operating system, part of the application development environment, downloadable tools, and so on. It is probable that other tools and utilities will be part of your efforts involving DTrace (for example, using system stat tools to get a big-picture view of system resource utilization). Throughout this book, you'll see examples of some of these tools being used as they apply to the subject at hand and aid in highlighting a specific point, and coverage of the utility will include only what is necessary for clarity.

Our approach in writing this book was that DTrace is best learned by example. This approach has several benefits. The volume of DTrace scripts and one-liners included in the text gives readers a chance to begin making effective and practical use of DTrace immediately. The examples and scripts in the book were inspired by the DTraceToolkit scripts, originally created by Brendan Gregg to meet his own needs and experiences analyzing system problems. The scripts in this book encapsulate those experiences but also introduce analysis of different topics in a focused and easy-to-follow manner, to aid learning. They generate answers to real and useful questions and serve as a starting point for building more complex scripts. Rather than an arbitrary collection of programs intended to highlight a potentially interesting feature of DTrace or the underlying system, the scripts and one-liners are all based on practical requirements, providing insight about the system under observation. Explanations are provided throughout that discuss the DTrace used, as well as the output generated.

DTrace was first introduced in Oracle Solaris 10 3/05 (the first release of Solaris 10) in March 2005. It is available in all Solaris 10 releases, as well as OpenSolaris, and has been ported to Mac OS X 10.5 (Leopard) and FreeBSD 7.1. Although much of DTrace is operating system–agnostic, there are differences, such as newer DTrace features that are not yet available everywhere.[1] Using DTrace to trace operating system–specific functions, especially unstable interfaces within the kernel, will of course be very different across the different operating systems (although the same methodologies will be applicable to all). These differences are discussed throughout the book as appropriate. The focus of the book is Oracle Solaris, with key DTrace scripts provided for Mac OS X and FreeBSD. Readers on those operating systems are encouraged to examine the Solaris-specific examples, which demonstrate principles of using DTrace and often only require minor changes to execute elsewhere. Scripts that have been ported to these other operating systems will be available on the *DTrace* book Web site, *www.dtracebook.com*.

How This Book Is Structured

This book is organized in three parts, each combining a logical group of chapters related to a specific area of DTrace or subject matter.

Part I, Introduction, is introductory text, providing an overview of DTrace and its features in Chapter 1, Introduction to DTrace, and a quick tour of the D Language in Chapter 2, D Language. The information contained in these chapters is intended to support the material in the remaining chapters but does not necessarily replace the more detailed language reference available in the online, wiki-based DTrace documentation (see “Supplemental Material and References”).

Part II, Using DTrace, gets you started using DTrace hands-on. Chapter 3, System View, provides an introduction to the general topic of system performance, observability, and debugging—the art of system forensics. Old hands and those who have read McDougall, Mauro, and Gregg (2006) may choose to pass over this chapter, but a holistic view of system and software behavior is as necessary to effective use of DTrace as knowledge of the language syntax. The next several chapters deal with functional areas of the operating system in detail: the I/O path—Chapter 4, Disk I/O, and Chapter 5, File Systems—is followed by Chapter 6, Network Lower-Level Protocols, and Chapter 7, Application-Level Protocols, on the network protocols. A change of direction occurs at Chapter 8, Languages, where application-level concerns become the focus. Chapter 8 itself covers programming

---

[1] This will improve after publication of this book, because other operating systems include the newer features.

languages and DTrace's role in the development process. Chapter 9, Applications, deals with the analysis of applications. Databases are dealt with specifically in Chapter 10, Databases.

Part III, Additional User Topics, continues the "using DTrace" theme, covering using DTrace in a security context (Chapter 11, Security), analyzing the kernel (Chapter 12, Kernel), tools built on top of DTrace (Chapter 13, Tools), and some tips and tricks for all users (Chapter 14, Tips and Tricks).

Each chapter follows a broadly similar format of discussion, strategy suggestions, checklists, and example programs. Functional diagrams are also included in the book to guide the reader to use DTrace effectively and quickly.

For further sources of information, see the online "Supplemental Material and References" section, as well as the annotated bibliography of textbook and online material provided at the end of the book.

Intended Audience

DTrace was designed for use by technical staff across a variety of different roles, skills, experience, and knowledge levels. That said, it is a software analysis and debugging tool, and any substantial use requires writing scripts in D. D is a structured language very similar to C, and users of that language can quickly take advantage of that familiarity. It is assumed that the reader will have some knowledge of operating system and software concepts and some programming background in scripting languages (Perl, shell, and so on) and/or languages (C, C++, and so on).

In addition, you should be familiar with the architecture of the platform you're using DTrace on. Textbooks on Solaris, FreeBSD, and Mac OS X are detailed in the bibliography.

To minimize the level of programming skill required, we have provided many DTrace scripts that you can use immediately without needing to write code. These also help you learn how to write your own DTrace scripts, by providing example solutions that are also starting points for customization. The DTraceToolkit[2] is a popular collection of such DTrace scripts that has been serving this role to date, created and mostly written by the primary author of this book. Building upon that success, we have created a book that is (we hope) the most comprehensive source for DTrace script examples.[3]

---

[2] This is linked on *www.brendangregg.com/dtrace.html* and *www.dtracebook.com*.

[3] The DTraceToolkit now needs updating to catch up!

This book will serve as a valuable reference for anyone who has an interest in or need to use DTrace, whether it is a necessary part of your day job, a student studying operating systems, or a casual user interested in figuring out why the hard drive on your personal computer is clattering away doing disk I/Os.

Specific audiences for this book include the following.

- **Systems administrators, database administrators, performance analysts, and support staff** responsible for the care and feeding of their production systems can use this book as a guide to diagnose performance and pathological behavior problems, understand capacity and resource usage, and work with developers and software providers to troubleshoot application issues and optimize system performance.
- **Application developers** can use DTrace for debugging applications and utilizing DTrace's User Statically Defined Tracing (USDT) for inserting DTrace probes into their code.
- **Kernel developers** can use DTrace for debugging kernel modules.
- **Students** studying operating systems and application software can use DTrace because the observability that it provides makes it a perfect tool to supplement the learning process. Also, there's the implementation of DTrace itself. DTrace is among the most well-thought-out and well-designed software systems ever created, incorporating brilliantly crafted solutions to the extremely complex problems inherent in building a dynamic instrumentation framework. Studying the DTrace design and source code serves as a world-class example of software engineering and computer science.

Note that there is a minimum knowledge level assumed on the part of the reader for the topics covered, allowing this book to focus on the application of DTrace for those topics.

Supplemental Material and References

Readers are encouraged to visit the Web site for this book: *www.dtracebook.com*.

All the scripts contained in the book, as well as reader feedback and comments, book errata, and subsequent material that didn't make the publication deadline, can be downloaded from the site.

Brendan Gregg's DTraceToolkit is free to download and contains more than 200 scripts covering every everything from disks and networks to languages and the kernel. Some of these are used in this text: *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/dtracetoolkit*.

The DTrace online documentation should be referenced as needed: *http://wikis.sun.com/display/DTrace/Documentation*.

The OpenSolaris DTrace Community site contains links and information, including projects and additional sources for scripts: *http://hub.opensolaris.org/bin/view/Community+Group+dtrace/*.

The following texts (found in the bibliography) can be referenced to supplement DTrace analysis and used as learning tools:

- McDougall and Mauro, 2006
- McDougall, Mauro, and Gregg, 2006
- Gove, 2007
- Singh, 2006
- Neville-Neil and McKusick, 2004

Acknowledgments

The authors owe a huge debt of gratitude to Deirdré Straughan for her dedication and support. Deirdré spent countless hours reviewing and editing material, substantially improving the quality of the book. Deirdré has also dedicated much of her time and energy to marketing and raising awareness of DTrace and this book both inside and outside of Oracle.

Dominic Kay was tireless in his dedication to careful review of every chapter in this book, providing detailed commentary and feedback that improved the final text tremendously. Darryl Gove also provided extraordinary feedback, understanding the material very well and providing numerous ideas for improving how topics are explained. And Peter Memishian provided incredible feedback and expertise in the short time available to pick through the longest chapter in the book, Chapter 6, and greatly improve its accuracy.

Kim Wimpsett, our copy editor, worked through the manuscript with incredible detail and in great time. With so many code examples, technical terms, and output samples, this is a very difficult and tricky text to edit. Thanks so much for the hard work and patience.

We are very grateful to everyone who provided feedback and content on some or all of the chapters in the short time frame available for such a large book, notably, Alan Hargreaves, Alan Maguire, Andrew Krasny, Andy Bowers, Ann Rice, Boyd Adamson, Darren Moffatt, Glenn Brunette, Greg Price, Jarod Jenson, Jim Fiori, Joel Buckley, Marty Itzkowitz, Nasser Nouri, Rich Burridge, Robert Watson, Rui Paulo, and Vijay Tatkar.

A special thanks to Alan Hargreaves for his insights and comments and contributing his USDT example and case study in Appendix E.

Thanks to Chad Mynhier and Tariq Magdon-Ismail for their contributions.

Thanks to Richard McDougall for so many years of friendship and inspiration and for the use of the RMCplex.

We'd like to thank the software engineers who made this all possible in the first place, starting with team DTrace at Sun Microsystems (Bryan Cantrill, Mike Shapiro, and Adam Leventhal) for inventing DTrace and developing the code, and team DTrace at Apple for their port of not only DTrace but many DTraceToolkit scripts (Steve Peters, James McIlree, Terry Lambert, Tom Duffy, and Sean Callanan); and we are grateful for the work that John Birrell performed to port DTrace to FreeBSD. We'd also like to thank the software engineers, too numerous to mention here, who created all the DTrace providers we have demonstrated throughout the book.

Thanks to the worldwide community that has embraced DTrace and generated a whirlwind of activity on the public forums, such as dtrace-discuss. These have been the source of many great ideas, examples, use cases, questions, and answers over the years that educate the community and drive improvements in DTrace.

And a special thanks to Greg Doench, senior editor at Pearson, for his help, patience, and enthusiasm for this project and for working tirelessly once all the material was (finally) delivered.

Personal Acknowledgments from Brendan

Working on this book has been an enormous privilege, providing me the opportunity to take an amazing technology and to demonstrate its use in a variety of new areas. This was something I enjoyed doing with the DTraceToolkit, and here was an opportunity to go much further, demonstrating key uses of DTrace in more than 50 different topics. This was also an ambitious goal: Of the 230+ scripts in this book, only 45 are from the DTraceToolkit; most of the rest had to be newly created and are released here for the first time. Creating these new scripts required extensive research, configuration of application environments and client workloads, experimentation, and testing. It has been exhausting at times, but it is satisfying to know that this should be a valuable resource for many.

A special thanks to Jim for creating the DTrace book project, encouraging me to participate, and then working hard together to make sure it reached completion. Jim is an inspiration to excellence; he co-authored *Solaris Internals* (McDougall and Mauro, 2006) with Richard McDougall, which I studied from cover to cover while I was learning DTrace. I was profoundly impressed by its comprehensive coverage, detailed explanations, and technical depth. I was therefore honored to be invited to collaborate on this book and to work with someone who had the experience and desire to take on a similarly ambitious project. Jim has an amazing can-do attitude and willingness to take on hard problems, which proved essential as we worked through the numerous topics in this book. Jim, thanks; we somehow survived!

Thanks, of course, are also due to team DTrace; it's been a privilege to work with them and learn from them as part of the Fishworks team. Especially sitting next to Bryan for four years: Learning from him, I've greatly improved my software analysis skills and will never forget to separate problems of implementation from problems of abstraction.

Thanks to the various Sun/Oracle teams I regularly work with, share problems with, and learn from, including the Fishworks, Performance Availability Engineering (PAE), Independent Software Vendor (ISV) engineering, and ZFS teams.

Thanks to Claire, for the love, support, and patience during the many months this was to take, and then the many months beyond which it actually took to complete. These months included the birth of our child, Mitchell, making it especially tough for her when I was working late nights and weekends on the book.

—Brendan Gregg
Walnut Creek, California (formerly Sydney, Australia)
September 2010

Personal Acknowledgments from Jim

Working on this book was extremely gratifying and, to a large degree, educational. I entered the project completely confident in my knowledge of DTrace and its use for observing complex systems. A few months into this project, I quickly realized I had only scratched the surface. It's been enormously rewarding to be able to improve my knowledge and skills as I worked on this book, while at the same time improving and adding more value to the quality of this text.

First and foremost, a huge thank you to Brendan. Brendan's expertise and sheer energy never ceased to amaze me. He consistently produced huge amounts of material—DTrace scripts, one-liners, and examples—at a rate that I would have never thought humanly possible. He continually supplied an endless stream of ideas, constantly improving the quality of his work and mine. He is uncompromising in his standards for correctness and quality, and this work is a reflection of Brendan's commitment to excellence. Brendan's enthusiasm is contagious—throughout this project, Brendan's desire to educate and demonstrate the power of DTrace, and its use for solving problems and understanding software, was an inspiration. His expertise in developing complex scripts that illuminate the behavior of a complex area of the kernel or an application is uncanny. Thanks, mate; it's been a heck of a ride. More than anything, this is your book.

Thanks to my manager, Fraser Gardiner, for his patience and support.

I want to thank the members of Fraser's team who I have the opportunity to work with and learn from every day: Andy Bowers, Matt Finch, Calum Mackay, Tim Uglow, and Rick Weisner, all of whom rightfully belong in the "scary smart" category.

Speaking of "scary smart," a special thanks to my friend Jon Haslam for answering a constant stream of DTrace questions and for his amazing contributions to DTrace.

Thanks to Chad Mynhier for his ideas, contributions, patience, and understanding.

Thanks to my friends Richard McDougall and Bob Sneed for all the support, advice, and time spent keeping me going over the years. And a special thank-you to Richard for use of the RMCplex.

Thanks to Donna, Frank, and Dominic for their love, patience, and support.

Thanks Lisa, for the love, support, and inspiration and for just being you.

—Jim Mauro
Green Brook, New Jersey
September 2010

About the Authors

Brendan Gregg is a performance specialist at Joyent and is known worldwide in the field of DTrace. Brendan created and developed the DTraceToolkit and is the coauthor of *Solaris Performance and Tools* (McDougall, Mauro, and Gregg, 2006) as well as numerous articles about DTrace. He was previously the performance lead for the Sun/Oracle ZFS storage appliance and a software developer on the Fishworks advanced development team at Sun, where he worked with the three creators of DTrace. He has also worked as a system administrator, performance consultant, and instructor, and he has taught DTrace worldwide including workshops that he authored. His software achievements include creating the DTrace IP, TCP, and UDP providers; the DTrace JavaScript provider; and the ZFS L2ARC. Many of Brendan's DTrace scripts are shipped by default in Mac OS X.

Jim Mauro is a senior software engineer for Oracle Corporation. Jim works in the Systems group, with a primary focus on systems performance. Jim's work includes internal performance-related projects, as well as working with Oracle customers on diagnosing performance issues on production systems. Jim has 30 years of experience in the computer industry, including 19 years with Sun Microsystems prior to the acquisition by Oracle. Jim has used DTrace extensively for his performance work since it was first introduced in Solaris 10 and has taught Solaris performance analysis and DTrace for many years.

Jim coauthored the first and second editions of *Solaris Internals* (McDougall and Mauro, 2006) and *Solaris Performance and Tools* (McDougall, Mauro, and Gregg, 2006) and has written numerous articles and white papers on various aspects of Solaris performance and internals.

9

Applications

DTrace has the ability to follow the operation of applications from within the application source code, through system libraries, through system calls, and into the kernel. This visibility allows the root cause of issues (including performance issues) to be found and quantified, even if it is internal to a kernel device driver or something else outside the boundaries of the application code. Using DTrace, questions such as the following can be answered.

- What transactions are occurring? With what latency?
- What disk I/O is the application performing? What network I/O?
- Why is the application on-CPU?

As an example, the following one-liner frequency counts application stack traces when the Apache Web server (`httpd`) performs the `read()` system call:

```
# dtrace -n 'syscall::read:entry /execname == "httpd"/ { @[ustack()] = count(); }'
dtrace: description 'syscall::read:entry ' matched 1 probe
[...]

              libc.so.1`__read+0x7
              libapr-1.so.0.3.9`apr_socket_recv+0xb0
              libaprutil-1.so.0.3.9`socket_bucket_read+0x5b
              httpd`ap_core_input_filter+0x294
              mod_ssl.so`bio_filter_in_read+0xbc
              libcrypto.so.0.9.8`BIO_read+0xaf
              libssl.so.0.9.8`ssl3_get_record+0xb5
              libssl.so.0.9.8`ssl3_read_n+0x144
```

continues

```
                libssl.so.0.9.8`ssl3_read_bytes+0x161
                libssl.so.0.9.8`ssl3_read_internal+0x66
                libssl.so.0.9.8`ssl3_read+0x16
                libssl.so.0.9.8`SSL_read+0x42
                mod_ssl.so`ssl_io_input_read+0xf0
                mod_ssl.so`ssl_io_filter_input+0xd0
                httpd`ap_rgetline_core+0x66
                httpd`ap_read_request+0x1d1
                httpd`ap_process_http_connection+0xe4
                httpd`ap_run_process_connection+0x28
                httpd`child_main+0x3d8
                httpd`make_child+0x86
                httpd`ap_mpm_run+0x410
                httpd`main+0x812
                httpd`_start+0x7d
                 31
```

The output has been truncated to show only the last stack trace. This stack trace was responsible for calling `read()` 31 times and shows the application code path through libssl (the Secure Sockets Layer library, because this was an HTTPS read). Each of the functions shown by the stack trace can be traced separately using DTrace, including function arguments, return value, and time.

The previous chapter focused on the programming languages of application software, particularly for developers who have access to the source code. This chapter focuses on application analysis for end users, regardless of language or layer in the software stack.

Capabilities

DTrace is capable of tracing every layer of the software stack, including examining the interactions of the various layers (see Figure 9-1).

Strategy

To get started using DTrace to examine applications, follow these steps (the target of each step is in bold):

1. Try the DTrace **one-liners** and **scripts** listed in the sections that follow and from the other chapters in the "See Also" section (which includes disk, file system, and network I/O).
2. In addition to those DTrace tools, familiarize yourself with any existing **application logs** and **statistics** that are available and also by any add-ons. (For example, before diving into Mozilla Firefox performance, try add-ons for performance analysis.) The information that these retrieve can show what is useful to investigate further with DTrace.

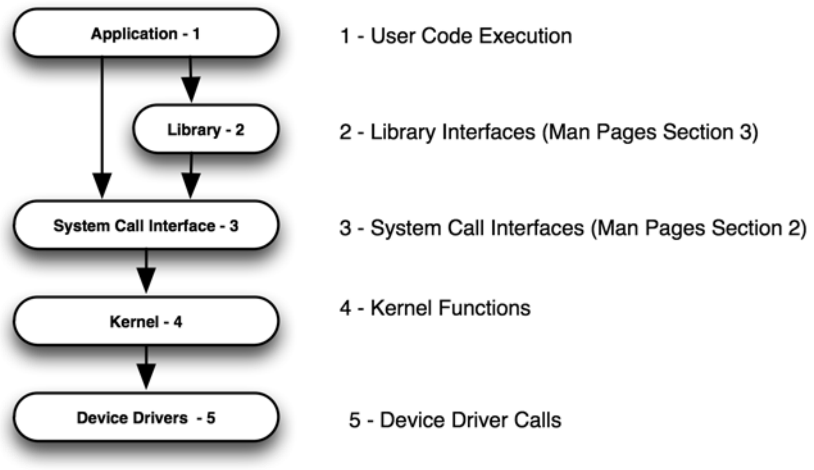


Figure 9-1 Software stack

3. Check whether any application **USDT providers** are available (for example, the mozilla provider for Mozilla Firefox).
4. Examine application behavior using the **syscall** provider, especially if the application has a high system CPU time. This is often an effective way to get a high-level picture of what the application is doing by examining what it is requesting the kernel to do. System call entry arguments and return errors can be examined for troubleshooting issues, and system call latency can be examined for performance analysis.
5. Examine application behavior in the context of **system resources**, such as CPUs, disks, file systems, and network interfaces. Refer to the appropriate chapter in this book.
6. Write tools to generate **known workloads**, such as performing a client transaction. It can be extremely helpful to have a known workload to refer to while developing DTrace scripts.
7. Familiarize yourself with application internals. Sources may include application documentation and source code, if available. DTrace can also be used to learn the internals of an application, such as by examining **stack traces** whenever the application performs I/O (see the example at the start of this chapter).
8. Use a **language provider** to trace application code execution, if one exists and is available (for example, perl). See Chapter 8, Languages.

9. Use the **pid provider** to trace the internals of the application software and libraries it uses, referring to the source code if available. Write scripts to examine higher-level details first (operation counts), and drill down deeper into areas of interest.

Checklist

Consider Table 9-1 a checklist of application issue types that can be examined using DTrace. This is similar to the checklist in Chapter 8 but is in terms of applications rather than the language.

Table 9-1 Applications Checklist

Issue	Description
on-CPU time	An application is hot on-CPU, showing high %CPU in `top(1)` or `prstat(1M)`. DTrace can identify the reason by sampling user stack traces with the profile provider and by tracing application functions with vtimestamps. Reasons for high on-CPU time may include the following: • Compression • Encryption • Dataset iteration (code path loops) • Spin lock contention • Memory I/O The actual make-up of CPU time, whether it is cycles on core (for example, for the Arithmetic Logic Unit) or cycles while stalled (for example, waiting for memory bus I/O) can be investigated further using the DTrace cpc provider, if available.
off-CPU time	Applications will spend time off-CPU while waiting for I/O, waiting for locks (not spinning), and while waiting to be dispatched on a CPU after returning to the ready to run state. These events can be examined and timed with DTrace, such as by using the sched provider to look at thread events. Time off-CPU during I/O, especially disk or network I/O, is a common cause of performance issues (for example, an application performing file system reads served by slow disks, or a DNS lookup during client login, waiting on network I/O to the DNS server). When interpreting off-CPU time, it is important to differentiate between time spent off-CPU because of the following: • Waiting on I/O during an application transaction • Waiting for work to do Applications may spend most of their time waiting for work to do, which is not typically a problem.

Table 9-1 Applications Checklist (*Continued*)

Issue	Description
Volume	Applications may be calling a particular function or code path too frequently; this is the simplest type of issue to DTrace: frequency count function calls. Examining function arguments may identify other inefficiencies, such as performing I/O with small byte sizes when larger sizes should be possible.
Locks	Waiting on locks can occur both on-CPU (spin) and off-CPU (wait). Locks are used for synchronization of multithreaded applications and, when poorly used, can cause application latency and thread serialization. Use DTrace to examine lock usage using the plockstat provider if available or using pid or profile.
Memory Allocation	Memory allocation can be examined in situations when applications consume excessive amounts of memory. Calls to manage memory (such as `malloc()`) can be traced, along with entry and return arguments.
Errors	Applications can encounter errors in their own code and from system libraries and system calls that they execute. Encountering errors is normal for software, which should be written to handle them correctly. However, it is possible that errors are being encountered but not handled correctly by the application. DTrace can be used to examine whether errors are occurring and, if so, their origin.

Providers

Table 9-2 shows providers of most interest when tracing applications.

Table 9-2 Providers for Applications

Provider	Description
proc	Trace application process and thread creation and destruction and signals.
syscall	Trace entry and return of operating system calls, arguments, and return values.
profile	Sample application CPU activity at a custom rate.
sched	Trace application thread scheduling events.
vminfo	Virtual memory statistic probes, based on `vmstat(1M)` statistics.
sysinfo	Kernel statistics probes, based on `mpstat(1M)` statistics.
plockstat	Trace user-land lock events.
cpc	CPU Performance Counters provider, for CPU cache hit/miss by function.
pid	Trace internals of the application including calls to system libraries.
language	Specific language provider: See Chapter 8.

You can find complete lists of provider probes and arguments in the DTrace Guide.[1]

pid Provider

The Process ID (pid) provider instruments user-land function execution, providing probes for function entry and return points and for every instruction in the function. It also provides access to function arguments, return codes, return instruction offsets, and register values. By tracing function entry and return, the elapsed time and on-CPU time during function execution can also be measured. It is available on Solaris and Mac OS X and is currently being developed for FreeBSD.[2]

The pid provider is associated with a particular process ID, which is part of the provider name: pid<PID>. The PID can be written literally, such as pid123, or specified using the macro variable `$target`, which provides the PID when either the `-p PID` or `-c command` option is used.

Listing pid provider function entry probes for the bash shell (running as PID 1122) yields the following:

```
# dtrace -ln 'pid$target:::entry' -p 1122
   ID   PROVIDER               MODULE                         FUNCTION NAME
12539    pid1122                 bash                           _start entry
12540    pid1122                 bash                            __fsr entry
12541    pid1122                 bash                             main entry
12542    pid1122                 bash               parse_long_options entry
12543    pid1122                 bash              parse_shell_options entry
12544    pid1122                 bash                       exit_shell entry
12545    pid1122                 bash                          sh_exit entry
12546    pid1122                 bash                 execute_env_file entry
12547    pid1122                 bash                run_startup_files entry
12548    pid1122                 bash              shell_is_restricted entry
12549    pid1122                 bash            maybe_make_restricted entry
12550    pid1122                 bash                           uidget entry
12551    pid1122                 bash                disable_priv_mode entry
12552    pid1122                 bash                      run_wordexp entry
12553    pid1122                 bash                  run_one_command entry
[...]
15144    pid1122       libcurses.so.1                           addstr entry
15145    pid1122       libcurses.so.1                          attroff entry
15146    pid1122       libcurses.so.1                           attron entry
15147    pid1122       libcurses.so.1                          attrset entry
15148    pid1122       libcurses.so.1                             beep entry
15149    pid1122       libcurses.so.1                             bkgd entry
[...]
15704    pid1122       libsocket.so.1                        endnetent entry
15705    pid1122       libsocket.so.1                      getnetent_r entry
15706    pid1122       libsocket.so.1                       str2netent entry
15707    pid1122       libsocket.so.1                   getprotobyname entry
```

---

[1] This is currently at *http://wikis.sun.com/display/DTrace/Documentation*.

[2] This is by Rui Paulo for the DTrace user-land project: *http://freebsdfoundation.blogspot.com/2010/06/dtrace-userland-project.html*.

```
15708    pid1122    libsocket.so.1                    getprotobynumber entry
15709    pid1122    libsocket.so.1                        getprotoent entry
[...]
19019    pid1122         libc.so.1                              fopen entry
19020    pid1122         libc.so.1                       _freopen_null entry
19021    pid1122         libc.so.1                            freopen entry
19022    pid1122         libc.so.1                            fgetpos entry
19023    pid1122         libc.so.1                            fsetpos entry
19024    pid1122         libc.so.1                              fputc entry
[...]
```

There were 8,003 entry probes listed. The previous truncated output shows a sample of the available probes from the bash code segment and three libraries: libcurses, libsocket, and libc. The probe module name is the segment name.

Listing all pid provider probes for the libc function `fputc()` yields the following:

```
# dtrace -ln 'pid$target::fputc:' -p 1122
   ID   PROVIDER            MODULE                          FUNCTION NAME
19024    pid1122         libc.so.1                             fputc entry
20542    pid1122         libc.so.1                             fputc return
20543    pid1122         libc.so.1                             fputc 0
20544    pid1122         libc.so.1                             fputc 1
20545    pid1122         libc.so.1                             fputc 3
20546    pid1122         libc.so.1                             fputc 4
20547    pid1122         libc.so.1                             fputc 7
20548    pid1122         libc.so.1                             fputc c
20549    pid1122         libc.so.1                             fputc d
20550    pid1122         libc.so.1                             fputc 13
20551    pid1122         libc.so.1                             fputc 16
20552    pid1122         libc.so.1                             fputc 19
20553    pid1122         libc.so.1                             fputc 1c
20554    pid1122         libc.so.1                             fputc 21
20555    pid1122         libc.so.1                             fputc 24
20556    pid1122         libc.so.1                             fputc 25
20557    pid1122         libc.so.1                             fputc 26
```

The probes listed are the entry and return probes for the `fputc()` function, as well as probes for each instruction offset in hexadecimal (0, 1, 3, 4, 7, c, d, and so on).

Be careful when using the pid provider, especially in production environments. Application processes vary greatly in size, and many production applications have large text segments with a large number of instrumentable functions, each with tens to hundreds of instructions and with each instruction another potential probe target for the pid provider. The invocation `dtrace -n 'pid1234:::'` will instruct DTrace to instrument every function entry and return and to instrument every instruction in process PID 1234. Here's an example:

```
solaris# dtrace -n 'pid1471:::'
dtrace: invalid probe specifier pid1471:::: failed to create offset probes in
'__1cFStateM_sub_Op_ConI6MpknENode__v_': Not enough space

solaris# dtrace -n 'pid1471:::entry'
dtrace: description 'pid1471:::entry' matched 26847 probes
```

Process PID 1471 was a Java JVM process. The first DTrace command attempted to insert a probe at every instruction location in the JVM but was unable to complete. The `Not enough space` error means the default number of 250,000 pid provider probes was not enough to complete the instrumentation. The second invocation in the example instruments the same process, but this time with the entry string in the name component of the probe, instructing DTrace to insert a probe at the entry point of every function in the process. In this case, DTrace found 26,847 instrumentation points.

Once a process is instrumented with the pid provider, depending on the number of probes and how busy the process is, using the pid provider will induce some probe effect, meaning it can slow the execution speed of the target process, in some cases dramatically.

Stability

The pid provider is considered an *unstable* interface, meaning that the provider interface (which consists of the probe names and arguments) may be subject to change between application software versions. This is because the interface is dynamically constructed based on the thousands of compiled functions that make up a software application. It is these functions that are subject to change, and when they do, so does the pid provider. This means that any DTrace scripts or one-liners based on the pid provider may be dependent on the application software version they were written for.

Although application software can and is likely to change between versions, many library interfaces are likely to remain unchanged, such as libc, libsocket, libpthread, and many others, especially those exporting standard interfaces such as POSIX. These can make good targets for tracing with the pid provider, because one-liners and scripts will have a higher degree of stability than when tracing application-specific software.

If a pid-based script has stopped working because of minor software changes, then ideally the script can be repaired with equivalent minor changes to match the newer software. If the software has changed significantly, then the pid-based script may need to be rewritten entirely. Because of this instability, it is recommended to use pid only when needed. If there are stable providers available that can serve a similar role, they should be used instead, and the scripts that use them will not need to be rewritten as the software changes.

Since pid is an unstable interface, the pid provider one-liners and scripts in this book are not guaranteed to work or be supported by software vendors.

The pid provider scripts in this book serve not just as examples of using the pid provider in D programs but also as example data that DTrace can make available and why that can be useful. If these scripts stop working, you can try fixing them or check for updated versions on the Web (try this book's Web site, *www.dtracebook.com*).

Arguments and Return Value

The arguments and return value for functions can be inspected on the pid entry and return probes.

- `pid<PID>:::entry`: The function arguments is (`uint64_t`) `arg0 ... argn`.
- `pid<PID>:::return`: The program counter is (`uint64_t`) `arg0`; the return value is (uint64_t) `arg1`.

The `uregs[]` array can also be accessed to examine individual user registers.

cpc Provider

The CPU Performance Counter (cpc) provider provides probes for profiling CPU events, such as instructions, cache misses, and stall cycles. These CPU events are based on the performance counters that the CPUs provide, which vary between manufacturers, types, and sometimes versions of the same type of CPU. A generic interface for the performance counters has been developed, the Performance Application Programming Interface (PAPI),[3] which is supported by the cpc provider in addition to the platform-specific counters. The cpc provider is fully documented in the cpc provider section of the DTrace Guide and is currently available only in Solaris Nevada.[4]

The cpc provider probe names have the following format:

```
cpc:::<event name>-<mode>-<optional mask-><count>
```

The event name may be a PAPI name or a platform-specific event name. On Solaris, events for the current CPU type can be listed using `cpustat(1M)`:

```
solaris# cpustat -h
Usage:
        cpustat [-c events] [-p period] [-nstD] [-T d|u] [interval [count]]
[...]
        Generic Events:
```

continues

---

[3] See *http://icl.cs.utk.edu/papi*.

[4] This was integrated in snv_109, defined by PSARC 2008/480, and developed by Jon Haslam. See his blog post about cpc, currently at *http://blogs.sun.com/jonh/entry/finally_dtrace_meets_the_cpu*.

```
        event[0-3]: PAPI_br_ins PAPI_br_msp PAPI_br_tkn PAPI_fp_ops
                 PAPI_fad_ins PAPI_fml_ins PAPI_fpu_idl PAPI_tot_cyc
                 PAPI_tot_ins PAPI_l1_dca PAPI_l1_dcm PAPI_l1_ldm
                 PAPI_l1_stm PAPI_l1_ica PAPI_l1_icm PAPI_l1_icr
                 PAPI_l2_dch PAPI_l2_dcm PAPI_l2_dcr PAPI_l2_dcw
                 PAPI_l2_ich PAPI_l2_icm PAPI_l2_ldm PAPI_l2_stm
                 PAPI_res_stl PAPI_stl_icy PAPI_hw_int PAPI_tlb_dm
                 PAPI_tlb_im PAPI_l3_dcr PAPI_l3_icr PAPI_l3_tcr
                 PAPI_l3_stm PAPI_l3_ldm PAPI_l3_tcm

        See generic_events(3CPC) for descriptions of these events

        Platform Specific Events:

        event[0-3]: FP_dispatched_fpu_ops FP_cycles_no_fpu_ops_retired
                 FP_dispatched_fpu_ops_ff LS_seg_reg_load
                 LS_uarch_resync_self_modify LS_uarch_resync_snoop
                 LS_buffer_2_full LS_locked_operation LS_retired_cflush
                 LS_retired_cpuid DC_access DC_miss DC_refill_from_L2
                 DC_refill_from_system DC_copyback DC_dtlb_L1_miss_L2_hit
                 DC_dtlb_L1_miss_L2_miss DC_misaligned_data_ref
[...]
        See "BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h
        Processors" (AMD publication 31116)
```

The first group, `Generic Events`, is the PAPI events and is documented on Solaris in the generic_events(3CPC) man page. The second group, `Platform Specific Events`, is from the CPU manufacturer and is typically documented in the CPU user guide referenced in the `cpustat(1M)` output.

The mode component of the probe name can be `user` for profiling user-mode, `kernel` for kernel-mode, or `all` for both.

The optional mask component is sometimes used by platform-specific events, as directed by the CPU user guide.

The final component of the probe name is the overflow count: Once this many of the specified event has occurred on the CPU, the probe fires on that CPU. For frequent events, such as cycle and instruction counts, this can be set to a high number to reduce the rate that the probe fires and therefore reduce the impact on target application performance.

cpc provider probes have two arguments: `arg0` is the kernel program counter or 0 if not executing in the kernel, and `arg1` is the user-level program counter or 0 if not executing in user-mode.

Depending on the CPU type, it may not be possible to enable more than one cpc probe simultaneously. Subsequent enablings will encounter a `Failed to enable probe` error. This behavior is similar to, and for the same reason as, the operating system, allowing only one invocation of `cpustat(1M)` at a time. There is a finite number of performance counter registers available for each CPU type.

The sections that follow have example cpc provider one-liners and output.

See Also

There are many topics relevant to application analysis, most of which are covered fully in separate chapters of this book.

- Chapter 3: System View
- Chapter 4: Disk I/O
- Chapter 5: File Systems
- Chapter 6: Network Lower-Level Protocols
- Chapter 7: Application-Level Protocols
- Chapter 8: Languages

All of these can be considered part of this chapter. The one-liners and scripts that follow summarize application analysis with DTrace and introduce some remaining topics such as signals, thread scaling, and the cpc provider.

One-Liners

For many of these, a Web server with processes named `httpd` is used as the target application. Modify `httpd` to be the name of the application process of interest.

proc provider

Trace new processes:

```
dtrace -n 'proc:::exec-success { trace(execname); }'
```

Trace new processes (current FreeBSD[5]):

```
dtrace -n 'proc:::exec_success { trace(execname); }'
```

New processes (with arguments):

```
dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
```

---

[5] FreeBSD 8.0; this will change to become `exec-success` (consistent with Solaris and Mac OS X), now that support for hyphens in FreeBSD probe names is being developed.

New threads created, by process:

```
dtrace -n 'proc:::lwp-create { @[pid, execname] = count(); }'
```

Successful signal details:

```
dtrace -n 'proc:::signal-send { printf("%s -%d %d", execname, args[2], args[1]->pr_pid); }'
```

syscall provider

System call counts for processes named `httpd`:

```
dtrace -n 'syscall:::entry /execname == "httpd"/ { @[probefunc] = count(); }'
```

System calls with non-zero `errno` (errors):

```
dtrace -n 'syscall:::return /errno/ { @[probefunc, errno] = count(); }'
```

profile provider

User stack trace profile at 101 Hertz, showing process name and stack:

```
dtrace -n 'profile-101 { @[execname, ustack()] = count(); }'
```

User stack trace profile at 101 Hertz, showing process name and top five stack frames:

```
dtrace -n 'profile-101 { @[execname, ustack(5)] = count(); }'
```

User stack trace profile at 101 Hertz, showing process name and stack, top ten only:

```
dtrace -n 'profile-101 { @[execname, ustack()] = count(); } END { trunc(@, 10); }'
```

User stack trace profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[ustack()] = count(); }'
```

User function name profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[ufunc(arg1)] = count(); }'
```

User module name profile at 101 Hertz for processes named `httpd`:

```
dtrace -n 'profile-101 /execname == "httpd"/ { @[umod(arg1)] = count(); }'
```

sched provider

Count user stack traces when processes named `httpd` leave CPU:

```
dtrace -n 'sched:::off-cpu /execname == "httpd"/ { @[ustack()] = count(); }'
```

pid provider

The pid provider instruments functions from a particular software version; these example one-liners may therefore require modifications to match the software version you are running. They can be executed on an existing process by using `-p PID` or by running a new process using `-c command`.

Count process segment function calls:

```
dtrace -n 'pid$target:a.out::entry { @[probefunc] = count(); }' -p PID
```

Count libc function calls:

```
dtrace -n 'pid$target:libc::entry { @[probefunc] = count(); }' -p PID
```

Count libc string function calls:

```
dtrace -n 'pid$target:libc:str*:entry { @[probefunc] = count(); }' -p PID
```

Trace libc `fsync()` calls showing file descriptor:

```
dtrace -n 'pid$target:libc:fsync:entry { trace(arg0); }' -p PID
```

Trace libc `fsync()` calls showing file path name:

```
dtrace -n 'pid$target:libc:fsync:entry { trace(fds[arg0].fi_pathname); }' -p PID
```

Count requested `malloc()` bytes by user stack trace:

```
dtrace -n 'pid$target::malloc:entry { @[ustack()] = sum(arg0); }' -p PID
```

Trace failed `malloc()` requests:

```
dtrace -n 'pid$target::malloc:return /arg1 == NULL/ { ustack(); }' -p PID
```

See the “C” section of Chapter 8 for more pid provider one-liners.

plockstat provider

As with the pid provider, these can also be run using the `-c` command.

Mutex blocks by user-level stack trace:

```
dtrace -n 'plockstat$target:::mutex-block { @[ustack()] = count(); }' -p PID
```

Mutex spin counts by user-level stack trace:

```
dtrace -n 'plockstat$target:::mutex-acquire /arg2/ { @[ustack()] = sum(arg2); }' -p PID
```

Reader/writer blocks by user-level stack trace:

```
dtrace -n 'plockstat$target:::rw-block { @[ustack()] = count(); }' -p PID
```

cpc provider

These cpc provider one-liners are dependent on the availability of both the cpc provider and the event probes (for Solaris, see `cpustat(1M)` to learn what events are available on your system). The following overflow counts (200,000; 50,000; and 10,000) have been picked to balance between the rate of events and fired DTrace probes.

User-mode instructions by process name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname] = count(); }'
```

User-mode instructions by process name and function name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode instructions for processes named `httpd` by function name:

```
dtrace -n 'cpc:::PAPI_tot_ins-user-200000 /execname == "httpd"/ { @[ufunc(arg1)] =
count(); }'
```

User-mode CPU cycles by process name and function name:

```
dtrace -n 'cpc:::PAPI_tot_cyc-user-200000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one instruction cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_icm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-one data cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l1_dcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-two cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode level-three cache misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_l3_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode conditional branch misprediction by process name and function name:

```
dtrace -n 'cpc:::PAPI_br_msp-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode resource stall cycles by process name and function name:

```
dtrace -n 'cpc:::PAPI_res_stl-user-50000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode floating-point operations by process name and function name:

```
dtrace -n 'cpc:::PAPI_fp_ops-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

User-mode TLB misses by process name and function name:

```
dtrace -n 'cpc:::PAPI_tlb_tl-user-10000 { @[execname, ufunc(arg1)] = count(); }'
```

One-Liner Selected Examples

There are additional examples of one-liners in the "Case Study" section.

New Processes (with Arguments)

New processes were traced on Solaris while the `man ls` command was executed:

```
solaris# dtrace -n 'proc:::exec-success { trace(curpsinfo->pr_psargs); }'
dtrace: description 'proc:::exec-success ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  13487        exec_common:exec-success   man ls
  0  13487        exec_common:exec-success   sh -c cd /usr/share/man; tbl /usr/share/
man/man1/ls.1 |neqn /usr/share/lib/pub/
```

```
   0  13487         exec_common:exec-success   tbl /usr/share/man/man1/ls.1
   0  13487         exec_common:exec-success   neqn /usr/share/lib/pub/eqnchar -
   0  13487         exec_common:exec-success   nroff -u0 -Tlp -man -
   0  13487         exec_common:exec-success   col -x
   0  13487         exec_common:exec-success   sh -c trap '' 1 15; /usr/bin/mv -f /tmp/
mpcJaP5g /usr/share/man/cat1/ls.1 2> /d
   0  13487         exec_common:exec-success   /usr/bin/mv -f /tmp/mpcJaP5g /usr/share/
man/cat1/ls.1
   0  13487         exec_common:exec-success   sh -c more -s /tmp/mpcJaP5g
   0  13487         exec_common:exec-success   more -s /tmp/mpcJaP5g
^C
```

The variety of programs that are executed to process `man ls` are visible, ending with the `more(1)` command that shows the man page.

Mac OS X currently doesn't provide the full argument list in `pr_psargs`, which is noted in the comments of the curpsinfo translator:

```
macosx# grep pr_psargs /usr/lib/dtrace/darwin.d
      char pr_psargs[80];     /* initial characters of arg list */
      pr_psargs = P->p_comm; /* XXX omits command line arguments XXX */
      pr_psargs = xlate <psinfo_t> ((struct proc *)(T->task->bsd_info)).pr_psargs; /*

XXX omits command line arguments XXX */
```

And using `pr_psargs` in `trace()` on Mac OS X can trigger `tracemem()` behavior, printing hex dumps from the address, which makes reading the output a little difficult. It may be easier to just use the `execname` for this one-liner for now. Here's an example of tracing `man ls` on Mac OS X:

```
macosx# dtrace -n 'proc:::exec-success { trace(execname); }'
dtrace: description 'proc:::exec-success ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  19374         posix_spawn:exec-success   sh
  0  19374         posix_spawn:exec-success   sh
  0  19368        __mac_execve:exec-success   sh
  0  19368        __mac_execve:exec-success   tbl
  0  19368        __mac_execve:exec-success   sh
  0  19368        __mac_execve:exec-success   grotty
  0  19368        __mac_execve:exec-success   more
  1  19368        __mac_execve:exec-success   man
  1  19368        __mac_execve:exec-success   sh
  1  19368        __mac_execve:exec-success   gzip
  1  19368        __mac_execve:exec-success   gzip
  1  19374         posix_spawn:exec-success   sh
  1  19368        __mac_execve:exec-success   groff
  1  19368        __mac_execve:exec-success   troff
  1  19368        __mac_execve:exec-success   gzip
^C
```

Note that the output is shuffled (the CPU ID change is a hint). For the correct order, include a time stamp in the output and postsort.

System Call Counts for Processes Called httpd

The Apache Web server runs multiple `httpd` processes to serve Web traffic. This can be a problem for traditional system call debuggers (such as `truss(1)`), which can examine only one process at a time, usually by providing a process ID. DTrace can examine all processes simultaneously, making it especially useful for multiprocess applications such as Apache.

This one-liner frequency counts system calls from all running Apache `httpd` processes:

```
solaris# dtrace -n 'syscall:::entry /execname == "httpd"/ { @[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 225 probes
^C

  accept                                                            1
  getpid                                                            1
  lwp_mutex_timedlock                                               1
  lwp_mutex_unlock                                                  1
  shutdown                                                          1
  brk                                                               4
  gtime                                                             5
  portfs                                                            7
  mmap64                                                           10
  waitsys                                                          30
  munmap                                                           33
  doorfs                                                           39
  openat                                                           49
  writev                                                           51
  stat64                                                           60
  close                                                            61
  fcntl                                                            73
  read                                                             74
  lwp_sigmask                                                      78
  getdents64                                                       98
  pollsys                                                         100
  fstat64                                                         109
  open64                                                          207
  lstat64                                                         245
```

The most frequently called system call was `lstat64()`, called 245 times.

User Stack Trace Profile at 101 Hertz, Showing Process Name and Top Five Stack Frames

This one-liner is a quick way to see not just who is on-CPU but what they are doing:

```
solaris# dtrace -n 'profile-101 { @[execname, ustack(5)] = count(); }'
dtrace: description 'profile-101 ' matched 1 probe
^C
[...]
  mpstat
              libc.so.1`p_online+0x7
```

```
              mpstat`acquire_snapshot+0x131
              mpstat`main+0x27d
              mpstat`_start+0x7d
               13
  httpd
              libc.so.1`__forkx+0xb
              libc.so.1`fork+0x1d
              mod_php5.2.so`zif_proc_open+0x970
              mod_php5.2.so`execute_internal+0x45
              mod_php5.2.so`dtrace_execute_internal+0x59
               42
  sched
              541
```

No stack trace was shown for sched (the kernel), since this one-liner is examining user-mode stacks (`ustack()`), not kernel stacks (`stack()`). This could be eliminated from the output by adding the predicate `/arg1/` (check that the user-mode program counter is nonzero) to ensure that only user stacks are sampled.

User-Mode Instructions by Process Name

To introduce this one-liner, a couple of test applications were written and executed called app1 and app2, each single-threaded and running a continuous loop of code. Examining these applications using `top(1)` shows the following:

```
last pid:  4378;  load avg:  2.13,  2.00,  1.62;  up 4+02:53:19       06:24:05
98 processes: 95 sleeping, 3 on cpu
CPU states: 73.9% idle, 25.2% user,  0.9% kernel,  0.0% iowait,  0.0% swap
Kernel: 866 ctxsw, 19 trap, 1884 intr, 2671 syscall
Memory: 32G phys mem, 1298M free mem, 4096M total swap, 4096M free swap

   PID USERNAME NLWP PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
  4319 root        1  10    0 1026M  513M cpu/3   10:50 12.50% app2
  4318 root        1  10    0 1580K  808K cpu/7   10:56 12.50% app1
[...]
```

`top(1)` reports that each application is using 12.5 percent of the total CPU capacity, which is a single core on this eight-core system. The Solaris `prstat -mL` breaks down the CPU time into microstates and shows this in terms of a single thread:

```
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  4318 root     100 0.0 0.0 0.0 0.0 0.0 0.0 0.0   0   8   0   0 app1/1
  4319 root     100 0.0 0.0 0.0 0.0 0.0 0.0 0.0   0   8   0   0 app2/1
[...]
```

`prstat(1M)` shows that each thread is running at 100 percent user time (USR). This is a little more information than simply %CPU from `top(1)`, and it indicates that these applications are both spending time executing their own code.

The cpc provider allows %CPU time to be understood in greater depth. This one-liner uses the cpc provider to profile instructions by process name. The probe specified fires for every 200,000th user-level instruction, counting the current process name at the time:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-user-200000 { @[execname] = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-user-200000 ' matched 1 probe
^C

  sendmail                                                          1
  dtrace                                                            2
  mysqld                                                            6
  sshd                                                              7
  nscd                                                             14
  httpd                                                            16
  prstat                                                           23
  mpstat                                                           52
  app2                                                            498
  app1                                                         154801
```

So, although the output from `top(1)` and `prstat(1M)` suggests that both applications are very similar in terms of CPU usage, the cpc provider shows that they are in fact very different. During the same interval, app1 executed roughly 300 times more CPU instructions than app2.

The other cpc one-liners can explain this further; app1 was written to continually execute fast register-based instructions, while app2 continually performs much slower main memory I/O.

User-Mode Instructions for Processes Named httpd by Function Name

This one-liner matches processes named `httpd` and profiles instructions by function, counting on every 200,000th instruction:

```
solaris# dtrace -n 'cpc:::PAPI_tot_ins-user-200000 /execname == "httpd"/ {
@[ufunc(arg1)]  = count(); }'
dtrace: description 'cpc:::PAPI_tot_ins-user-200000 ' matched 1 probe
^C

  httpd`ap_invoke_handler                                           1
  httpd`pcre_exec                                                   1
  libcrypto.so.0.9.8`SHA1_Update                                    1
[...]
  libcrypto.so.0.9.8`bn_sqr_comba8                                 39
  libz.so.1`crc32_little                                           41
  libcrypto.so.0.9.8`sha1_block_data_order                         50
  libcrypto.so.0.9.8`_x86_AES_encrypt                              88
  libz.so.1`compress_block                                        103
  libcrypto.so.0.9.8`bn_mul_add_words                             117
  libcrypto.so.0.9.8`bn_mul_add_words                             127
  libcrypto.so.0.9.8`bn_mul_add_words                             133
  libcrypto.so.0.9.8`bn_mul_add_words                             134
```

```
    libz.so.1`fill_window                                          222
    libz.so.1`deflate_slow                                         374
    libz.so.1`longest_match                                       1022
```

The functions executing the most instructions are in the libz library, which performs compression.

User-Mode Level-Two Cache Misses by Process Name and Function Name

This example is included to suggest what to do when encountering this error:

```
solaris# dtrace -n 'cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
dtrace: invalid probe specifier cpc:::PAPI_l2_tcm-user-10000 { @[execname, ufunc(arg1)] =
 count(); }: probe description cpc:::PAPI_l2_tcm-user-10000 does not match any probes
```

This system does have the cpc provider; however, this probe is invalid. After checking for typos, check whether the event name is supported on this system using `cpustat(1M)` (Solaris):

```
solaris# cpustat -h
Usage:
        cpustat [-c events] [-p period] [-nstD] [-T d|u] [interval [count]]
[...]
        Generic Events:

        event[0-3]: PAPI_br_ins PAPI_br_msp PAPI_br_tkn PAPI_fp_ops
                 PAPI_fad_ins PAPI_fml_ins PAPI_fpu_idl PAPI_tot_cyc
                 PAPI_tot_ins PAPI_l1_dca PAPI_l1_dcm PAPI_l1_ldm
                 PAPI_l1_stm PAPI_l1_ica PAPI_l1_icm PAPI_l1_icr
                 PAPI_l2_dch PAPI_l2_dcm PAPI_l2_dcr PAPI_l2_dcw
                 PAPI_l2_ich PAPI_l2_icm PAPI_l2_ldm PAPI_l2_stm
                 PAPI_res_stl PAPI_stl_icy PAPI_hw_int PAPI_tlb_dm
                 PAPI_tlb_im PAPI_l3_dcr PAPI_l3_icr PAPI_l3_tcr
                 PAPI_l3_stm PAPI_l3_ldm PAPI_l3_tcm

        See generic_events(3CPC) for descriptions of these events

        Platform Specific Events:

        event[0-3]: FP_dispatched_fpu_ops FP_cycles_no_fpu_ops_retired
[...]
```

This output shows that the PAPI_l2_tcm event (level-two cache miss) is not supported on this system. However, it also shows that PAPI_l2_dcm (level-two data cache miss) and PAPI_l2_icm (level-two instruction cache miss) are supported. Adjusting the one-liner for, say, data cache misses only is demonstrated by the following one-liner:

```
solaris# dtrace -n 'cpc:::PAPI_l2_dcm-user-10000 { @[execname, ufunc(arg1)] = count(); }'
dtrace: description 'cpc:::PAPI_l2_dcm-user-10000 ' matched 1 probe
^C

  dtrace                    libproc.so.1`byaddr_cmp                                  1
  dtrace                    libproc.so.1`symtab_getsym                               1
  dtrace                    libc.so.1`memset                                         1
  mysqld                    mysqld`srv_lock_timeout_and_monitor_thread               1
  mysqld                    mysqld`sync_array_print_long_waits                       1
  dtrace                    libproc.so.1`byaddr_cmp_common                           2
  dtrace                    libc.so.1`qsort                                          2
  dtrace                    libproc.so.1`optimize_symtab                             3
  dtrace                    libproc.so.1`byname_cmp                                  6
  dtrace                    libc.so.1`strcmp                                        17
  app2                      app2`main                                              399
```

This one-liner can then be run for instruction cache misses so that both types of misses can be considered.

Should the generic PAPI events be unavailable or unsuitable, the platform-specific events (as listed by `cpustat(1M)`) may allow the event to be examined, albeit in a way that is tied to the current CPU version.

Scripts

Table 9-3 summarizes the scripts that follow and the providers they use.

procsnoop.d

This is a script version of the "New Processes" one-liner shown earlier. Tracing the execution of new processes provides important visibility for applications that call

Table 9-3 Application Script Summary

Script	Description	Provider
`procsnoop`	Snoop process execution	proc
`procsystime`	System call time statistics by process	syscall
`uoncpu.d`	Profile application on-CPU user stacks	profile
`uoffcpu.d`	Count application off-CPU user stacks by time	sched
`plockstat`	User-level mutex and read/write lock statistics	plockstat
`kill.d`	Snoop process signals	syscall
`sigdist.d`	Signal distribution by source and destination processes	syscall
`threaded.d`	Sample multithreaded CPU usage	profile

the command line; some applications can call shell commands so frequently that it becomes a performance issue—one that is difficult to spot in traditional tools (such as `prstat(1M)` and `top(1)`) because the processes are so short-lived.

Script

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option switchrate=10hz

dtrace:::BEGIN
{
        printf("%-8s %5s %6s %6s %s\n", "TIME(ms)", "UID", "PID", "PPID",
            "COMMAND");
        start = timestamp;
}

proc:::exec-success
{
        printf("%-8d %5d %6d %6d %s\n", (timestamp - start) / 1000000,
            uid, pid, ppid, curpsinfo->pr_psargs);
}
```

Script procsnoop.d

Example

The following shows the Oracle Solaris commands executed as a consequence of restarting the `cron` daemon via `svcadm(1M)`:

```
solaris# procsnoop.d
TIME(ms)   UID    PID   PPID COMMAND
3227         0  13273  12224 svcadm restart cron
3709         0  13274    106 /sbin/sh -c exec /lib/svc/method/svc-cron
3763         0  13274    106 /sbin/sh /lib/svc/method/svc-cron
3773         0  13275  13274 /usr/bin/rm -f /var/run/cron_fifo
3782         0  13276  13274 /usr/sbin/cron
```

The `TIME(ms)` column is printed so that the output can be postsorted if desired (DTrace may shuffle the output slightly because it collects buffers from multiple CPUs).

See Also: execsnoop

A program called `execsnoop` exists from the DTraceToolkit, which has similar functionality to that of `procsnoop`. It was written originally for Oracle Solaris and is now shipped on Mac OS X by default. `execsnoop` wraps the D script in the shell so that command-line options are available:

```
macosx# execsnoop -h
USAGE: execsnoop [-a|-A|-ehjsvZ] [-c command]
       execsnoop                # default output
                -a              # print all data
                -A              # dump all data, space delimited
                -e              # safe output, parseable
                -j              # print project ID
                -s              # print start time, us
                -v              # print start time, string
                -Z              # print zonename
                -c command      # command name to snoop
  eg,
        execsnoop -v            # human readable timestamps
        execsnoop -Z          # print zonename
        execsnoop -c ls         # snoop ls commands only
```

`execsnoop` traces process execution by tracing the `exec()` system call (and variants), which do differ slightly between operating systems. Unfortunately, system calls are not a stable interface, even across different versions of the same operating system. Small changes to `execsnoop` have been necessary to keep it working across different versions of Oracle Solaris, because of subtle changes with the names of the `exec()` system calls. The lesson here is to always prefer the stable providers, such as the proc provider (which is stable) instead of syscall (which isn't).

procsystime

`procsystime` is a generic system call time reporter. It can count the execution of system calls, their elapsed time, and on-CPU time and can produce a report showing the system call type and process details. It is from the DTraceToolkit and shipped on Mac OS X by default in `/usr/bin`.

Script

The essence of the script is explained here; the actual script is too long and too uninteresting (mostly dealing with command-line options) to list; see the DTraceToolkit for the full listing.

```
syscall:::entry
/self->ok/
{
      @Counts[probefunc] = count();
      self->start = timestamp;
      self->vstart = vtimestamp;
}

syscall:::return
/self->start/
{
      this->elapsed = timestamp - self->start;
      this->oncpu = vtimestamp - self->vstart;
```

```
        @Elapsed[probefunc] = sum(this->elapsed);
        @CPU[probefunc] = sum(this->cpu);
        self->start = 0;
        self->vstart = 0;
}
```

A `self->ok` variable is set beforehand to true if the current process is supposed to be traced. The code is then straightforward: Time stamps are set on the entry to syscalls so that deltas can be calculated on the return.

Examples

Examples include usage and file system archive.

Usage

Command-line options can be listed using `-h`:

```
solaris# procsystime -h
lox# ./procsystime -h
USAGE: procsystime [-aceho] [ -p PID | -n name | command ]
                  -p PID          # examine this PID
                  -n name         # examine this process name
                  -a              # print all details
                  -e              # print elapsed times
                  -c              # print syscall counts
                  -o              # print CPU times
                  -T              # print totals
  eg,
       procsystime -p 1871     # examine PID 1871
       procsystime -n tar      # examine processes called "tar"
       procsystime -aTn bash   # print all details for bash
       procsystime df -h       # run and examine "df -h"
```

File System Archive

The `tar(1)` command was used to archive a file system, with procsystime tracing elapsed times (which is the default) for processes named `tar`:

```
solaris# procsystime -n tar
Tracing... Hit Ctrl-C to end...
^C

Elapsed Times for processes tar,

         SYSCALL          TIME (ns)
           fcntl              58138
         fstat64              96490
          openat             280246
           chdir            1444153
           write            8922505
          open64           15294117
```

continues

```
          openat64           16804949
             close           17855422
        getdents64           46679462
         fstatat64           98011589
              read         1551039139
```

Most of the elapsed time for the `tar(1)` command was in the `read()` syscall, which is expected because `tar(1)` is reading files from disk (which is slow I/O). The total time spent waiting for `read()` syscalls during the procsystime trace was 1.55 seconds.

uoncpu.d

This is a script version of the DTrace one-liner to profile the user stack trace of a given application process name. As one of the most useful one-liners, it may save typing to provide it as a script, where it can also be more easily enhanced.

Script

```
1       #!/usr/sbin/dtrace -s
2
3       profile:::profile-1001
4       /execname == $$1/
5       {
6             @["\n  on-cpu (count @1001hz):", ustack()] = count();
7       }
```

Script uoncpu.d

Example

Here the `uoncpu.d` script is used to frequency count the user stack trace of all currently running Perl programs. Note `perl` is passed as a command-line argument, evaluated in the predicate (line 4):

```
# uoncpu.d perl
dtrace: script 'uoncpu.d' matched 1 probe
^C
[...output truncated...]

  on-cpu (count @1001hz):
              libperl.so.1`Perl_sv_setnv+0xc8
              libperl.so.1`Perl_pp_multiply+0x3fe
              libperl.so.1`Perl_runops_standard+0x3b
              libperl.so.1`S_run_body+0xfa
              libperl.so.1`perl_run+0x1eb
              perl`main+0x8a
              perl`_start+0x7d
              105
```

```
  on-cpu (count @1001hz):
              libperl.so.1`Perl_pp_multiply+0x3f7
              libperl.so.1`Perl_runops_standard+0x3b
              libperl.so.1`S_run_body+0xfa
              libperl.so.1`perl_run+0x1eb
              perl`main+0x8a
              perl`_start+0x7d
              111
```

The hottest stacks identified include the `Perl_pp_multiply()` function, suggesting that Perl is spending most of its time doing multiplications. Further analysis of those functions and using the perl provider, if available (see Chapter 8), could confirm.

uoffcpu.d

As a companion to `uoncpu.d`, the `uoffcpu.d` script measures the time spent off-CPU by user stack trace. This time includes device I/O, lock wait, and dispatcher queue latency.

Script

```
#!/usr/sbin/dtrace -s

sched:::off-cpu
/execname == $$1/
{
      self->start = timestamp;
}

sched:::on-cpu
 /self->start/
 {
       this->delta = (timestamp - self->start) / 1000;
       @["off-cpu (us):", ustack()] = quantize(this->delta);
       self->start = 0;
 }
```

Script uoffcpu.d

Example

Here the `uoffcpu.d` script was used to trace CPU time of bash shell processes:

```
# uoffcpu.d bash
dtrace: script 'uoffcpu.d' matched 6 probes
^C
[...]
```

continues

```
    off-cpu (us):
                libc.so.1`__waitid+0x7
                libc.so.1`waitpid+0x65
                bash`0x8090627
                bash`wait_for+0x1a4
                bash`execute_command_internal+0x6f1
                bash`execute_command+0x5b
                bash`reader_loop+0x1bf
                bash`main+0x7df
                bash`_start+0x7d

             value  ------------- Distribution ------------- count
            262144 |                                         0
            524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
           1048576 |                                         0

    off-cpu (us):
                libc.so.1`__read+0x7
                bash`rl_getc+0x47
                bash`rl_read_key+0xeb
                bash`readline_internal_char+0x99
                bash`0x80d945a
                bash`0x80d9481
                bash`readline+0x55
                bash`0x806e11f
                bash`0x806dff4
                bash`0x806ed06
                bash`0x806f9b4
                bash`0x806f3a4
                bash`yyparse+0x4b9
                bash`parse_command+0x80
                bash`read_command+0xd9
                bash`reader_loop+0x147
                bash`main+0x7df
                bash`_start+0x7d

             value  ------------- Distribution ------------- count
             32768 |                                         0
             65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@            5
            131072 |@@@@@@@@@@@                              2
            262144 |                                         0
```

While tracing, in another `bash` shell, the command `sleep 1` was typed and executed. The previous output shows the keystroke latency (mostly 65 ms to 131 ms) of the read commands, as well as the time spent waiting for the `sleep(1)` command to complete (in the 524 to 1048 ms range, which matches expectation: 1000 ms).

Note the user stack frame generated by the `ustack()` function contains a mix of symbol names and hex values (for example, `bash`0x806dff4`) in the output. This can happen for one of several reasons whenever `ustack()` is used. DTrace actually collects and stores the stack frames has hex values. User addresses are resolved to symbol names as a postprocessing step before the output is generated. It is possible DTrace will not be able to resolve a user address to a symbol name if any of the following is true:

- The user process being traced has exited before the processing can be done.

- The symbol table has been stripped, either from the user process binary or from the shared object libraries it has linked.
- We are executing user code out of data via jump tables.[6]

plockstat

`plockstat(1M)` is a powerful tool to examine user-level lock events, providing details on contention and hold time. It uses the DTrace plockstat provider, which is available for developing custom user-land lock analysis scripts. The plockstat provider (and the `plockstat(1M)` tool) is available on Solaris and Mac OS X and is currently being developed for FreeBSD.

Script

`plockstat(1M)` is a binary executable that dynamically produces a D script that is sent to libdtrace (instead of a static D script sent to libdtrace via `dtrace(1M)`). If it is of interest, this D script can be examined using the `-V` option:

```
solaris# plockstat -V -p 12219
plockstat: vvvv D program vvvv
plockstat$target:::rw-block
{
        self->rwblock[arg0] = timestamp;
}
plockstat$target:::mutex-block
{
        self->mtxblock[arg0] = timestamp;
}
plockstat$target:::mutex-spin
{
        self->mtxspin[arg0] = timestamp;
}
plockstat$target:::rw-blocked
/self->rwblock[arg0] && arg1 == 1 && arg2 != 0/
{
        @rw_w_block[arg0, ustack(5)] =
            sum(timestamp - self->rwblock[arg0]);
        @rw_w_block_count[arg0, ustack(5)] = count();
        self->rwblock[arg0] = 0;
        rw_w_block_found = 1;
}
[...output truncated...]
```

Example

Here the `plockstat(1M)` command traced all lock events (`-A` for both hold and contention) on the Name Service Cache Daemon (`nscd`) for 60 seconds:

---

[6] See *www.opensolaris.org/jive/thread.jspa?messageID=436419&#436419*.

```
solaris# plockstat -A -e 60 -p `pgrep nscd`
Mutex hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30  1302583 0x814c08c                    libnsl.so.1`rpc_fd_unlock+0x4d
  326    15687 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x6c
    7   709342 libumem.so.1`umem_cache_lock libumem.so.1`umem_cache_applyall+0x5e
  112    16702 0x80b67b8                    nscd`lookup_int+0x611
    3   570898 0x81a0548                    libscf.so.1`scf_handle_bind+0x231
   60    24592 0x80b20e8                    nscd`_nscd_mutex_unlock+0x8d
   50    24306 0x80b2868                    nscd`_nscd_mutex_unlock+0x8d
   30    19839 libnsl.so.1`_ti_userlock     libnsl.so.1`sig_mutex_unlock+0x1e
    7    83100 libumem.so.1`umem_update_lock libumem.so.1`umem_update_thread+0x129
[...output truncated...]

R/W reader hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30    95341 0x80c0e14                    nscd`_nscd_get+0xb8
  120    15586 nscd`nscd_nsw_state_base_lock nscd`_get_nsw_state_int+0x19c
   60    20256 0x80e0a7c                    nscd`_nscd_get+0xb8
  120     9806 nscd`addrDB_rwlock           nscd`_nscd_is_int_addr+0xd1
   30    39155 0x8145944                    nscd`_nscd_get+0xb8
[...output truncated...]

R/W writer hold

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   30    16293 nscd`addrDB_rwlock           nscd`_nscd_del_int_addr+0xeb
   30    15440 nscd`addrDB_rwlock           nscd`_nscd_add_int_addr+0x9c
    3    14279 nscd`nscd_smf_service_state_lock nscd`query_smf_state+0x17b

Mutex block

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    2   119957 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e

Mutex spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    1    37959 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e

Mutex unsuccessful spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
    2    42988 0x8089ab8                    nscd`_nscd_restart_if_cfgfile_changed+0x3e
```

While tracing, there were very few contention events and many hold events. Hold events are normal for software execution and are ideally as short as possible, while contention events can cause performance issues as threads are waiting for locks.

The output has caught a spin event for the lock at address `0x8089ab8` (no symbol name) from the code path location ``nscd`_nscd_restart_if_cfgfile_changed+0x3e``, which was for 38 us. This means a thread span on-CPU for 38 us before being able to grab the lock. On two other occasions, the thread gave up spinning after an average of 43 us (unsuccessful spin) and was blocked for 120 us (block), both also shown in the output.

kill.d

The `kill.d` script prints details of process signals as they are sent, such as the PID source and destination, signal number, and result. It's named `kill.d` after the `kill()` system call that it traces, which is used by processes to send signals.

Script

This is based on the `kill.d` script from the DTraceToolkit, which uses the syscall provider to trace the `kill()` syscall. The proc provider could also be used via the `signal-*` probes, which will match other signals other than via `kill()` (see `sigdist.d` next).

```
1   #!/usr/sbin/dtrace -s
2
3   #pragma D option quiet
4
5   dtrace:::BEGIN
6   {
7          printf("%-6s %12s %6s %-8s %s\n",
8              "FROM", "COMMAND", "SIG", "TO", "RESULT");
9   }
10
11  syscall::kill:entry
12  {
13          self->target = (int)arg0;
14          self->signal = arg1;
15  }
16
17  syscall::kill:return
18  {
19          printf("%-6d %12s %6d %-8d %d\n",
20              pid, execname, self->signal, self->target, (int)arg0);
21          self->target = 0;
22          self->signal = 0;
23  }
```

Script kill.d

Note that the target PID is cast as a signed integer on line 13; this is because the `kill()` syscall can also send signals to process groups by providing the process group ID as a negative number, instead of the PID. By casting it, it will be correctly printed as a signed integer on line 19.

Example

Here the `kill.d` script has traced the `bash` shell sending signal 9 (`SIGKILL`) to PID 12838 and sending signal 2 (`SIGINT`) to itself, which was a Ctrl-C. `kill.d` has also traced `utmpd` sending a 0 signal (the null signal) to various processes: This signal is used to check that PIDs are still valid, without signaling them to do anything (see `kill(2)`).

```
# kill.d
FROM       COMMAND    SIG TO       RESULT
12224         bash      9 12838    0
3728         utmpd      0 4174     0
3728         utmpd      0 3949     0
3728         utmpd      0 10621    0
3728         utmpd      0 12221    0
12224         bash      2 12224    0
```

sigdist.d

The `sigdist.d` script shows which processes are sending which signals to other processes, including the process names. This traces all signals: the `kill()` system call as well as kernel-based signals (for example, alarms).

Script

This script is based on `/usr/demo/dtrace/sig.d` from Oracle Solaris and uses the proc provider `signal-send` probe.

```
#!/usr/sbin/dtrace -s
[...]
#pragma D option quiet

dtrace:::BEGIN
{
      printf("Tracing... Hit Ctrl-C to end.\n");
}

proc:::signal-send
{
      @Count[execname, stringof(args[1]->pr_fname), args[2]] = count();
}

dtrace:::END
{
      printf("%16s %16s %6s %6s\n", "SENDER", "RECIPIENT", "SIG", "COUNT");
      printa("%16s %16s %6d %6@d\n", @Count);
}
```

Script sigdist.d

Example

The `sigdist.d` script has traced the bash shell sending signal 9 (`SIGKILL`) to a sleep process and also signal 2 (`SIGINT`, Ctrl-C) to itself. It's also picked up `sshd` sending `bash` the `SIGINT`, which happened via a syscall `write()` of the Ctrl-C to the ptm (`STREAMS` pseudo-tty master driver) device for `bash`, not via the `kill()` syscall.

```
# sigdist.d
Tracing... Hit Ctrl-C to end.
^C
          SENDER        RECIPIENT    SIG  COUNT
            bash             bash      2      1
            bash            sleep      9      1
            sshd             bash      2      1
            sshd           dtrace      2      1
           sched             bash     18      2
            bash             bash     20      3
           sched         sendmail     14      3
           sched         sendmail     18      3
           sched          proftpd     14      7
           sched        in.mpathd     14     10
```

threaded.d

The `threaded.d` script provides data for quantifying how well multithreaded applications are performing, in terms of parallel execution across CPUs. If an application has sufficient CPU bound work and is running on a system with multiple CPUs, then ideally the application would have multiple threads running on those CPUs to process the work in parallel.

Script

This is based on the `threaded.d` script from the DTraceToolkit.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

profile:::profile-101
/pid != 0/
{
      @sample[pid, execname] = lquantize(tid, 0, 128, 1);
}

 profile:::tick-1sec
 {
      printf("%Y,\n", walltimestamp);
      printa("\n @101hz   PID: %-8d CMD: %s\n%@d", @sample);
      printf("\n");
      trunc(@sample);
 }
```

Script threaded.d

Example

To demonstrate `threaded.d`, two programs were written (called test0 and test1) that perform work on multiple threads in parallel. One of the programs was coded with a lock “serialization” issue, where only the thread holding the lock can really make forward progress. See whether you can tell which one:

```
# threaded.d
2010 Jul  4 05:17:09,

 @101hz   PID: 12974    CMD: test0

           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@@@@@@                                28
               3 |@@                                       6
               4 |@@@@@@@@@@@                              32
               5 |@@@@@                                    14
               6 |@@@@@                                    15
               7 |@@@                                      8
               8 |@@                                       5
               9 |@@@                                      10
              10 |                                         0

 @101hz   PID: 12977    CMD: test1

           value  ------------- Distribution ------------- count
               1 |                                         0
               2 |@@@@                                     77
               3 |@@@@@@                                   97
               4 |@@@@                                     77
               5 |@@@@@                                    87
               6 |@@@@                                     76
               7 |@@@@@@                                   101
               8 |@@@@                                     76
               9 |@@@@@@                                   100
              10 |                                         0

[...]
```

`threaded.d` prints output every second, which shows a distribution plot where `value` is the thread ID and `count` is the number of samples during that second. By glancing at the output, both programs had every thread sampled on-CPU during the one second, so the issue may not be clear. The clue is in the counts: `threaded.d` is sampling at 101 Hertz (101 times per second), and the sample counts for test0 only add up to 118 (a little over one second worth of samples on one CPU), whereas test1 adds up to 691. The program with the issue is test0, which is using a fraction of the CPU cycles that test1 is able to consume in the same interval.

This was a simple way to analyze the CPU execution of a multithreaded application. A more sophisticated approach would be to trace kernel scheduling events (the sched provider) as the application threads stepped on- and off-CPU.

Case Studies

In this section, we apply the scripts and methods discussed in this chapter to observe and measure applications with DTrace.

Firefox idle

This case study examines the Mozilla Firefox Web browser version 3, running on Oracle Solaris.

The Problem

Firefox is 8.9 percent on-CPU yet has not been used for hours. What is costing 8.9 percent CPU?

```
# prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 27060 brendan   856M  668M sleep   59    0   7:30:44 8.9% firefox-bin/17
 27035 brendan   150M  136M sleep   59    0   0:20:51 0.4% opera/3
 18722 brendan   164M   38M sleep   59    0   0:57:53 0.1% java/18
  1748 brendan  6396K 4936K sleep   59    0   0:03:13 0.1% screen-4.0.2/1
 17303 brendan   305M  247M sleep   59    0  34:16:57 0.1% Xorg/1
 27754 brendan  9564K 3772K sleep   59    0   0:00:00 0.0% sshd/1
 19998 brendan    68M 7008K sleep   59    0   2:41:34 0.0% gnome-netstatus/1
 27871 root     3360K 2792K cpu0    49    0   0:00:00 0.0% prstat/1
 29805 brendan    54M   46M sleep   59    0   1:53:23 0.0% elinks/1
[...]
```

Profiling User Stacks

The `uoncpu.d` script (from the "Scripts" section) was run for ten seconds:

```
# uoncpu.d firefox-bin
dtrace: script 'uoncpu.d' matched 1 probe
^C
[...output truncated...]

  on-cpu (count @1001hz):
              libmozjs.so`js_FlushPropertyCacheForScript+0xe6
              libmozjs.so`js_DestroyScript+0xc1
              libmozjs.so`JS_EvaluateUCScriptForPrincipals+0x87
              libxul.so`__1cLnsJSContextOEvaluateString6MrknSnsAString_internal_pvpnMn
sIPrincip8
              libxul.so`__1cOnsGlobalWindowKRunTimeout6MpnJnsTimeout__v_+0x59c
              libxul.so`__1cOnsGlobalWindowNTimerCallback6FpnInsITimer_pv_v_+0x2e
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x144
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
```

continues

```
              libxul.so`__1cMnsAppStartupDRun6M_I_+0x34
              libxul.so`XRE_main+0x35e3
              firefox-bin`main+0x223
              firefox-bin`_start+0x7d
               42
```

The output was many pages long and includes C++ signatures as function names (they can be passed through `c++filt` to improve readability). The hottest stack is in libmozjs, which is SpiderMonkey—the Firefox JavaScript engine. However, the count for this hot stack is only 42, which, when the other counts from the numerous truncated pages are tallied, is likely to represent only a fraction of the CPU cycles. (`uoncpu.d` can be enhanced to print a total sample count and the end to make this ratio calculation easy to do.)

Profiling User Modules

Perhaps an easier way to find the origin of the CPU usage is to not aggregate on the entire user stack track but just the top-level user module. This won't be as accurate—a user module may be consuming CPU by calling functions from a generic library such as libc—but it is worth a try:

```
# dtrace -n 'profile-1001 /execname == "firefox-bin"/ { @[umod(arg1)] = count(); }
tick-60sec { exit(0); }'
dtrace: description 'profile-1001 ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  63284                      :tick-60sec

  libsqlite3.so                                                     1
  0xf0800000                                                        2
  libplds4.so                                                       2
  libORBit-2.so.0.0.0                                               5
  0xf1600000                                                        8
  libgthread-2.0.so.0.1400.4                                       10
  libgdk-x11-2.0.so.0.1200.3                                       14
  libplc4.so                                                       16
  libm.so.2                                                        19
  libX11.so.4                                                      50
  libnspr4.so                                                     314
  libglib-2.0.so.0.1400.4                                         527
  0x0                                                             533
  libflashplayer.so                                              1143
  libc.so.1                                                      1444
  libmozjs.so                                                    2671
  libxul.so                                                      4143
```

The hottest module was libxul, which is the core Firefox library. The next was libmozjs (JavaScript) and then libc (generic system library). It is possible that libmozjs is responsible for the CPU time in both libc and libxul, by calling functions from them. We'll investigate libmozjs (JavaScript) first; if this turns out to be a dead end, we'll return to libxul.

Function Counts and Stacks

To investigate JavaScript, the DTrace JavaScript provider can be used (see Chapter 8). For the purposes of this case study, let's assume that such a convenient provider is not available. To understand what the libmosjs library is doing, we'll first frequency count function calls:

```
# dtrace -n 'pid$target:libmozjs::entry { @[probefunc] = count(); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target:libmozjs::entry ' matched 1617 probes
^C

  CloseNativeIterators                                                1
  DestroyGCArenas                                                     1
  JS_CompareValues                                                    1
  JS_DefineElement                                                    1
  JS_FloorLog2                                                        1
  JS_GC                                                               1
[...]
  JS_free                                                         90312
  js_IsAboutToBeFinalized                                         92414
  js_GetToken                                                     99666
  JS_DHashTableOperate                                           102908
  GetChar                                                        109323
  fun_trace                                                      132924
  JS_GetPrivate                                                  197322
  js_TraceObject                                                 213983
  JS_TraceChildren                                               228323
  js_SearchScope                                                 267826
  js_TraceScopeProperty                                          505450
  JS_CallTracer                                                 1923784
```

The most frequent function called was `JS_CallTracer()`, which was called almost two million times during the ten seconds that this one-liner was tracing. To see what it does, the source code could be examined; but before we do that, we can get more information from DTrace including frequency counting the user stack trace to see who is calling this function:

```
# dtrace -n 'pid$target:libmozjs:JS_CallTracer:entry { @[ustack()] =
count(); }' -p `pgrep firefox-bin`
[...]

              libmozjs.so`JS_CallTracer
              libmozjs.so`js_TraceScopeProperty+0x54
              libmozjs.so`js_TraceObject+0xd5
              libmozjs.so`JS_TraceChildren+0x351
              libxul.so`__1cLnsXPConnectITraverse6MpvrnbInsCycleCollectionTraversalCal
lback__I_+0xc7
              libxul.so`__1cQnsCycleCollectorJMarkRoots6MrnOGCGraphBuilder__v_+0x96
              libxul.so`__1cQnsCycleCollectorPBeginCollection6M_i_+0xf1
              libxul.so`__1cbGnsCycleCollector_beginCollection6F_i_+0x26
              libxul.so`__1cZXPCCycleCollectGCCallback6FpnJJSContext_nKJSGCStatus__i_+0xd8
              libmozjs.so`js_GC+0x5ef
```

continues

```
              libmozjs.so`JS_GC+0x4e
              libxul.so`__1cLnsXPConnectHCollect6M_i_+0xaf
              libxul.so`__1cQnsCycleCollectorHCollect6MI_I_+0xee
              libxul.so`__1cYnsCycleCollector_collect6F_I_+0x28
              libxul.so`__1cLnsJSContextGNotify6MpnInsITimer__I_+0x375
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x12d
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
            40190
```

The stack trace here has been truncated (increase the `ustackframes` tunable to see all); however, enough has been seen for this and the truncated stack traces to see that they originate from `JS_GC()`—a quick look at the code confirms that this is JavaScript Garbage Collect.

Function CPU Time

Given the name of the garbage collect function, a script can be quickly written to check the CPU time spent in it (named `jsgc.d`):

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

pid$target::JS_GC:entry
{
        self->vstart = vtimestamp;
}

pid$target::JS_GC:return
/self->vstart/
{
        this->oncpu = (vtimestamp - self->vstart) / 1000000;
        printf("%Y GC: %d CPU ms\n", walltimestamp, this->oncpu);
        self->vstart = 0;
}
```

Script jsgc.d

This specifically measures the elapsed CPU time (vtimestamp) for `JS_GC()`. (Another approach would be to use the profile provider and count stack traces that included `JS_GC()`.)

Here we execute `jsgc.d`:

```
# jsgc.d -p `pgrep firefox-bin`
2010 Jul  4 01:06:57 GC: 331 CPU ms
2010 Jul  4 01:07:38 GC: 316 CPU ms
2010 Jul  4 01:08:18 GC: 315 CPU ms
^C
```

So, although GC is on-CPU for a significant time, more than 300 ms per call, it's not happening frequently enough to explain the 9 percent CPU average of Firefox. This may be a problem, but it's not the problem. (This is included here for completeness; this is the exact approach used to study this issue.)

Another frequently called function was `js_SearchScope()`. Checking its stack trace is also worth a look:

```
# dtrace -n 'pid$target:libmozjs:js_SearchScope:entry { @[ustack()] =
count(); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target:libmozjs:js_SearchScope:entry ' matched 1 probe
^C
[...output truncated...]

              libmozjs.so`js_SearchScope
              libmozjs.so`js_DefineNativeProperty+0x2f1
              libmozjs.so`call_resolve+0x1e7
              libmozjs.so`js_LookupProperty+0x3d3
              libmozjs.so`js_PutCallObject+0x164
              libmozjs.so`js_Interpret+0x9cd4
              libmozjs.so`js_Execute+0x3b4
              libmozjs.so`JS_EvaluateUCScriptForPrincipals+0x58
              libxul.so`__1cLnsJSContextOEvaluateString6MrknSnsAString_internal_pvpnMn
sIPrincipal_pkcIIp1pi_I_+0x2e8
              libxul.so`__1cOnsGlobalWindowKRunTimeout6MpnJnsTimeout__v_+0x59c
              libxul.so`__1cOnsGlobalWindowNTimerCallback6FpnInsITimer_pv_v_+0x2e
              libxul.so`__1cLnsTimerImplEFire6M_v_+0x144
              libxul.so`__1cMnsTimerEventDRun6M_I_+0x51
              libxul.so`__1cInsThreadQProcessNextEvent6Mipi_I_+0x143
              libxul.so`__1cVNS_ProcessNextEvent_P6FpnJnsIThread_i_i_+0x44
              libxul.so`__1cOnsBaseAppShellDRun6M_I_+0x3a
              libxul.so`__1cMnsAppStartupDRun6M_I_+0x34
              libxul.so`XRE_main+0x35e3
              firefox-bin`main+0x223
              firefox-bin`_start+0x7d
             9287
```

This time, the function is being called by `js_Execute()`, the entry point for JavaScript code execution (and itself was called by `JS_EvaluateUCScriptForPrincipals()`). Here we are modifying the earlier script to examine on-CPU time (now `jsexecute.d`):

```
#!/usr/sbin/dtrace -s

pid$target::js_Execute:entry
{
        self->vstart = vtimestamp;
}

pid$target::js_Execute:return
/self->vstart/
{
        this->oncpu = vtimestamp - self->vstart;
        @["js_Execute Total(ns):"] = sum(this->oncpu);
        self->vstart = 0;
}
```

Script jsexecute.d

Here we run it for ten seconds:

```
# jsexecute.d -p `pgrep firefox-bin` -n 'tick-10sec { exit(0); }'
dtrace: script 'jsexecute.d' matched 2 probes
dtrace: description 'tick-10sec ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  64907                      :tick-10sec

  js_Execute Total(ns):                                      427936779
```

This shows 428 ms of time in `js_Execute()` during those ten seconds, and so this CPU cost can explain about half of the Firefox CPU time (this is a single-CPU system; therefore, there is 10,000 ms of available CPU time every 10 seconds, so this is about 4.3 percent of CPU).

The JavaScript functions could be further examined with DTrace to find out why this JavaScript program is hot on-CPU, in other words, what exactly it is doing (the DTrace JavaScript provider would help here, or a Firefox add-on could be tried).

Fetching Context

Here we will find what is being executed: preferably the URL. Examining the earlier stack trace along with the Firefox source (which is publically available) showed the JavaScript filename is the sixth argument to the `JS_EvaluateUCScriptForPrincipals()` function. Here we are pulling this in and frequency counting:

```
# dtrace -n 'pid$target::*EvaluateUCScriptForPrincipals*:entry { @[copyinstr(arg5)] =
 count(); } tick-10sec { exit(0); }' -p `pgrep firefox-bin`
dtrace: description 'pid$target::*EvaluateUCScriptForPrincipals*:entry ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1  64907                      :tick-10sec

  http://www.example.com/js/st188.js                                     7056
```

The name of the URL has been modified in this output (to avoid embarrassing anyone); it pointed to a site that I didn't think I was using, yet their script was getting executed more than 700 times per second anyway, which is consuming (wasting!) at least 4 percent of the CPU on this system.

The Fix

An add-on was already available that could help at this point: SaveMemory, which allows browser tabs to be paused. The DTrace one-liner was modified to print continual one-second summaries, while all tabs were paused as an experiment:

```
# dtrace -n 'pid$target::*EvaluateUCScriptForPrincipals*:entry { @[copyinstr(arg5)] =
 count(); } tick-1sec { printa(@); trunc(@); }' -p `pgrep firefox-bin`
[...]
  1  63140                        :tick-1sec
  http://www.example.com/js/st188.js                                        697

  1  63140                        :tick-1sec
  http://www.example.com/js/st188.js                                        703

  1  63140                        :tick-1sec
file:///export/home/brendan/.mozilla/firefox/3c8k4kh0.default/extensions/%7Be4a8a97b-f
2ed-450b-b12d-ee082ba24781%7D/components/greasemonkey.js                  1
  http://www.example.com/js/st188.js                                        126


  1  63140                        :tick-1sec

  1  63140                        :tick-1sec
```

The execution count for the JavaScript program begins at around 700 executions per second and then vanishes when pausing all tabs. (The output has also caught the execution of `greasemonkey.js`, executed as the add-on was used.)

`prstat(1M)` shows the CPU problem is no longer there (shown after waiting a few minutes for the %CPU decayed average to settle):

```
# prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
 27035 brendan   150M  136M sleep   49    0   0:27:15 0.2% opera/4
 27060 brendan   407M  304M sleep   59    0   7:35:12 0.1% firefox-bin/17
 28424 root     3392K 2824K cpu1    49    0   0:00:00 0.0% prstat/1
[...]
```

Next, the browser tabs were unpaused one by one to identify the culprit, while still running the DTrace one-liner to track JavaScript execution by file. This showed that there were seven tabs open on the same Web site that was running the JavaScript program—each of them executing it about 100 times per second. The Web site is a popular blogging platform, and the JavaScript was being executed by what appears to be an inert icon that links to a different Web site (but as we found out—it is not inert).[7] The exact operation of that JavaScript program can now be investigated using the DTrace JavaScript provider or a Firefox add-on debugger.

Conclusion

A large component of this issue turned out to be a rogue JavaScript program, an issue that could also have been identified with Firefox add-ons. The advantage of

---

[7] An e-mail was sent to the administrators of the blogging platform to let them know.

using DTrace is that if there is an issue, the root cause can be identified—no matter where it lives in the software stack. As an example of this,[8] about a year ago a performance issue was identified in Firefox and investigated in the same way—and found to be a bug in a kernel frame buffer driver (video driver); this would be extremely difficult to have identified from the application layer alone.

Xvnc

Xvnc is a Virtual Network Computing (VNC) server that allows remote access to X server–based desktops. This case study represents examining an Xvnc process that is CPU-bound and demonstrates using the syscall and profile providers.

When performing a routine check of running processes on a Solaris system by using `prstat(1)`, it was discovered that an Xvnc process was the top CPU consumer. Looking just at that process yields the following:

```
solaris# prstat -c -Lmp 5459
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  5459 nobody    86  14 0.0 0.0 0.0 0.0 0.0 0.0   0  36 .2M 166 Xvnc/1
```

We can see the Xvnc process is spending most of its time executing in user mode (USR, 86 percent) and some of its time in the kernel (SYS, 14 percent). Also worth noting is it is executing about 200,000 system calls per second (SCL value of `.2M`).

syscall Provider

Let's start by checking what those system calls are. This one-liner uses the syscall provider to frequency count system calls for this process and prints a summary every second:

```
solaris# dtrace -qn 'syscall:::entry /pid == 5459/ { @[probefunc] =
count(); } tick-1sec { printa(@); trunc(@); }'

  read                                                              4
  lwp_sigmask                                                      34
  setcontext                                                       34
  setitimer                                                        68
  accept                                                        48439
  gtime                                                         48439
  pollsys                                                       48440
  write                                                         97382
```

continues

---

8. I'd include this as a case study here, if I had thought to save the DTrace output at the time.

```
  read                                                      4
  lwp_sigmask                                              33
  setcontext                                               33
  setitimer                                                66
  gtime                                                 48307
  pollsys                                               48307
  accept                                                48308
  write                                                 97117
```

Because the rate of system calls was relatively high, as reported by `prstat(1M)`, we opted to display per-second rates with DTrace. The output shows more than 97,000 `write()` system calls per second and just more than 48,000 `accept()`, `poll()`, and `gtime()` calls.

Let's take a look at the target of all the writes and the requested number of bytes to write:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459/ { @[fds[arg0].fi_pathname,
arg2] = count(); }'
^C

  /var/adm/X2msgs                                              26               8
  /devices/pseudo/mm@0:null                                  8192            3752
  /var/adm/X2msgs                                              82          361594
  /var/adm/X2msgs                                              35          361595
```

The vast majority of the writes are to a file, `/var/adm/X2msgs`. The number of bytes to write was 82 bytes and 35 bytes for the most part (more than 361,000 times each). Checking that file yields the following:

```
solaris# ls -l /var/adm/X2msgs
-rw-r--r--   1 root     nobody   2147483647 Aug 13 15:05 /var/adm/X2msgs
solaris# tail /var/adm/X2msgs
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
 XserverDesktop: XserverDesktop::wakeupHandler: unable to accept new
              connection: Invalid argument (22)
```

Looking at the file Xvnc is writing to, we can see it is getting very large (more than 2GB), and the messages themselves appear to be error messages. We will explore that more closely in just a minute.

Given the rate of 97,000 writes per second, we can already extrapolate that each write is taking much less than 1 ms (1/97000 = 0.000010), so we know the data is probably being written to main memory (since the file resides on a file system and the writes are not synchronous, they are being satisfied by the in-memory file system cache). We can of course time these writes with DTrace:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459/
{ @[fds[arg0].fi_fs] = count(); }'
^C
  specfs                                                          2766
  zfs                                                           533090

solaris# cat -n w.d
1   #!/usr/sbin/dtrace -qs
2
3   syscall::write:entry
4   /pid == 5459 && fds[arg0].fi_fs == "zfs"/
5   {
6           self->st = timestamp;
7   }
8   syscall::write:return
9   /self->st/
10  {
11          @ = quantize(timestamp - self->st);
12          self->st = 0;
13  }

solaris# ./w.d
^C

           value  ------------- Distribution ------------- count
             256 |                                         0
             512 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1477349
            1024 |                                         2312
            2048 |                                         3100
            4096 |                                         250
            8192 |                                         233
           16384 |                                         145
           32768 |                                         90
           65536 |                                         0
```

Before measuring the write time, we wanted to be sure we knew the target file system type of the file being written, which was ZFS. We used that in the predicate in the `w.d` script to measure write system calls for this process (along with the process PID test). The output of `w.d` is a quantize aggregation that displays wall clock time for all the write calls executed to a ZFS file system from that process during the sampling period. We see that most of the writes fall in the 512-nanosecond to 1024-nanosecond range, so these are most certainly writes to memory.

We can determine the user code path leading up to the writes by aggregating on the user stack when the write system call is called:

```
solaris# dtrace -qn 'syscall::write:entry /pid == 5459 && fds[arg0].fi_fs ==
"zfs"/ { @[ustack()] = count(); }'
^C
[...]
```

```
              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1a5
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           430879

              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1eb
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           430879
```

We see two very similar stack frames, indicating a log event is causing the Xvnc process to write to its log file.

We can even use DTrace to observe what is being written to the file, by examining the contents of the buffer pointer from the `write(2)` system call. It is passed to the `copyinstr()` function, both to copy the data from user-land into the kernel address space and to treat it as a string:

```
solaris# dtrace -n 'syscall::write:entry /pid == 5459/ { @[copyinstr(arg1)] =
count(); }'
dtrace: description 'syscall::write:entry ' matched 1 probe
^C

Sun Aug 22 00:09:05 2010
ent (22)
keupHandler: unable to accept new
             st!
Ltd.
See http://www.realvnc.com for information on VNC.
                1

Sun Aug 22 00:09:06 2010
ent (22)
keupHandler: unable to accept new
             st!
                2
[...]
upHandler: unable to accept new connection: Invalid argument (22)XserverDesktop::wakeu
pHandler: unable to accept new connection: Invalid argument (22)XserverDesktop::wakeup
Handler: unable to accept new connection: Invalid argument (22)XserverDesktop::wake
```

```
                59
valid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: I
nvalid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: In
valid argument (22)XserverDesktop::wakeupHandler: unable to accept new connection: In
                59
```

This shows the text being written to the log file, which largely contains errors describing invalid arguments used for new connections. Remember that our initial one-liner discovered more than 48,000 `accept()` system calls per-second—it would appear that these are failing because of invalid arguments, which is being written as an error message to the `/var/adm/X2msgs` log.

DTrace can confirm that the `accept()` system calls are failing in this way, by examining the error number (errno) on syscall return:

```
solaris# dtrace -n 'syscall::accept:return /pid == 5459/ { @[errno] = count(); }'
dtrace: description 'syscall::accept:return ' matched 1 probe
^C

       22           566135

solaris# grep 22 /usr/include/sys/errno.h
#define     EINVAL      22    /* Invalid argument                 */
```

All the `accept()` system calls are returning with errno 22, `EINVAL` (Invalid argument). The reason for this can be investigated by examining the arguments to the `accept()` system call.

```
solaris# dtrace -n 'syscall::accept:entry /execname == "Xvnc"/ { @[arg0, arg1,
arg2] = count(); }'
dtrace: description 'syscall::accept:entry ' matched 1 probe
^C

                3                0                0           150059
```

We see the first argument to accept is 3, which is the file descriptor for the socket. The second two arguments are both `NULL`, which *may* be the cause of the `EINVAL` error return from accept. It is possible it is valid to call `accept` with the second and third arguments as `NULL` values,[9] in which case the Xvnc code is not handling the error return properly. In either case, the next step would be to look at the Xvnc source code and find the problem. The code is burning a lot of CPU with calls to `accept(2)` that are returning an error and each time generating a log file write.

---

[9] Stevens (1998) indicates that it is.

While still using the syscall provider, the user code path for another of the other hot system calls can be examined:

```
solaris# dtrace -n 'syscall::gtime:entry /pid == 5459/ { @[ustack()] = count(); }'
dtrace: description 'syscall::gtime:entry ' matched 1 probe
^C

              libc.so.1`__time+0x7
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0xce
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
           370156
```

This shows that calls to `gtime(2)` are part of the log file writes in the application, based on the user function names we see in the stack frames.

profile Provider

To further understand the performance of this process, we will sample the on-CPU code at a certain frequency, using the profile provider.

```
solaris# dtrace -n 'profile-997hz /arg1 && pid == 5459/ { @[ufunc(arg1)] = count(); }'
dtrace: description 'profile-997hz ' matched 1 probe
^C
[...]
  libc.so.1`memcpy                                                  905
  Xvnc`_ZN14XserverDesktop12blockHandlerEP6fd_set                   957
  libgcc_s.so.1`uw_update_context_1                                1155
  Xvnc`_ZN3rdr15SystemExceptionC2EPKci                             1205
  libgcc_s.so.1`execute_cfa_program                                1278
  libc.so.1`strncat                                                1418
  libc.so.1`pselect                                                1686
  libstdc++.so.6.0.3`_Z12read_uleb128PKhPj                         1700
  libstdc++.so.6.0.3`_Z28read_encoded_value_with_basehjPKhPj       2198
  libstdc++.so.6.0.3`__gxx_personality_v0                          2445
  libc.so.1`_ndoprnt                                               3918
```

This one-liner shows which user functions were on-CPU most frequently. It tests for user mode (`arg1`) and the process of interest and uses the `ufunc()` function to convert the user-mode on-CPU program counter (`arg1`) into the user function name. The most frequent is a libc function, `_ndoprnt()`, followed by several functions from the standard C++ library.

For a detailed look of the user-land code path that is responsible for consuming CPU cycles, aggregate on the user stack:

```
solaris# dtrace -n 'profile-997hz /arg1 && pid == 5459/ { @[ustack()] =
count(); } tick-10sec { trunc(@, 20); exit(0); }'
^c
[...]
              libstdc++.so.6.0.3`__gxx_personality_v0+0x29f
              libgcc_s.so.1`_Unwind_RaiseException+0x88
              libstdc++.so.6.0.3`__cxa_throw+0x64
              Xvnc`_ZN7network11TcpListener6acceptEv+0xb3
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              125

              libc.so.1`memset+0x10c
              libgcc_s.so.1`_Unwind_RaiseException+0xb7
              libstdc++.so.6.0.3`__cxa_throw+0x64
              Xvnc`_ZN7network11TcpListener6acceptEv+0xb3
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              213
```

Note that only the two most frequent stack frames are shown here. We see the event loop in the Xvnc code and visually decoding the mangled function names; we can see a function with `network` `TCPListener` `accept` in the function name. This makes sense for an application like Xvnc, which would be listening on a network socket for incoming requests and data. And we know that there's an issue with the issued `accept(2)` calls inducing a lot of looping around with the error returns.

We can also take a look at the kernel component of the CPU cycles consumed by this process, again using the profile provider and aggregating on kernel stacks:

```
solaris# dtrace -n 'profile-997hz /pid == 5459 && arg0/ { @[stack()] = count(); }'
^c
[...]
              unix`mutex_enter+0x10
              genunix`pcache_poll+0x1a5
              genunix`poll_common+0x27f
              genunix`pollsys+0xbe
              unix`sys_syscall32+0x101
               31

              unix`tsc_read+0x3
              genunix`gethrtime+0xa
              unix`pc_gethrestime+0x31
              genunix`gethrestime+0xa
              unix`gethrestime_sec+0x11
              genunix`gtime+0x9
```

```
              unix`sys_syscall32+0x101
               41

              unix`tsc_read+0x3
              genunix`gethrtime_unscaled+0xa
              genunix`syscall_mstate+0x4f
              unix`sys_syscall32+0x11d
              111

              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0
              229
```

The kernel stack is consistent with previously observed data. We see system call processing (remember, this process is doing 200,000 system calls per second), we see the gtime system call stack in the kernel, as well as the poll system call kernel stack. We could measure this to get more detail, but the process profile was only 14 percent kernel time, and given the rate and type of system calls being executed by this process, there is minimal additional value in terms of understanding the CPU consumption by this process in measuring kernel functions.

For a more connected view, we can trace code flow from user mode through the kernel by aggregating on both stacks:

```
solaris# dtrace -n 'profile-997hz /pid == 5459/ { @[stack(), ustack()] =
count(); } tick-10sec { trunc(@, 2); exit(0); }'
dtrace: description 'profile-997hz ' matched 2 probes
CPU     ID                    FUNCTION:NAME
  1 122538                      :tick-10sec


              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0

              libc.so.1`_write+0x7
              libc.so.1`_ndoprnt+0x2816
              libc.so.1`fprintf+0x99
              Xvnc`_ZN3rfb11Logger_File5writeEiPKcS2_+0x1eb
              Xvnc`_ZN3rfb6Logger5writeEiPKcS2_Pc+0x36
              Xvnc`_ZN3rfb9LogWriter5errorEPKcz+0x2d
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x28b
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              211

              unix`lock_try+0x8
              genunix`post_syscall+0x3b6
              genunix`syscall_exit+0x59
              unix`sys_syscall32+0x1a0
```

continues

```
              libc.so.1`_so_accept+0x7
              Xvnc`_ZN7network11TcpListener6acceptEv+0x18
              Xvnc`_ZN14XserverDesktop13wakeupHandlerEP6fd_seti+0x13d
              Xvnc`vncWakeupHandler+0x3d
              Xvnc`WakeupHandler+0x36
              Xvnc`WaitForSomething+0x28d
              Xvnc`Dispatch+0x76
              Xvnc`main+0x3e5
              Xvnc`_start+0x80
              493
```

Here we see the event loop calling into the `accept(3S)` interface in libc and entering the system call entry point in the kernel. The second set of stack frames shows the log write path. One of the stacks has also caught `_ndoprnt`, which we know from earlier to be the hottest on-CPU function, calling `write()` as part of Xvnc logging.

Conclusions

The initial analysis with standard operating system tools showed that the single-threaded Xvnc process was CPU bound, spending most of its CPU cycles in user-mode and performing more than 200,000 system calls per second. DTrace was used to discover that the application was continually encountering new connection failures because of invalid arguments (`accept(2)`) and was writing this message to a log file, thousands of times per second.

Summary

With DTrace, applications can be studied like never before: following the flow of code from the application source, through libraries, through system calls, and through the kernel. This chapter completed the topics for application analysis; see other chapters in this book for related topics, including the analysis of programming languages, disk, file system, and network I/O.

Index