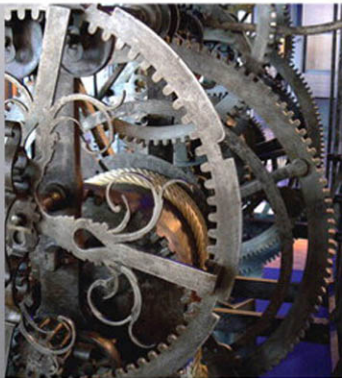


DTrace

DYNAMIC TRACING IN ORACLE® SOLARIS,
MAC OS X, AND FREEBSD



Brendan Gregg • Jim Mauro
Foreword by Bryan Cantrill

DTrace

This page intentionally left blank



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

Foreword	xxi
Preface	xxv
Acknowledgments	xxxii
About the Authors	xxxv

Part I Introduction

Chapter 1	Introduction to DTrace	1
	What Is DTrace?	1
	Why Do You Need It?	1
	Capabilities	2
	Dynamic and Static Probes	4
	DTrace Features	4
	A First Look	6
	Overview	8
	Consumers	9
	Probes	10
	Providers	11
	Predicates	13

	Actions	13
	Aggregations	13
	D Language	14
	Architecture	16
	Summary	17
Chapter 2	D Language	19
	D Language Components	20
	Usage	20
	Program Structure	21
	Probe Format	21
	Predicates	22
	Actions	23
	Probes	23
	Wildcards	23
	BEGIN and END	24
	profile and tick	24
	syscall Entry and Return	25
	Variables	26
	Types	26
	Operators	27
	Scalar	28
	Associative Arrays	29
	Structs and Pointers	29
	Thread Local	30
	Clause Local	30
	Built-in	31
	Macro	32
	External	33
	Aggregations	33
	Types	34
	quantize()	34
	lquantize()	35

trunc() and clear()	36
normalize()	36
printa()	36
Actions	37
trace()	37
printf()	38
tracemem()	39
copyin()	39
stringof() and copyinstr()	39
strlen() and strjoin()	40
stack(), ustack(), and jstack()	40
sizeof()	41
exit()	41
Speculations	41
Translators	42
Others	42
Options	43
Example Programs	44
Hello World	44
Tracing Who Opened What	44
Tracing fork() and exec()	45
Counting System Calls by a Named Process	45
Showing Read Byte Distributions by Process	45
Profiling Process Names	46
Timing a System Call	47
Snoop Process Execution	48
Summary	49
Part II	Using DTrace
Chapter 3	System View
	51
Start at the Beginning	52
System Methodology	53
System Tools	54

Observing CPUs	56
CPU Strategy	56
CPUs and Interrupts	85
CPU Events	88
CPU Summary	94
Observing Memory	95
Memory Strategy	95
Memory Checklist	96
Memory Providers	96
Memory One-Liners	97
Memory Analysis	98
User Process Memory Activity	101
Kernel Memory	118
Memory Summary	124
Observing Disk and Network I/O	125
I/O Strategy	125
I/O Checklist	125
I/O Providers	126
I/O One-Liners	127
I/O Analysis	128
Disk I/O	134
Network I/O	141
Summary	148
Chapter 4 Disk I/O	151
Capabilities	152
Disk I/O Strategy	154
Checklist	155
Providers	156
io Provider	157
fbt Provider	163
One-Liners	165
One-Liner Examples	166

Scripts	172
io Provider Scripts	173
SCSI Scripts	211
SATA Scripts	236
IDE Scripts	250
SAS Scripts	259
Case Studies	269
Shouting in the Data Center: A Personal Case Study (Brendan)	269
DTracing an Unfamiliar I/O Driver (SATA)	273
Conclusion	290
Summary	290
Chapter 5 File Systems	291
Capabilities	292
Logical vs. Physical I/O	295
Strategy	295
Checklist	296
Providers	297
fsinfo Provider	298
io Provider	300
One-Liners	300
One-Liners: syscall Provider Examples	304
One-Liners: vminfo Provider Examples	308
One-Liners: fsinfo Provider Examples	308
One-Liners: sdt Provider Examples	312
Scripts	313
Syscall Provider	315
fsinfo Scripts	327
VFS Scripts	335
UFS Scripts	351
ZFS Scripts	357
HFS+ Scripts	370

PCFS Scripts	375
HSFS Scripts	376
UDFS Scripts	378
NFS Client Scripts	379
TMPFS Scripts	385
Case Study	387
ZFS 8KB Mirror Reads	387
Conclusion	397
Summary	397
Chapter 6 Network Lower-Level Protocols	399
Capabilities	400
Strategy	402
Checklist	403
Providers	404
mib Provider	405
ip Provider	408
Network Providers	411
fbt Provider	415
One-Liners	422
Scripts	445
Socket Scripts	447
IP Scripts	469
TCP Scripts	481
UDP Scripts	517
ICMP Scripts	521
XDR Scripts	529
Ethernet Scripts	533
Common Mistakes	548
Receive Context	548
Send Context	550
Packet Size	553
Stack Reuse	554
Summary	555

Chapter 7	Application-Level Protocols	557
	Capabilities	558
	Strategy	558
	Checklist	559
	Providers	560
	fbt Provider	561
	pid Provider	562
	One-Liners	563
	Scripts	574
	NFSv3 Scripts	576
	NFSv4 Scripts	592
	CIFS Scripts	599
	HTTP Scripts	609
	DNS Scripts	621
	FTP Scripts	625
	iSCSI Scripts	633
	Fibre Channel Scripts	646
	SSH Scripts	649
	NIS Scripts	663
	LDAP Scripts	664
	Multiscripts	666
	Summary	668
Chapter 8	Languages	669
	Capabilities	671
	Strategy	672
	Checklist	674
	Providers	675
	Languages	676
	Assembly	677
	C	679
	User-Land C	680
	Kernel C	681
	Probes and Arguments	681

Struct Types	682
Includes and the Preprocessor	683
C One-Liners	684
C One-Liners Selected Examples	687
See Also	688
C Scripts	689
C++	689
Function Names	690
Object Arguments	690
Java	691
Example Java Code	693
Java One-Liners	693
Java One-Liners Selected Examples	694
Java Scripts	696
See Also	705
JavaScript	705
Example JavaScript Code	707
JavaScript One-Liners	708
JavaScript One-Liners Selected Examples	709
JavaScript Scripts	712
See Also	718
Perl	719
Example Perl Code	720
Perl One-Liners	720
Perl One-Liners Selected Examples	721
Perl Scripts	722
PHP	731
Example PHP Code	733
PHP One-Liners	734
PHP One-Liners Selected Examples	735
PHP Scripts	736
Python	740
Example Python Code	741

Python One-Liners	741
Python One-Liners Selected Examples	742
Python Scripts	744
Ruby	751
Example Ruby Code	752
Ruby One-Liners	753
Ruby One-Liners Selected Examples	753
Ruby Scripts	755
See Also	762
Shell	764
Example Shell Code	765
Shell One-Liners	765
Shell One-Liners Selected Examples	766
Shell Scripts	768
See Also	774
Tcl	774
Example Tcl Code	776
Tcl One-Liners	776
Tcl One-Liners Selected Examples	777
Tcl Scripts	778
Summary	782
Chapter 9 Applications	783
Capabilities	784
Strategy	784
Checklist	786
Providers	787
pid Provider	788
cpc Provider	791
See Also	793
One-Liners	793
One-Liner Selected Examples	798

Scripts	804
procsnoop.d	804
procsystime	806
uoncpu.d	808
uoffcpu.d	809
plockstat	811
kill.d	813
sigdist.d	814
threaded.d	815
Case Studies	817
Firefox idle	817
Xvnc	824
Summary	832
Chapter 10 Databases	833
Capabilities	834
Strategy	835
Providers	836
MySQL	837
One-Liners	838
One-Liner Selected Examples	840
Scripts	841
See Also	850
PostgreSQL	851
One-Liners	853
One-Liner Selected Examples	854
Scripts	854
See Also	858
Oracle	858
Examples	858
Summary	865

Part III Additional User Topics

Chapter 11	Security	867
	Privileges, Detection, and Debugging	867
	DTrace Privileges	868
	DTrace-Based Attacks	869
	Sniffing	869
	Security Audit Logs	870
	HIDS	871
	Policy Enforcement	871
	Privilege Debugging	872
	Reverse Engineering	874
	Scripts	875
	sshkeysnoop.d	875
	shellsnoop	878
	keylatency.d	882
	cuckoo.d	884
	watchexec.d	886
	nosetuid.d	888
	nosnoopforyou.d	890
	networkwho.d	891
	Summary	892
Chapter 12	Kernel	893
	Capabilities	894
	Strategy	896
	Checklist	897
	Providers	897
	fbt Provider	898
	Kernel Tracing	903
	Kernel Memory Usage	908
	Anonymous Tracing	917
	One-Liners	918
	One-Liner Selected Examples	925

Scripts	932
intrstat	932
lockstat	934
koncpu.d	937
koffcpu.d	938
taskq.d	939
priclass.d	941
cswstat.d	943
putnexts.d	944
Summary	945
Chapter 13 Tools	947
The DTraceToolkit	948
Locations	948
Versions	949
Installation	949
Scripts	949
Script Example: cpuwalk.d	957
Chime	962
Locations	962
Examples	963
DTrace GUI Plug-in for NetBeans and Sun Studio	966
Location	966
Examples	966
DLight, Oracle Solaris Studio 12.2	966
Locations	969
Examples	969
Mac OS X Instruments	971
Locations	972
Examples	972
Analytics	973
The Problem	973
Solving the Problem	974

Toward a Solution	975
Appliance Analytics	976
Summary	985
Chapter 14 Tips and Tricks	987
Tip 1: Known Workloads	987
Tip 2: Write Target Software	989
Tip 3: Use grep to Search for Probes	991
Tip 4: Frequency Count	991
Tip 5: Time Stamp Column, Postsort	992
Tip 6: Use Perl to Postprocess	993
Tip 7: Learn Syscalls	994
Tip 8: timestamp vs. vtimestamp	995
Tip 9: profile:::profile-997 and Profiling	996
Tip 10: Variable Scope and Use	997
Thread-Local Variables	997
Clause-Local Variables	998
Global and Aggregation Variables	999
Tip 11: strlen() and strcmp()	999
Tip 12: Check Assumptions	1000
Tip 13: Keep It Simple	1001
Tip 14: Consider Performance Impact	1001
Tip 15: drops and dynvardrops	1003
Tip 16: Tail-Call Optimization	1003
Further Reading	1003
Appendix A DTrace Tunable Variables	1005
Appendix B D Language Reference	1011
Appendix C Provider Arguments Reference	1025
Providers	1025
Arguments	1038
bufinfo_t	1038

devinfo_t	1038
fileinfo_t	1038
cpuinfo_t	1039
lwpsinfo_t	1039
psinfo_t	1039
conninfo_t	1040
pktinfo_t	1040
csinfo_t	1040
ipinfo_t	1040
ifinfo_t	1041
ipv4info_t	1041
ipv6info_t	1041
tcpinfo_t	1042
tcpsinfo_t	1042
tcplsinfo_t	1043
Appendix D DTrace on FreeBSD	1045
Enabling DTrace on FreeBSD 7.1 and 8.0	1045
DTrace for FreeBSD: John Birrell	1047
Appendix E USDT Example	1051
USDT Bourne Shell Provider	1052
Compared to SDT	1052
Defining the Provider	1052
Adding a USDT Probe to Source	1053
Stability	1055
Case Study: Implementing a Bourne Shell Provider	1057
Where to Place the Probes	1059
Appendix F DTrace Error Messages	1063
Privileges	1063
Message	1063
Meaning	1063
Suggestions	1064

Drops	1064
Message	1064
Meaning	1064
Suggestions	1064
Aggregation Drops	1065
Message	1065
Meaning	1065
Suggestions	1065
Dynamic Variable Drops	1066
Message	1066
Meaning	1066
Suggestions	1066
Invalid Address	1066
Message	1066
Meaning	1066
Suggestions	1067
Maximum Program Size	1067
Message	1067
Meaning	1067
Suggestions	1067
Not Enough Space	1068
Message	1068
Meaning	1068
Suggestions	1068
Appendix G DTrace Cheat Sheet	1069
Synopsis	1069
Finding Probes	1069
Finding Probe Arguments	1070
Probes	1070
Vars	1070
Actions	1071
Switches	1071

Pragmas	1071
One-Liners	1072
Bibliography	1073
Suggested Reading	1073
Vendor Manuals	1075
FreeBSD	1075
Mac OS X	1075
Solaris	1076
Glossary	1077
Index	1089



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 arcanelly 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 DTrace-Toolkit—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 DTraceToolkit² 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 Burrige, 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-Ismael 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 part 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.

This page intentionally left blank

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.

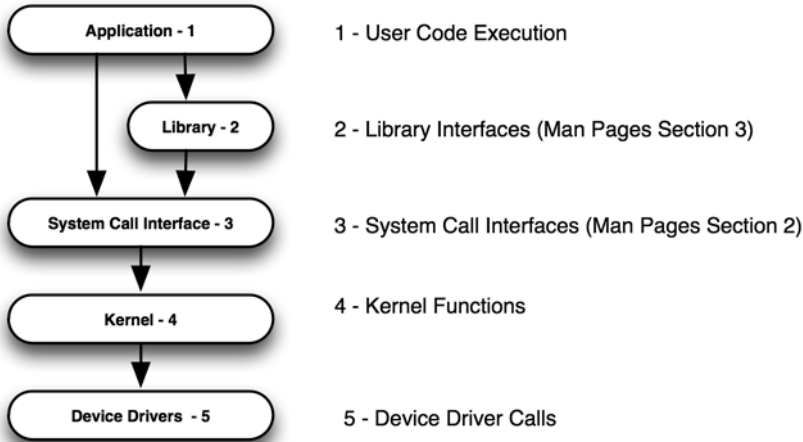


Figure 9-1 Software stack

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	<p>An application is hot on-CPU, showing high %CPU in <code>top(1)</code> or <code>prstat(1M)</code>. DTrace can identify the reason by sampling user stack traces with the profile provider and by tracing application functions with <code>vtime-stamps</code>. Reasons for high on-CPU time may include the following:</p> <ul style="list-style-type: none"> • Compression • Encryption • Dataset iteration (code path loops) • Spin lock contention • Memory I/O <p>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 <code>cpc</code> provider, if available.</p>
off-CPU time	<p>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 <code>sched</code> 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:</p> <ul style="list-style-type: none"> • Waiting on I/O during an application transaction • Waiting for work to do <p>Applications may spend most of their time waiting for work to do, which is not typically a problem.</p>

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 <code>malloc()</code>) 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 <code>vmstat(1M)</code> statistics.
sysinfo	Kernel statistics probes, based on <code>mpstat(1M)</code> 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://freebsd.foundation.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: lib-curses, 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
'__lcFStateM_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_ocr
            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_ocr 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 FreeBSD⁵):

```
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
ptime                 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
nsd                     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_fm1_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`syntab_getsym          1
dtrace          libc.so.1`memset                    1
mysqld          mysql`srv_lock_timeout_and_monitor_thread 1
mysqld          mysql`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_syntab         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

```

1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4  #pragma D option switchrate=10hz
5
6  dtrace::BEGIN
7  {
8      printf("%-8s %5s %6s %6s %s\n", "TIME(ms)", "UID", "PID", "PPID",
9          "COMMAND");
10     start = timestamp;
11 }
12
13 proc::exec-success
14 {
15     printf("%-8d %5d %6d %6d %s\n", (timestamp - start) / 1000000,
16         uid, pid, ppid, curpsinfo->pr_psargs);
17 }

```

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.

```

1  syscall::entry
2  /self->ok/
3  {
4      @Counts[probefunc] = count();
5      self->start = timestamp;
6      self->vstart = vtimestamp;
7  }
8
9  syscall::return
10 /self->start/
11 {
12     this->elapsed = timestamp - self->start;
13     this->oncpu = vtimestamp - self->vstart;

```

```

14         @Elapsed[probefunc] = sum(this->elapsed);
15         @CPU[probefunc] = sum(this->cpu);
16         self->start = 0;
17         self->vstart = 0;
18     }

```

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

```

1      #!/usr/sbin/dtrace -s
2
3      sched::off-cpu
4      /execname == $$/
5      {
6          self->start = timestamp;
7      }
8
9      sched::on-cpu
10     /self->start/
11     {
12         this->delta = (timestamp - self->start) / 1000;
13         @"off-cpu (us):", ustack()] = quantize(this->delta);
14         self->start = 0;
15     }

```

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 as 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񪣃.

```

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.

```
1      #!/usr/sbin/dtrace -s
[...]
45     #pragma D option quiet
46
47     dtrace::BEGIN
48     {
49         printf("Tracing... Hit Ctrl-C to end.\n");
50     }
51
52     proc::signal-send
53     {
54         @Count[execname, stringof(args[1]->pr_fname), args[2]] = count();
55     }
56
57     dtrace::END
58     {
59         printf("%16s %16s %6s %6s\n", "SENDER", "RECIPIENT", "SIG", "COUNT");
60         printa("%16s %16s %6d %6d\n", @Count);
61     }
```

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.

```
1      #!/usr/sbin/dtrace -s
2
3      #pragma D option quiet
4
5      profile:::profile-101
6      /pid != 0/
7      {
8          @sample[pid, execname] = lquantize(tid, 0, 128, 1);
9      }
10
11     profile:::tick-1sec
12     {
13         printf("%Y,\n", walltimestamp);
14         printa("\n @101hz   PID: %-8d CMD: %s\n@d", @sample);
15         printf("\n");
16         trunc(@sample);
17     }
```

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`__1cLnsJSCContextOEvaluateString6MrknSnsAString_internal_pvpmMn
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 libmozjs 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`__1cLnsXPConnectITraverse6MpvrbnInsCycleCollectionTraversalCal
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`):

```

1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4
5  pid$target::JS_GC:entry
6  {
7      self->vstart = vtimestamp;
8  }
9
10 pid$target::JS_GC:return
11 /self->vstart/
12 {
13     this->oncpu = (vtimestamp - self->vstart) / 1000000;
14     printf("%Y GC: %d CPU ms\n", walltimestamp, this->oncpu);
15     self->vstart = 0;
16 }

```

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_Interpreter+0x9cd4
libmozjs.so`js_Execute+0x3b4
libmozjs.so`JS_EvaluateUCScriptForPrincipals+0x58
libxul.so`__1cLnsJSContextOEvaluateString6MrknSnsAString_internal_pvpmMn
sIPrincipal_pkcIIPipi_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`):

```
1 #!/usr/sbin/dtrace -s
2
3 pid$target::js_Execute:entry
4 {
5     self->vstart = vtimestamp;
6 }
7
8 pid$target::js_Execute:return
9 /self->vstart/
10 {
11     this->oncpu = vtimestamp - self->vstart;
12     @["js_Execute Total (ns):"] = sum(this->oncpu);
13     self->vstart = 0;
14 }
```

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 cpul  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::wakeu
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`_ZN3rdrl5SystemExceptionC2EPKci            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_gethrtime+0x31
genunix`gethrtime+0xa
unix`gethrtime_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

Symbols

!, 1022–1023
!=, 28
", 880, 1021
%, 27
%=, 1023
&, 28, 1022–1023
&&, 22, 28, 1022–1023
&=, 1023
(), 1023–1024
*, 27
*=, 1023
+, 27, 1023
++, 1022
+=, 1023
=, 1023
==, 28
??. 160, 203
@, 14, 33
[], 1023–1024
^, 1022–1023
^=, 1023–1024
^^, 1022–1023
|=, 1023–1024
\$1..\$N, 32
\$\$1..\$\$N, 32
| (or), 28, 491, 1022–1024
| | (OR), 28, 458
~ (tilde), 1022–1023

^^ (XOR), 28
^ (xor), 28
:, 23, 200, 545
?., 1024
, (comma operator), 1024
- =, 1023
/=, 1023
/ (division), 27
- (subtraction), 27
--, 1022
` (backquote) character, 33, 64, 78, 231
%@ format code, 36
* (asterisk) pattern-matching character, 69–70, 133
' single quote, 24, 194, 201, 231, 880, 1021
>, 28
>=, 28
>>, 28
<, 28
<=, 28
<<, 28

A

-a, 917, 1007
-A, 917
@a, 36, 527
accept(), 427, 445, 453, 468, 828
accept-established, 482

- Access control list (ACL), 925
- Actions, 13, 23
 - copyin(), 39, 1014
 - copyinstr(), 39–40, 1014
 - exit(), 41, 1014
 - jstack(), 40–41, 1017
 - printf(), 38, 1017
 - sizeof(), 41
 - speculations, 41–42
 - stack(), 40–41, 1017, 1071
 - stringof(), 39–40
 - strjoin(), 40, 1016
 - strlen(), 40, 687–688, 1000, 1016
 - trace(), 37, 684–685
 - tracemem(), 39, 1017
 - translators, 42
 - ustack(), 40–41, 1008, 1017, 1071
- Active (TCP term), 482
- Active service time, 213
- Adaptive-block, 920–921
- Adaptive mutex, 1015, 1077
- Address family, 449–454, 1015
- Administrator privileges, 868–869
- Advanced Host Controller Interface (AHCI), 237, 289
- AES_encrypt() function, 651–654
- AF_INET, AF_INET6, 449–452
- aggrate, 1006, 1016–1017
- Aggregation drops (error), 1065
- Aggregations, 13–14, 1077
 - buffers, 1006, 1065, 1077
 - functions, 33–34, 1017–1018
 - lquantize(), 35
 - normalize(), 36
 - printa(), 36–37
 - quantize(), 34–35
 - trunc() and clear(), 36
 - types, 34
 - variables, 999, 1017
- aggszize, 1006, 1065, 1077
- aggsortkey, 37
- aggsortkeypos, 37
- aggsortpos, 37, 459
- aggsortrev, 37
- ahci, 237, 289
- aio_read(), 306
- Alert (\a), 1021
- Analytics
 - abstractions, 974
 - breakdown statistics, 979
 - control bar, 983
 - control descriptions, 983
 - controls, 983
 - datasets, 984
 - diagnostic cycle, 975
 - drill-downs, 981–983
 - heat maps, 979–981
 - hierarchical breakdowns, 979–980
 - load *vs.* architecture, 975
 - real time, 975
 - shouting in the data center, 269–273
 - statistics, 977
 - visualizations, 975
 - worksheets, 983
- Anchored probes, 1077
- Anonymous memory segment, 103
- Anonymous state, 1007
- Anonymous tracing, 917–918
- Apache, 610–611
- Apache Web server, 560, 732, 783–784, 800
- Appends output, 43, 1071
- Apple, 370, 620, 949, 972
 - see also* Mac OS X
- Application-level protocols
 - capabilities, 400–401
 - checklist, 559–560
 - providers
 - fbt provider, 561
 - iSCSI scripts, 634–638
 - one-liners
 - fc provider, 568
 - http provider, 567
 - http provider examples, 573
 - iscsi provider, 567
 - nfsv3 provider, 563
 - nfsv4 provider, 564–566
 - NFSv3 provider examples, 569–571
 - NFSv4 provider examples, 571–572
 - smb provider, 566
 - smb provider examples, 572–573
 - syscall provider, 563
 - syscall provider examples, 568–569
 - pid provider, 562
 - scripts
 - CIFS scripts
 - cifserrors.d, 605–607
 - cifsfbtinfo.d, 607–609
 - cifsfileio.d, 603
 - cifsops.d, 602–603
 - cifsrwnoop.d, 600–601
 - cifsrwtime.d, 604
 - DNS scripts
 - dnsgetname.d, 623–625
 - getaddrinfo.d, 622–623
 - Fibre Channel scripts
 - fcerror.d, 647–649
 - fcwho.d, 647

- FTP scripts
 - ftpdfileio.d, 626–627
 - ftpdxfer.d, 625–626
 - proftpdcmd.d, 627–629
 - proftpdio.d, 632–633
 - proftpdtime.d, 630–632
 - tnftpdcmd.d, 630
- HTTP scripts
 - httpclients.d, 612–613
 - httpdurls.d, 616–618
 - httperrors.d, 614
 - httpio.d, 614–615
 - weblatency.d, 618–621
- iSCSI scripts
 - iscsicmds.d, 643–644
 - iscsirwsnoop.d, 640–641
 - iscsirwtime.d, 641–643
 - iscsiterr.d, 644–646
 - iscsiwho.d, 638–639
 - providers, 634–638
- LDAP scripts
 - ldapsyslog.d, 664–666
- multiscripts, 666–668
- network script summary, 574–576
- NFSv3 scripts
 - nfsv3commit.d, 585–587
 - nfsv3errors.d, 588–590
 - nfsv3fbtrws.d, 590–592
 - nfsv3fileio.d, 581
 - nfsv3ops.d, 580
 - nfsv3rwsnoop.d, 578–579
 - nfsv3rwtime.d, 582–583
 - nfsv3syncwrite.d, 584
- NFSv4 scripts
 - nfsv4commit.d, 595
 - nfsv4deleg.d, 597–599
 - nfsv4errors.d, 595–597
 - nfsv4fileio.d, 594
 - nfsv4ops.d, 594
 - nfsv4rwsnoop.d, 594
 - nfsv4rwtime.d, 595
 - nfsv4syncwrite.d, 595
- NIS scripts, 663–664
- SSH scripts
 - scpwatcher.d, 661–663
 - sshcipher.d, 649–655
 - sshconnect.d, 657–661
 - sshdactivity.d, 655–657
- strategy, 558–559
- Applications
 - capabilities, 784
 - case studies
 - Firefox idle, 817–824
 - Xvnc, 824–832
 - checklist, 786–787
 - providers
 - cpc provider, 791–792
 - one-liner examples
 - new processes (with arguments), 798–799
 - system call counts, 800
 - user-mode instructions, 801–803
 - user-mode level-two cache misses, 803–804
 - user stack trace profile at 101 hertz, 800–801
 - one-liners
 - cpc provider, 797–798
 - pid provider, 795–796
 - plockstat provider, 796
 - proc provider, 793–794
 - profile provider, 794–795
 - sched provider, 795
 - syscall provider, 794
 - pid provider, 788–791
 - script summary, 804
 - scripts
 - execsnoop, 805–806
 - kill.d, 813–814
 - plockstat, 811–813
 - procsnoop.d, 804–806
 - procsystime, 806–808
 - sigdist.d, 814–815
 - threaded.d, 815–816
 - uoffcpu.d, 809–811
 - uoncpu.d, 808–809
 - strategy, 784–786
- Arc_get_data_buf(), 901
- Architecture, 16–17
- arg0, 31, 61, 174–175
- arg1, 61
- Args[], 31
- Arguments
 - bufinfo_t, 1038
 - conninfo_t, 1040
 - cpuinfo_t, 1039
 - csinfo_t, 1040
 - devinfo_t, 1038
 - fileinfo_t, 1038
 - ifinfo_t, 1041
 - ipinfo_t, 1040
 - ipv4info_t, 1041
 - ipv6info_t, 1041
 - lwpsinfo_t, 1039
 - pktinfo_t, 1040
 - psinfo_t, 1039
 - tcpinfo_t, 1042
 - tcplsinfo_t, 1043
 - tcpsinfo_t, 1042

Arguments and return value
 kernel functions, 901–903
 pid provider, 791
 Arithmetic operators, 27, 1021–1022
 Array operators, 545
 Assembly language, 677–679
 Assignment operators, 27
 Associative arrays, 29, 81, 200, 240, 285–286, 1078
 Assumptions, 1000
 Asynchronous writes, 332, 584
 Asynchronous write workloads, 241
 AT attachment disk driver, 251
 ata, 251
 Automatic drilldown analysis, 964
 avg() function, 34, 81
 awk, 20

B

-b flag, 1003, 1006
 Backquote (`) character, 33, 64, 78, 231
 Backslash (\), 1021
 Backspace (\b), 491, 1021
 Bart, 206–207
 basename(), 736
 B_ASYNC, 159
 B_DONE, 159
 B_ERROR, 159
 b_flags, 157–159, 178
 B_PAGEIO, 159
 B_PHYS, 159
 B_READ, 159
 B_WRITE, 159
 BEGIN, 44
 BEGIN and END, 24
 Berkeley Internet Name Daemon (BIND), 575, 623–624
 Binary arithmetic operators, 1021
 Binary assignment operators, 1023
 Binary bitwise operators, 1022
 Binary logical operators, 1022
 Binary relational operators, 1022
 Birrell, John, 1047
 Bitwise operators, 28, 1022–1023
 Blank fields, 24
 Blowfish, 651, 654
 Boolean operators, 28
 Boot processes, 917–918
 Bourne shell, 764–774
 Bourne shell provider, 1052–1061
 Breakdowns, 979
 broken.php, 733, 736
 bsdtar(1) command, 164, 210
 Buckley, Joel, 229

buffer-read-done, 852
 buffer-read-start, 852
 Buffer resizing, 1006
 buffer-sync-start, 852
 buffer-sync-written, 852
 bufinfo_t, 157–159, 1026, 1038
 bufpolicy, 1006, 1078
 bufresize, 1006
 Bufsize, 1006
 bufsize, 43, 1064, 1071, 1084
 Built-in functions, 1014–1019
 Built-in variables, 31–32, 1011–1013
 Bus adapter driver, 234
 Bytes read by filename, 302, 309–310
 Bytes written by filename, 302, 310

C

-c, 43, 528, 788, 795–796, 1006
 -C, 231, 478, 683–684
 C (language)
 includes and the preprocessor, 683
 kernel C, 681
 one-liner examples
 count kernel function calls, 688
 function entry arguments, 687
 user stack trace, 687–688
 one-liners
 fbt provider, 685–686
 pid provider, 684–685
 profile provider, 686–687
 probes and arguments, 681–682
 script summary, 689
 scripts, 689
 struct types, 682–683
 user-land C, 680
 C++ language, 689–691
 Cache allocations, 909–911, 922
 Cache file system read, 331–332
 Cache misses, 923–924
 Cantrill, Bryan, 269, 661, 973, 1003
 Carriage return (\r), 1021
 Case studies
 Bourne shell provider, 1057–1061
 disk I/O, 269–290
 file systems, 387–398
 Firefox idle, 817–824
 Xvnc, 824–832
 cd(1), 301
 CD-ROMs, 376–378
 c++filt, 432, 690
 char, 26
 Character escape sequences, 1021
 chdir(), 569

Cheat sheet, 1069–1072
 Chime (tool), 962–965
 CIFS, 1078

- count of operations by client address, 572
- count of operations by file path name, 573
- frequency of operations by type, 572
- read I/O size distribution, 573

 CIFS scripts

- cifserrors.d, 575, 605–607
- cifsfbtncfile.d, 575, 607–609
- cifsfileio.d, 575, 603
- cifsops.d, 575, 602–603
- cifsrswnoop.d, 575, 600–601
- cifsrwtime.d, 575, 604

 cipher, 649–655
 cipher_crypt(), 653
 Class-loading probes, 691
 Clause, 9, 21
 Clause-local variables, 30–31, 998
 cleanrate, 1006
 clear(), 34, 36
 CLI queries, 842–843
 Client initiator, 636
 Client-server components, 835
 close(), 445
 cmdk, 251
 cnwrite(), 885
 Command-line aliases, 1005
 Command-line hints, 161–162
 Comment / uncomment characters, 1078, 1088
 Common Internet File System. *see* CIFS
 Compact C Type Format (CTF), 682, 1079
 Compression, 652
 COMSTAR, 634, 638–640
 Conditional branch misprediction, 798, 924
 Conditional statements, 22
 Connection latency, 414
 connection-start/connection-done, 838
 connections, 399
 conninfo_t, 1040
 Contention. *see* Locks and lock contention
 Context switch time, 943
 Controls, 983
 Cool Stack, 731
 copyin(), 39, 624, 1002, 1014, 1067
 copyinstr(), 39–40, 304, 622–624, 679, 687, 1002, 1014, 1067
 count(), 34
 Count file systems calls, 302–303
 Count function calls, 710–711, 735, 742, 767
 Count interrupts, 921
 Count kernel function calls, 688
 Count line execution by filename and line number, 754, 767

Count method calls by filename, 754
 Count of operations, 563, 570
 Count subroutine calls by file, 721–722, 741
 Count system calls, 45, 824, 925
 cpc provider, 787, 791–792, 797–798, 923–925
 CPU cross calls by kernel stack trace, 928
 CPU events, 791–792
 CPU Performance Counter (cpc). *see* cpc provider
 cpuinfo_t, 1039
 CPUs, 846–847, 856
 CPUs, tracking

- analysis, 60–85
- checklist, 57–58
- events, 87–94
- interrupts, 85–88
- one-liners, 58–60
- providers, 58

 cpustat(1M), 791–792, 803
 CR (Change Request), 1079
 Cross calls, 390–393, 897, 928
 crypt functions, 650
 csinfo_t, 409, 1040
 curpsinfo, 31
 curthread, 31

D

-D (dtrace(1M)), 199, 231
 D language, 14–16

- actions
 - copyin(), 39
 - copyinstr(), 39–40
 - exit(), 41
 - jstack(), 40–41
 - list of, 1014–1019
 - printf(), 38
 - sizeof(), 41
 - speculations, 41–42
 - stack(), 40–41
 - stringof(), 39–40
 - strjoin(), 40
 - strlen(), 40
 - trace(), 37
 - tracemem(), 39
 - translators, 42
 - ustack(), 40–41
- aggregations
 - lquantize(), 35
 - normalize(), 36
 - printa(), 36–37
 - quantize(), 34–35
 - trunc() and clear(), 36
 - types, 34

D language (*continued*)

components

- actions, 23
- predicates, 22
- probe format, 21–22
- program structure, 21
- usage, 20–21

example programs

- counting system calls by a named process, 45
- Hello World, 44
- profiling process names, 46–47
- showing read byte distributions by process, 45–46
- snoop process execution, 48–49
- timing a system call, 47–48
- tracing fork() and exec(), 45

options, 43–44

probes

- BEGIN and END, 24
- profile and tick, 24–25
- syscall entry and return, 25
- wildcards, 23–24

variables

- associative arrays, 29
- built-in, 31–32
- clause local, 30–31
- external, 33
- macro, 32
- operators, 27–28
- scalar, 28
- structs and pointers, 29
- thread local, 30
- types, 26–27

dad (driver), 251

Data cache misses by function name, 931

Data corruption, 242

Data recording actions, 1016–1017

Databases

capabilities, 834–835

client-server components, 835

MySQL

- one-liner examples, 840–841
- one-liners, 838–840
- script summary, 841
- scripts, 841–851
 - libmysql_snoop.d, 849–850
 - mysqld_pid_qtime.d, 848–849
 - mysqld_qchit.d, 844–845
 - mysqld_qlower.d, 846–847
 - mysqld_qsnoop.d, 841–844

Oracle, 858–865

PostgreSQL

- one-liner examples, 854–858

one-liners, 853–854

- pid provider, 854
- postgresql provider, 853
- script summary, 855
- scripts, 854–858

- providers, 836–837
- strategy, 835–836

Datasets, 984

dcmd (d-command, mdb(1)), 99

Debuggers/debugging, 2–3, 261, 671, 682, 800, 898, 1005

Decryption, 871

Default cipher, 650

defaultargs, 43, 334, 360, 373, 656, 1007, 1071

DES_encrypt3(), 650

destructive, 43, 886–890, 1007, 1018–1019

Device drivers, 537–543, 917–918

Device insertion, 242–243

devinfo_t, 160, 1038

DFCI, 534

Diagnostic cycle, 975

DIF (DTrace Intermediate Format), 1079

Direct Memory Access, 242

Directory Name Lookup Cache (dnlc), 314, 346, 952

Dirty data, 310, 332, 347, 349, 369

Disk and network I/O activity

- analysis, 128–134
- checklist, 125
- disk I/O, 134–141
- one-liners, 127–128
- providers, 126–127
- strategy, 125

Disk I/O

capabilities, 152–154

case studies, 269–290

checklist, 155–156

IDE scripts, 250

ideerr.d, 173, 257

idelatency.d, 173, 252–254

iderw.d, 173, 255–257

io provider scripts

- bitesize.d, 181–183
- disklatency.d, 172, 175–177
- geomiosnoop.d, 172, 209–210
- iolatency.d, 172–175
- iopattern, 172, 207–209
- iosnoop, 172, 187–203
- iotop, 204–207
- iotypes.d, 178–179
- rwtime.d, 179–181
- seeksize.d, 184–187

providers

- fbt provider, 163–166

- io provider, 157–163, 165
 - one-liner examples, 166–171
 - one-liners, 165–166
 - SAS scripts
 - mptevents.d, 173, 264–267
 - mptlatency.d, 173, 267–269
 - mptsassscsi.d, 173, 263–267
 - SATA scripts
 - satacmds.d, 172, 237–243
 - satalatency.d, 173, 248–250
 - satareasons.d, 173, 246–248
 - satarw.d, 172, 243–246
 - scsi.d, 172, 236
 - SCSI scripts
 - SCSI probes, 212–213
 - scsicmds.d, 172, 218–221
 - scsi.d, 229–236
 - scsilatency.d, 172, 221–223
 - scsireasons.d, 172, 227–229
 - scsirw.d, 172, 223–226
 - sdqueue.d, 172, 213–215
 - sdretry.d, 172, 215–218
 - size aggregation, 167
 - size by process ID, 166–167
 - size distribution, 840–841
 - strategy, 154–155
 - Disk queueing, 201–202
 - Disk reads and writes, 232–233
 - Disk reads with multipathing, 234
 - Disk time, 205–206
 - Dispatcher queue, 529, 938, 950, 1079
 - Displays (Chime), 963–964
 - Distribution plots, 13–14, 34, 45, 98, 165, 310–312, 433, 841, 855
 - DLight, Oracle Solaris Studio 12.2, 966–971
 - DLPI, 534
 - dmake, 308
 - dnlcps.d, 314, 346–347, 952
 - DNS scripts, 621
 - dnsgetname.d, 623–625
 - getaddrinfo.d, 622–623
 - do_copy_fault_nta(), 931
 - DOF (Dtrace Object Format), 1079
 - done probe, 157
 - doorfs(), 658
 - Double quote, 880, 1021
 - Downloading and installing
 - Chime, 962–963
 - DTrace GUI plug-in, 966
 - DTraceToolkit, 948–949
 - Mac OS X Instruments, 971–972
 - Drill-downs, 964–965, 981–983
 - Driver interface, 542
 - Driver internals, 538
 - Drops, 699, 870, 935, 1003, 1064–1066
 - DTrace GUI Plug-in for NetBeans and Sun Studio, 966
 - DTrace Guide. See *Solaris Dynamic Tracing Guide*
 - dtrace provider, 11
 - dtrace(7d), 1080
 - dtrace_kernel, 868, 872, 1064
 - dtrace(1M), 19, 1080
 - dtrace_proc, 868, 1064
 - DTraceToolkit
 - downloading and installing, 948–949
 - man page, 959–960
 - script example: cpuwalk.d, 957–961
 - script summary, 949–957
 - scripts, 949–957
 - versions, 949
 - dtrace_user, 868, 1064
 - DVDs, 378–379
 - Dynamic probes, 4, 1080
 - dynvardrops, 343, 1003, 1066, 1079–1080
 - dynvarsize, 43, 1003, 1007, 1066
- ## E
- egrep(1), 539
 - Elevator seeking, 199
 - Encrypted sessions, 871
 - enqueue probe (sched provider), 84
 - Entropy stat, 709, 711–712
 - Entry (syscall), 25
 - Erickson, Tom, 962
 - er_kernel (kernel profiler tool), 966
 - errno, 25, 31, 794, 1080
 - Error(s)
 - ciferrors.d, 605–606
 - codes, 467–468
 - disk I/O, 156
 - error messages
 - aggregation drops, 1065
 - drops, 1064–1065
 - dynamic variable drops, 1066
 - invalid address, 1066–1067
 - maximum program size, 1067
 - not enough space, 1067
 - privileges, 1063–1064
 - file system I/O, 297
 - fserrors.d, 326–327
 - httperrors.d, 614
 - network I/O, 404
 - network I/O checklist, 404
 - nfsv3errors.d, 588
 - nfsv4errors.d, 595
 - number, 170–171
 - PHP, 736

Error(s) (*continued*)

- socket system call errors, 467–468
- soerrors.d, 465
- translation table, 595–597

Ethernet scripts

- device driver tracing, 537–543
- Mac tracing with fbt, 534
- macops.d, 534–537
- ngelink.d, 546–547
- ngesnnoop.d, 544–546

Ethernet vs. Wi-Fi, 462**Example programs**

- counting system calls, 45
- Hello World, 44
- profiling process names, 46–47
- showing read byte distributions by process, 45–46
- snoop process execution, 48–49
- timing a system call, 47–48
- tracing fork() and exec(), 45
- tracing open(2), 44–45

Exclusive time, 703, 718, 730, 750, 762**execname**, 31, 110, 1080**exec_simple_query()**, 854**execsnnoop**, 805–806**exit()**, 41, 1002, 1014**ExtendedDTraceProbes**, 692, 694, 696**External Data Representation**. *see* XDR**External variables**, 33**F****-F (dtrace(1M))**, 438, 1007**Failed to enable probe (error)**, 792**Fast File System (FFS)**, 351**fasttrap**, 868, 1064, 1068**fbt**, 12, 155–156, 163–166, 170, 298, 352, 405**fbt-based script maintenance**, 418**fc provider**, 568, 646**fc provider probes and arguments**, 1025–1026**FC (Fibre Channel) scripts**, 646

fcerror.d, 647–649

fewho.d, 647

fds[], 68, 131, 145, 300, 429, 1080**fds[].fi_fs** variable, 91**fdsp[.fi_dirname]** variable, 161**File System Archive**, 807**File systems**

- capabilities, 292–295
- case study, 387–398
- checklist, 296–297
- functional diagram, 293
- providers
 - fsinfo provider, 298–300

one-liners

- fbt provider, 303
- fsinfo provider, 302
- sdt provider, 303
- syscall provider, 300–301
- vfs provider, 303
- vminfo provider, 302

one-liners: fsinfo provider examples

- bytes read by filename, 309–310
- bytes written by filename, 310
- calls by fs operation, 308–309
- calls by mountpoint, 309
- read/write I/O size distribution, 310–312

one-liners: sdt provider examples, 312–313**one-liners: syscall provider examples**

- frequency count stat() files, 305
- reads by file system type, 306–307
- trace file creat() calls with process name, 304–305
- trace file opens with process name, 304
- tracing cd, 306
- writes by file system type, 307
- writes by process name and file system type, 307

one-liners: vminfo provider examples, 308**scripts****fsinfo scripts**

- fssnoop.d, 333–335
- fswho.d, 328
- readtype.d, 329–332
- writetype.d, 332–333

HFS+ scripts

- hfsfileread.d, 374–375
- hfslower.d, 372–374
- hfsnnoop.d, 371–372

HSFS scripts, 376–378**NFS client scripts**

- nfs3fileread.d, 383–384
- nfs3sizes.d, 382–383
- nfswizard.d, 379–381

PCFS scripts, 375–376**syscall provider**

- fserrors.d, 326–327
- fsrtpk.d, 320–322
- fsrwcoun.d, 317–319
- fsrwtime.d, 319–320
- mmap.d, 324–325
- rwsnnoop, 322–323
- sysfs.d, 315–317

TMPFS scripts

- tmpgetpage.d, 386–387
- tmpusers.d, 385–386

UDFS scripts, 378–379

- UFS scripts
 - ufsimiss.d, 356–357
 - ufsreadahead.d, 354–356
 - ufssnoop.d, 352–354
 - VFS scripts
 - dnlcps.d, 346–347
 - fsflush_cpu.d, 347–349
 - fsflush.d, 349–351
 - maclife.d, 344–345
 - macvfssnoop.d, 338–340
 - sollife.d, 343–344
 - solvfssnoop.d, 336–338
 - vfslife.d, 345
 - vfssnoop.d, 340–343
 - ZFS scripts
 - perturbation.d, 366–368
 - spasync.d, 369–370
 - zffslower.d, 360–361
 - zffssnoop.d, 358–359
 - zioprint.d, 361–363
 - ziosnoop.d, 363–365
 - ziotype.d, 365–366
 - strategy, 295–296
 - write operation, 295
 - Filebench, 296, 559, 989
 - fileinfo_t, 160–161, 1038
 - filesort (probes), 838
 - filesort-start/filesort-done, 838
 - fill buffer, 1080, 1084
 - Find *vs.* Bart, 206–207
 - Firefox case study
 - fetching context, 822
 - function counts and stacks, 819
 - function CPU time, 820
 - profiling user modules, 818
 - profiling user stacks, 817
 - First-byte latency, 460–461, 499
 - Floating-point data types, 1020
 - Floating-point suffixes, 1021
 - Floating-point types, 27
 - flowindent, 43, 104, 438, 684, 685, 896, 903–906, 1007
 - flush write-cache, 241–242
 - fop interface, 303, 336, 349
 - Formfeed (`\f`), 1021
 - FreeBSD, 164, 949, 1075, 1080
 - AF_INET values, 451
 - hyphens in probe names, 793*n*
 - iostat(8), 125
 - kmem layer, 123
 - netstat(8), 125
 - stack trace, 170, 421
 - system tools, 55
 - FreeBSD 7.1 and 8.0
 - installing DTrace, 1045–1046
 - Frequency count, 991–992
 - Frequency count fbt, 166, 278–279
 - Frequency count functions, 166, 171
 - Frequency count sdt, 276–277
 - fsinfo, 126, 132
 - fsinfo provider, 298–300, 302
 - fsinfo provider examples, 308–312
 - fsinfo provider probes and arguments, 1026
 - fsinfo scripts
 - fssnoop.d, 333–335
 - fswho.d, 328
 - readtype.d, 329–332
 - writetype.d, 332–333
 - FTP Analytics, 625
 - FTP scripts
 - ftpdfileio.d, 626–627
 - ftpdxfer.d, 625–626
 - proftpcmd.d, 627–629
 - proftpdio.d, 632–633
 - proftpdtime.d, 630–632
 - tnftpcmd.d, 630
 - Function arguments, 283–285
 - Function Boundary Tracing. *see* fbt
 - Function counts and stacks, 819
 - Function CPU time, 820
 - function-entry, 752
 - Function execution, 672–673
 - Function names, 690, 1014–1019
 - function-return, 752
- ## G
- Garbage collection, 691, 751, 753, 759, 820
 - gc++filt, 432, 690
 - GEOM, 164, 172, 209–210
 - Gerhard, Chris, 229
 - GET, 616
 - gld, 405
 - GLDv3, 534, 1081
 - Global and aggregation variables, 33, 350, 997–999
 - Global zone, 870–872
- ## H
- h (dtrace1M), 807
 - Hardware address translation (HAT), 928
 - Hargreaves, Alan, 1051
 - Haslam, Jon, 350
 - HC (High Capacity), 407
 - Header files, 683
 - Heat maps, 979–981
 - Hertz rates, 24–25, 61
 - HFS+, 370, 929

- HFS+ scripts
 - hfsslower.d, 372–374
 - hfssnoop.d, 371–372
 - hfs_file_is_compressed(), 929
 - HIDS (Host-based Intrusion Detection Systems), 871
 - Hierarchal File System. *see* HFS+
 - Hierarchical breakdowns, 979–980
 - High Sierra File System (HSFS), 376
 - Hold events, 812
 - Horizontal tab (\t), 1021
 - Host name lookup latency, 660
 - Hot code paths, 897, 944, 990, 996
 - hotkernel, 64
 - hotspot, 675, 691, 694
 - HotSpot VM, 691
 - hotuser, 64
 - HSFS scripts, 376–378
 - HTTP
 - flow diagram, 610
 - scripts
 - httpclients.d, 612–613
 - httpdurls.d, 616–618
 - httperrors.d, 614
 - httpio.d, 614–615
 - weblatency.d, 618–621
 - summarize user agents, 573
 - HTTP files opened by the httpd server, 563, 568
 - http provider, 567
 - http provider examples, 573
 - httpd, 563, 568, 783–784, 795, 802–803
 - Hyphens in probe names, 793*n*
- I**
- %I, 488
 - IA (interactive scheduling class), 942
 - ICMP, 1081
 - ICMP event by kernel stack trace, 424, 439
 - ICMP event trace, 424, 437
 - ICMP scripts
 - icmpsnoop.d, 447, 522–525
 - icmpstat.d, 447, 521
 - superping.d, 447, 526–529
 - IDE driver reference, 251
 - IDE scripts
 - ideerr.d, 257–259
 - idelatency.d, 252–254
 - iderw.d, 255–257
 - ifinfo_t, 410, 1041
 - Inbound TCP connections, 441, 446, 486–487, 489
 - Inclusive time, 703, 718, 730, 750, 762, 908
 - inet*() functions, 590, 608
 - inet_ntoa(), 455, 502
 - inet_ntoa6(), 455, 502
 - inet_ntop(), 451
 - Instruction cache misses by function name, 931–932
 - Instruments (Mac OS X tool), 971–972
 - Integer data types, 1020
 - Integer suffixes, 1021
 - Integer type aliases, 1020
 - Integer variable types, 26–27
 - Internet Control Message Protocol (ICMP). *see* ICMP
 - Internet Small Computer System Interface (iSCSI). *see* iSCSI
 - Interrupt load, 58
 - Interrupt start count, 921
 - Interrupts, 85–88, 932, 962
 - intrstat(1M), 16, 85, 932–934
 - Intrusion detection, 871, 886
 - Invalid address (error), 1066–1067
 - Invasion of privacy issues, 875–877
 - I/O
 - analysis, 130
 - checklist, 127
 - one-liners, 129
 - providers, 128
 - strategy, 127
 - io probes, 157–158
 - io provider, 165, 637–638, 840
 - bufinfo_t, 158–159
 - command-line hints, 161–162
 - devinfo_t, 160
 - fileinfo_t, 160–161
 - probes and arguments, 1026
 - io provider scripts
 - bitesize.d, 172, 181–183
 - disklatency.d, 172, 175–177
 - geomiosnoop.d, 209–210
 - iolatency.d, 172–175, 270
 - iopattern, 207–209
 - iosnoop, 187–203
 - iotop, 172, 204–207
 - iotypes.d, 178–179
 - rwtime.d, 179–181
 - seeksize.d, 184–187
 - iostat(8), 55, 125
 - iostat(1M), 55, 125, 134, 288, 863
 - iotop, 204–207
 - IP event statistics, 424, 435
 - IP-layer network traffic, 126
 - ip probe arguments, 408
 - ip provider, 404, 425
 - csinfo_t, 409
 - ifinfo_t, 410
 - ipinfo_t, 409

- ip_v4info_t, 410
- ip_v6info_t, 410
- pktinfo_t, 409
- ip provider development, 473
- ip provider examples, 440
- ip provider probes, 408
- ip provider probes and arguments, 1027
- IP scripts
 - fbt provider, 470–474
 - ipfbtsnoop.d, 478–481
 - ipio.d, 475–477
 - ipproto.d, 477–478
 - ipstat.d, 474–475
- ipfbtsnoop.d, 446, 478–481
- ipIfStatsHCInOctets (probe), 407
- ipIfStatsHCOctets (probe), 407
- ipinfo_t, 409, 1040
- ip_input(), 481, 555
- ipio.d, 446, 475–477
- ip_output(), 419, 555
- ipproto.d, 446, 477–478
- ipstat.d, 446, 474–475
- ip_v4info_t, 410
- ip_v4info_t, 1041
- ip_v6info_t, 410
- ip_v6info_t, 1041
- iSCSI
 - client initiator, 636
 - functional diagram, 634
 - provider, 667, 635
 - target server, 635
- iscsi provider probes and arguments, 1027
- iSCSI scripts
 - iscsicmds.d, 643–644
 - iscsirwsnoop.d, 640–641
 - iscsirwtime.d, 641–643
 - iscsiterr.d, 644–646
 - iscsiwho.d, 638–639
 - providers
 - fbt provider, 635–637
 - io provider, 637–638
 - iscsi provider, 635
- iscsi_iodone(), 637
- iscsit_op_scsi_cmd(), 636
- iscsit_xfer_scsi_data(), 636

J

- Java
 - code, 693
 - one-liner examples, 694–696
 - one-liners, 693–694
 - script summary, 696
 - scripts

- j_calls.d, 696–698
 - j_calltime.d, 701–704
 - j_flow.d, 698–700
 - j_thread.d, 704–705
- Java virtual machine (JVM), 691
- JavaScript (language)
 - code, 707–708
 - one-liner examples
 - count function calls by function filename, 710–711
 - object entropy stat, 711–712
 - trace function calls, 710
 - trace program execution showing filename and line number, 709
 - one-liners, 708–709
 - script summary, 712
 - scripts
 - js_calls.d, 712–713
 - js_calltime.d, 715–718
 - js_flowinfo.d, 670, 713–715, 952
 - js_stat.d, 718
- JavaScript Garbage Collect, 820
- JBODs, 269–273
- JNI functions, 692
- Joyent, 751
- jstack(), 40–41, 108, 743, 1017
- jstackframes, 1007, 1017
- jstackstrsize, 44, 1007, 1017

K

- kalloc(), 916–917
- Kernel
 - capabilities, 894–895
 - checklist, 897–898
 - clock interrupt, 61
 - destructive actions, 1018
 - functional diagram, 895
 - ktrace.d, 903–906
 - lock events, 934–935
 - memory
 - allocation, 122, 914–915, 922
 - Mac OS X, 122–124
 - tools, 118–120
 - memory allocations, 915–916
 - profiler tool, 966
 - profiling, 64–70, 72
 - providers
 - anonymous tracing, 917–918
 - fbt provider
 - arguments and return value, 901–903
 - module name, 900–901
 - probe count, 899–900
 - stability, 898–899

Kernel, providers (*continued*)

- kernel memory usage, 908–917
- kernel tracing, 903–908
- one-liner examples
 - count system calls by type, 925
 - CPU cross calls by kernel stack trace, 928
 - kernel function call counts for functions
 - beginning with `hfs_` by module, 929
 - kernel-mode instructions by function name, 930
 - kernel-mode instructions by module name, 930–931
 - kernel-mode level-one data cache misses by function name, 931
 - kernel-mode level-one instruction cache misses by function name, 931–932
 - kernel module name profile at 1001 hertz, 927
 - kernel stack backtrace counts for calls to function `foo()`, 929
 - kernel stack trace profile at 1001 hertz, 925–927
 - kernel thread name profile at 1001 hertz (`freebsd:`), 928
- one-liners
 - cpc provider, 923–925
 - fbt provider, 921–923
 - lockstat provider, 920–921
 - profile provider, 919
 - sched provider, 920
 - sdt provider, 921
 - syscall provider, 919
 - sysinfo provider, 920
 - vminfo provider, 920
- script summary, 932
- scripts
 - `cswstat.d`, 932, 943–944
 - `intrstat`, 932–934
 - `koffcpu.d`, 932, 938–939
 - `koncpu.d`, 932, 937–938
 - `lockstat`, 934–937
 - `priclass.d`, 932, 941–943
 - `putnexts.d`, 932, 944–945
 - `taskq.d`, 932, 939–941
- stacks, 168–170
- statistics, 896
- strategy, 896–897
- Kernel file system flush thread, 347
- `kernel_memory_allocate()`, 122–123, 914–915, 922
- keycache (probes), 838
- Keys, 15, 33, 36, 1082
- Keystroke captures, 875–876
- Keywords, table of, 1019

KILL signal, 888, 890

- `kmem`, 119
 - `kmem_alloc()`, 911–912, 916
 - `kmem_cache_alloc()`, 909–910
 - `kmem_cache_free()`, 910
 - `kmem_free()`, 916
- `kstat(1M)`, 55, 118, 896, 983

L

`-l (dtrace(1M))`, 1071

Languages

Assembly, 677–679

C

- includes and the preprocessor, 683
- kernel C, 681
- one-liner examples
 - count kernel function calls, 688
 - show user stack trace, 687–688
 - trace function entry arguments, 687
- one-liners
 - fbt provider, 685–686
 - pid provider, 684–685
 - profile provider, 686–687
- probes and arguments, 681–682
- scripts, 689
- struct types, 682–683
- user-land C, 680

C++, 690–691

capabilities, 671–672

checklist, 674

Java

- code, 693
- one-liner examples, 694–696
- one-liners, 693–694
- scripts
 - `j_calls.d`, 696–698
 - `j_calltime.d`, 701–704
 - `j_flow.d`, 698–700
 - `j_thread.d`, 704–705

JavaScript

- code, 707–708
- `js_stat.d`, 718
- one-liner examples
 - count function calls by function filename, 710–711
 - object entropy stat, 711–712
 - trace function calls showing function name, 710
 - trace program execution showing filename and line number, 709
- one-liners, 708–709
- scripts
 - `js_calls.d`, 712–713

- js_calltime.d, 715–718
 - js_flowinfo.d, 713–715
 - Perl
 - code, 720
 - one-liner examples, 721–722
 - one-liners, 720–721
 - scripts
 - pl_calls.d, 723–725
 - pl_calltime.d, 728–731
 - pl_flowinfo.d, 725–728
 - pl_who.d, 722–723
 - PHP
 - code, 733
 - one-liner examples
 - count function calls by filename, 735
 - trace function calls showing function name, 735
 - trace PHP errors, 736
 - one-liners, 734–735
 - script summary, 736
 - scripts
 - php_calls.d, 736
 - php_flowinfo.d, 738
 - providers, 675–679
 - Python
 - code, 741
 - one-liner examples
 - count function calls by file, 742
 - profile stack traces, 743–744
 - trace function calls, 742
 - one-liners, 741
 - scripts
 - py_calls.d, 745–746
 - py_calltime.d, 748–751
 - py_flowinfo.d, 746–748
 - py_who.d, 744–745
 - Ruby
 - code, 752
 - one-liner examples
 - count line execution by filename and line number, 754
 - count method calls by filename, 754
 - trace method calls showing class and method, 754
 - one-liners, 753
 - scripts
 - rb_calls.d, 756–757
 - rb_calltime.d, 759–762
 - rb_flowinfo.d, 757–759
 - rb_who.d, 755–756
 - scripting, 669
 - Shell
 - code, 765
 - one-liner examples
 - count function calls by filename, 767
 - count line execution by filename and line number, 767
 - trace function calls showing function name, 766
 - one-liners, 765–766
 - scripts
 - sh_calls.d, 769–771
 - sh_flowinfo.d, 771–774
 - sh_who.d, 768–769
 - strategy, 672–673
 - Tcl
 - code, 776
 - one-liner examples, 777–778
 - one-liners, 776–777
 - scripts
 - tcl_calls.d, 779–780
 - tcl_insflow.d, 782
 - tcl_procfow.d, 780–782
 - tcl_who.d, 778–779
 - Latency, 156
 - disk I/O, 285–287, 269–273
 - by driver instance, 234–236
 - file system I/O, 296
 - heat maps, 980
 - network I/O checklist, 403
 - TCP connection, 414
 - latency.d, 288
 - LDAP scripts, 664–666
 - Leventhal, Adam, 1, 1003
 - libc, 105, 680, 684, 789, 829
 - libc fsync() calls, 796
 - libc function calls, 795
 - libcurses, 788–789
 - libdtrace(3LIB), 1082
 - Libmysql_snoop.d, 849–850
 - libsocket, 789–790
 - libssl (Secure Sockets Layer library), 784
 - Local ports, 442
 - Locks and lock contention, 58, 87–88, 674, 787, 811–813, 816, 897, 935
 - lockstat(1M), 12, 16, 62, 87, 811–813, 920–921, 934–937
 - Logical operators, 1022
 - Loopback traffic, 408, 493, 525
 - lquantize(), 34, 35, 270, 630
 - lwpid, 81–82
 - lwpsinfo_t, 1039
- ## M
- Mac OS X
 - AF_INET values, 451
 - disk I/O, 177

- Mac OS X (*continued*)
 - ether_frameout(), 418
 - fbt provider, 418, 421, 474
 - iostat(8), 125
 - kernel memory allocation, 122
 - netstat(8), 125
 - system tools, 55
 - Mac OS X Instruments, 971–972
 - Mac OS X Internals, 296, 370
 - Mac OS X Interprocess Communication (IPC) and IO Kit path, 915
 - Mac OS X tracing with fbt, 534
 - mach_kernel, 900
 - Macro variables, 32
 - MacRuby, 751
 - Maguire, Alan, 500
 - malloc(), 674, 676, 763, 787, 796, 922
 - Man(ual) pages for scripts, 948
 - Matteson, Ryan, 610
 - max() function, 34, 81
 - Maximum program size (error), 1067
 - mdb(1), 2, 261, 677, 902, 909
 - mdb(1) kmastat dcmd, 118
 - MediaWiki, 735, 737–738, 842–843
 - Memory allocation, 787
 - Memory Management Unit (MMU), 1082, 1087
 - Memory monitoring
 - analysis, 98–101
 - checklist, 96
 - kernel memory, 118–124
 - one-liners, 97–98
 - providers, 96–97
 - strategy, 95
 - user process memory activity, 101–117
 - Memory usage, 908–917
 - memstat dcmd (d-command), 99
 - Method compilation probes, 691
 - MIB (Message Information Base), 126, 404, 1082
 - mib probes, 407
 - mib provider, 404–408, 423
 - mib provider examples
 - ICMP event by kernel stack trace, 439
 - ICMP event trace, 437
 - IP event statistics, 435
 - SNMP MIB event count, 434–435
 - TCP event statistics, 436
 - UDP event statistics, 437
 - Microsoft FAT16, 375
 - Microsoft FAT32, 375
 - Millisecond to nanosecond conversion, 846
 - min() function, 34, 81
 - Minor faults, 920, 952
 - modinfo(1M), 918
 - Monitor probes, 691
 - Mountpoint, 302, 309–312
 - Mozilla Firefox, 45–46, 109, 428, 706, 769
 - mpstat(1M), 2, 55–57, 72–73, 88, 91, 388
 - mpt, 260–262, 1082
 - Multipathing, 234
 - Multiple aggregations, 37
 - Multithreaded applications, 815, 957, 967
 - Mutex blocks, 796
 - Mutex lock, 87
 - Mutex spin counts, 796
 - mutex_enter(), 66–67, 86–87, 931–932
 - MySQL
 - C API, 849–850
 - DTrace probes, 838
 - one-liner examples, 840–841
 - one-liners, 838–840
 - Reference Manual, 850
 - references, 850–851
 - script summary, 841
 - scripts
 - libmysql_snoop.d, 849–850
 - mysqld_pid_qtime.d, 848–849
 - mysqld_qchit.d, 844–845
 - mysqld_qlower.d, 846–847
 - mysqld_qsnop.d, 841–844
- ## N
- n, 43, 322, 1071
 - Namecache, 210, 340–341, 345–346
 - NetBeans IDE, 962, 966–967
 - netstat(1M), 55, 125, 402, 406, 455
 - network (probes), 838
 - Network Address Translation (NAT), 555
 - Network device driver tracing with fbt, 537–543
 - Network file system. *see* NFS
 - Network I/O, 141–148
 - Network I/O checklist, 403, 559–560
 - Network I/O providers, 560–561
 - Network Information Service, 1083
 - Network Intrusion Detection Systems (NIDS), 871
 - Network lower-level protocols
 - capabilities, 400–402
 - checklist, 403–404
 - common mistakes
 - packet size, 553
 - receive context, 548–550
 - send context, 550–553
 - stack reuse, 554–555
 - providers
 - fbt provider
 - receive, 419–422
 - send, 416–419

- ip provider
 - csinfo_t, 409
 - ifinfo_t, 410
 - ipinfo_t, 409
 - ipv4info_t, 410
 - ipv6info_t, 410
 - pktinfo_t, 409
- mib provider, 405–408
- network providers, 411–415
- one-liners
 - ip provider, 425
 - ip provider examples, 440
 - mib provider, 423
 - mib provider examples, 434–439
 - syscall provider, 422
 - syscall provider examples, 427–434
 - tcp provider, 425
 - tcp provider examples, 441–445
 - udp provider, 427
 - udp provider examples, 445
- planned network provider argument types, 412
- planned network provider arguments, 412
- planned network providers, 412
- scripts
 - Ethernet scripts
 - Mac tracing with fbt, 534
 - macops.d, 534–537
 - network device driver tracing with fbt, 537–543
 - ngelink.d, 546–547
 - ngesnoop.d, 544–546
 - ICMP scripts
 - icmpsnoop.d, 522–525
 - icmptest.d, 521
 - superping.d, 526–529
 - IP scripts
 - fbt provider, 470–474
 - ipfbtsnoop.d, 478–481
 - ipio.d, 475–477, 476
 - ipproto.d, 477–478
 - ipstat.d, 474–475
 - socket scripts
 - soaccept.d, 453–455
 - socketio.d, 457–458
 - socketiosort.d, 458–460
 - soclose.d, 455–457
 - soconnect.d, 449–453
 - soerrors.d, 465–468
 - so1stbyte.d, 460–462
 - sotop.d, 463–464
 - TCP scripts
 - fbt provider, 483–485
 - tcp provider, 482–483
 - tcpaccept.d, 486–487
 - tcpacceptx.d, 488
 - tcpbytes.d, 494
 - tcpconnect.d, 489
 - tcpconnlat.d, 497–499
 - tcpfbtwatch.d, 501–503
 - tcpio.d, 491–493
 - tcpioshort.d, 490
 - tcpnmap.d, 496–497
 - tcp_rwndclosed.d, 500
 - tcpsize.d, 495
 - tcpsnoop.d, 503–516
 - tcpstat.d, 485–486
 - tcp1stbyte.d, 499
 - UDP scripts
 - fbt provider, 517
 - udp provider, 517
 - udpio.d, 520–521
 - udpstat.d, 518–520
 - XDR scripts, 529–533
 - strategy, 402–403
 - Network packet sniffer, 890
 - Network providers, 411–415
 - Network script summary, 445–447, 574–576
 - Network-sniffing tools, 400
 - Network statistic tools, 402
 - New Processes (with Arguments), 798–799
 - Newline (\n), 1021
 - NFS client back-end I/O, 157
 - NFS client scripts
 - nfs3fileread.d, 383–384
 - nfs3sizes.d, 382–383
 - nfswizard.d, 379–381
 - NFS I/O, 162–163
 - nfsstat, 588
 - nfsv3 probes, 577
 - nfsv3 provider, 563
 - NFSv3 provider examples, 569–571
 - nfsv3 provider probes and arguments, 1028–1030
 - NFSv3 scripts
 - nfsv3commit.d, 585–587
 - nfsv3disk.d, 666–668
 - nfsv3errors.d, 588–590
 - nfsv3fbtrws.d, 590–592
 - nfsv3fileio.d, 581
 - nfsv3ops.d, 580
 - nfsv3rwsnoop.d, 578–579
 - nfsv3rwtime.d, 582–583
 - nfsv3syncwrite.d, 584
 - NFSv4 scripts
 - nfsv4commit.d, 595
 - nfsv4deleg.d, 597–599
 - nfsv4errors.d, 595–597
 - nfsv4fileio.d, 594

- NFSv4 scripts (*continued*)
 - nfsv4ops.d, 594
 - nfsv4rwsnoop.d, 594
 - nfsv4rwtime.d, 595
 - nfsv4syncwrite.d, 595
 - nfsv4 provider, 564–566
 - NFSv4 provider examples, 571–572
 - nfsv4 provider probes and arguments, 1030–1034
 - nge driver (Nvidia Gigabit Ethernet), 537
 - NIDS, 871
 - NIS (Network Information Service), 1083
 - NIS scripts, 663–664
 - nmap port scan, 453
 - Nonglobal (local) zone, 870–872
 - normalize(), 34, 36, 143
 - Not enough space (error), 1067
 - Nouri, Nasser, 966
 - nscd (Name Service Cache Daemon), 452, 461, 660, 811
 - nspec, 1007
 - ntohs(), 451, 502, 508
 - NULL character (\0), 1021
 - Nvidia, 237
 - nv_sata, 237, 275
- O**
- o, 43, 727, 935, 1071
 - Object arguments, C++, 690–691
 - Object entropy stat, 709, 711–712
 - Octal value (\0oo), 1021
 - Off-CPU sched provider probe, 674, 786, 897, 932
 - On-CPU sched provider probe, 58–61, 674, 786, 897, 932
 - One-liners
 - C, 684–687
 - cheat sheet, 1072
 - cpu, 58–60
 - disk I/O, 165–166
 - file systems, 300–303
 - I/O, 127–128
 - Java, 693–694
 - JavaScript, 708–709
 - kernel, 918
 - memory, 97–98
 - MySQL, 838
 - network, 411, 422–427
 - Perl, 720–721
 - PHP, 734
 - PostgreSQL, 853
 - provider, 563–568, 793
 - Python, 741
 - Ruby, 753
 - Shell, 765
 - Tcl, 776
 - OpenSolaris, xxx, 1, 336, 411, 451, 949, 1083
 - OpenSolaris security group site, 873
 - OpenSolaris Web site, 962
 - OpenSSH, 649, 876
 - Operator(s)
 - arithmetic, 27, 1021–1022
 - assignment, 27
 - associativity, 1023–1024
 - binary arithmetic, 1021
 - binary bitwise, 1022
 - binary logical, 1022
 - binary relational, 1022
 - boolean, 28
 - precedence, 1023–1024
 - relational, 28, 1022
 - ternary, 28, 178, 195
 - unary arithmetic, 1022
 - unary bitwise, 1023
 - unary logical, 1022
 - @ops aggregation, 602
 - Options, 43–44
 - or (|), 28, 491, 1022–1024
 - OR (| |), 28, 458
 - Oracle, 858–865
 - Oracle Solaris, xxv
 - DTrace privileges, 868
 - Studio 12, 672
 - Studio IDE, 966
 - see also* Solaris
 - Oracle Sun Web Stack, 731, 733
 - Oracle Sun ZFS Storage Appliance, 599, 625
 - OSI model, 400
 - Outbound TCP connections, 489
- P**
- p, 664, 849
 - p PID, 43, 684, 788, 795
 - %P, 488
 - Pacheco, David, 610
 - Packet sniffers, 525, 890
 - Packets (network), 553, 483
 - Page-ins, 95–97, 111–113, 297, 308, 1083
 - Page-outs, 95–97, 297, 1083
 - pagefault, 96, 114–115, 119, 1083
 - panic(), 42, 1007, 1018
 - Passive (TCP term), 482
 - Passive FTP transfers, 629
 - Password sniffing, 869
 - pause(), 923
 - PCFS scripts, 375–376

- Performance Application Programming Interface (PAPI), 791–792, 803
- Perl language, 993–994
 - bug #73630, 720
 - code, 720
 - one-liner examples, 721–722
 - one-liners, 720–721
 - provider, 719
 - script summary, 722
 - scripts
 - pl_calls.d, 723–725
 - pl_calltime.d, 728–731
 - pl_flowinfo.d, 725–728
 - pl_who.d, 722–723
- Perturbations, 269–273
- pgrep(1), 629–631, 849
- PHP
 - code, 733
 - one-liner examples
 - count function calls by filename, 735
 - trace errors, 736
 - trace function calls showing function name, 735
 - one-liners, 734–735
 - script summary, 736
 - scripts
 - php_calls.d, 736
 - php_flowinfo.d, 736, 738
 - php_flowtime.d, 739
 - php_syscolors.d, 739
- pid (process ID), 31, 33, 97–98, 165, 167–168, 788, 790–791
- pid provider, 98, 562, 788–791, 795–796, 839–840, 854
- ping, 447, 462, 522, 525–529, 1081
- Pipe (|) character, 28, 491, 1022–1024
- pktinfo_t, 409, 1040
- Platform Specific Events, 792
- plockstat, 58, 96–97, 689, 787, 811–813
- plockstat provider, 796
- Policy enforcement, 871–872
- Population functions, 1077
- Port closed, 493, 510–511
- Port number, 455
- Port scan, 453, 496
- POSIX, 622, 790, 1084
- PostgreSQL
 - documentation, 858
 - one-liner examples, 854–858
 - one-liners, 853–854
 - probes, 851–852
 - script summary, 855
 - scripts, 854–858
- PostgreSQL-DTrace-Toolkit, The*, 858
- postgresql provider, 853
- Postprocessing, 993–994
- ppid, 31
- ppriv(1), 872, 868
- Predicates, 9, 12, 22, 63, 1084
- Prefetch, 313, 329
- Prefetch requests, 313
- Prefetch Technologies, 610
- Preprocessor, 683
- Principal buffer, 43, 1001, 1003, 1006, 1064–1065, 1080, 1084
- printa(), 34, 36–37, 519
- printf(), 38, 520, 1017
- priv-err, 872, 874
- priv-ok, 872, 874
- Privacy violations, 875–877
- Privilege debugging, 872–874
- Privileges, 868, 1063–1064
- Privileges, detection, and debugging
 - HIDS, 871
 - policy enforcement, 871–872
 - privilege debugging, 872–874
 - reverse engineering, 874–875
 - security audit logs, 870
 - sniffing, 869
- probefunc, 31, 71, 91, 110, 132
- probemod, 31, 71
- probenam, 31, 110
- probeprov, 31
- proc provider, 11, 793–794
- proc provider probes and arguments, 1034
- Process destructive actions, 1019
- Process ID (pid) provider. *see* pid provider
- Process name, 307
- Process watching, 881
- Processes paging in from the file system, 308
- Processors. *see* CPUs
- procstat(1), 55
- proctime, 806–808
- Production queries, 843
- profile, 24–25, 46, 61, 996–997, 1084
- profile provider, 11, 58–59, 63, 794–795, 919
- Profile Python Stack Traces, 743–744
- Profiling process names, 46–47
- Profiling user modules, 818
- Profiling user stacks, 817
- Program counter (PC), 61
- Program execution flow, 673
- Programming language providers, 675
- Promiscuous mode, 525, 544, 875, 890–891
- Provider, 11, 1084
- Provider arguments reference
 - bufinfo_t, 1038

Provider arguments reference, arguments (*continued*)

- conninfo_t, 1040
- cpuinfo_t, 1039
- csinfo_t, 1040
- devinfo_t, 1038
- fileinfo_t, 1038
- ifinfo_t, 1041
- ipinfo_t, 1040
- ipv4info_t, 1041
- ipv6info_t, 1041
- lwpsinfo_t, 1039
- pktinfo_t, 1040
- psinfo_t, 1039
- tcpinfo_t, 1042
- tcpinfo_t, 1043
- tcpsinfo_t, 1042
- fc provider probes and arguments, 1025–1026
- fsinfo provider probes and arguments, 1026
- io provider probes and arguments, 1026
- ip provider probes and arguments, 1027
- iscsi provider probes and arguments, 1027
- nfsv3 provider probes and arguments, 1028–1030
- nfsv4 provider probes and arguments, 1030–1034
- proc provider probes and arguments, 1034
- sched provider probes and arguments, 1035
- srp provider probes and arguments, 1035
- sysevent provider probes and arguments, 1036
- tcp provider probes and arguments, 1036
- udp provider probes and arguments, 1036
- xpv provider probes and arguments, 1037
- Providers for Various Shells Web site, 764–765
- prstat(1), 73–74, 77–78, 82, 100
- prstat(1M), 55, 60, 73, 74, 801
- ps(1), 62, 100
- PSARC, 764, 1084
- psinfo_t, 1039
- Python language
 - code, 741
 - one-liner examples, 742–744
 - one-liners, 741
 - patches and bugs, 740
 - script summary, 744
 - scripts
 - py_calldist.d, 750
 - py_calls.d, 744–746
 - py_calltime.d, 744, 748–751
 - py_cpudist.d, 750
 - py_cputime.d, 750
 - py_flowinfo.d, 746–748
 - py_flowtime.d, 748
 - py_syscolors.d, 748
 - py_who.d, 744–745

Q

- q, 43–44, 69, 880–881, 885, 1007
- quantize(), 34–35, 138, 148, 270, 571
- Query (probes), 838
- Query cache hit rate, 841, 844–845
- Query count summary, 840
- Query execution (probes), 838
- Query parsing (probes), 838
- Query processing, database, 836
- Query time distribution plots, 841, 848–849
- Question mark, 160, 203, 1021
- Quiet mode, 43–44, 69, 880–881, 885, 1007, 1071
- Quote marks
 - backquote, 33, 64, 78, 231
 - double, 880, 1021
 - single, 24, 194, 201, 231, 880, 1021

R

- raise(), 872, 888–891, 1007, 1019
- Random I/O, 202, 208–209
- Random reads, 579
- Random workload, 185–186
- Read-aheads, 197, 298, 314, 354–355, 377, 989
- Read I/O size distribution, 571
- Read workload, 220
- Reader/writer locks, 9, 796, 1085
- read_nocancel(), 306
- Reads by file system type, 306–307
- Receive (network), 408, 419–422
- Receive context, 548–550
- Relational operators, 28, 1022
- Remote host latency, 661
- Remote hosts, 442–443
- Return (syscall), 25
- Reusable kernel objects, 909
- Reverse engineering, 874–875
- RFC, 473, 481, 517, 1015, 1085
- Ring buffer, 1084–1085
- RIP protocol, 562
- Root privileges, 20
- Root user privileges, 868–869
- Round-trip time (RTT), 477
- RT (real time), 942
- ruby-dtrace, 751
- Ruby language
 - code, 752
 - one-liner examples, 753–755
 - one-liners, 753
 - provider, 751
 - script summary, 755
 - scripts
 - rb_calls.d, 756–757
 - rb_calltime.d, 759–762

rb_flowinfo.d, 757–759
rb_who.d, 755–756

S

- s file, 43
- sar(1), 55
- SAS driver reference, 260
- SAS scripts
 - mptevents.d, 264–267
 - mptlatency.d, 267–269
 - mptsasscsi.d, 263–267
- sata, 275
- SATA command, 279–290
- SATA driver reference, 237
- SATA DTracing
 - documentation, 274
 - frequency count fbt, 278–279
 - frequency count sdt, 276–277
 - function arguments, 283–285
 - latency, 285–287
 - stable providers, 275
 - stack backtraces, 280–283
 - testing, 288
 - unstable providers: fbt, 277–278
 - unstable providers: sdt, 275–276
- SATA scripts
 - satacmds.d, 172, 237–243
 - satalatency.d, 248–250
 - satareasons.d, 246–248
 - satarw.d, 243–246
- SATA stack, 274
- Scalar globals, 31–32
- Scalar variables, 28
- sched, 405
- Sched (scheduler), 202–203
- sched provider, 60, 97, 405, 795, 920
- sched provider probes and arguments, 1035
- Scheduling class, 57, 347, 420, 932, 941–942, 952
- scp, 308, 649–651, 654
- Script summaries
 - application, 804
 - C, 689
 - disk I/O, 172–173
 - DTraceToolkit, 949–957
 - file systems, 313–315
 - Java, 696
 - JavaScript, 712
 - kernel, 932
 - MySQL, 841
 - network, 445–447, 574–576
 - Perl, 722
 - PHP, 736
 - PostgreSQL, 855
 - Python, 744
 - Ruby, 755
 - security, 875
 - shell, 768
 - Tcl, 778
- Scripting languages, 669
- Scripts
 - applications scripts, 804
 - execsnoop, 805–806
 - kill.d, 813–814
 - plockstat, 811–813
 - procsnoop.d, 804–806
 - procsystime, 806–808
 - sigdist.d, 814–815
 - threaded.d, 815–816
 - uoffcpu.d, 809–811
 - uoncpu.d, 808–809
- C language, 689
- CIFS scripts, 599
 - cifsevents.d, 605–607
 - cifsfbtinfo.d, 607–609
 - cifsfileio.d, 603
 - cifsops.d, 602–603
 - cifsrsnoop.d, 600–601
 - cifsrwtime.d, 604
- DNS scripts, 621
 - dnsgetname.d, 623–625
 - getaddrinfo.d, 622–623
- DTrace Toolkit scripts list, 949–961
- ethernet scripts, 533
 - Mac tracing with fbt, 534
 - macops.d, 534–537
 - network device driver tracing, 537–543
 - ngelink.d, 546–547
 - ngesnoop.d, 544–546
- Fibre Channel scripts, 646
 - fcerror.d, 647–649
 - fcwho.d, 647
- fsinfo scripts, 327
 - fssnoop.d, 333–335
 - fswho.d, 328
 - readtype.d, 329–332
 - writetype.d, 332–333
- FTP scripts, 625
 - ftpdfileio.d, 626–627
 - ftpdxfer.d, 625–626
 - proftpdcmd.d, 627–629
 - proftpdio.d, 632–633
 - proftpdtime.d, 630–632
 - tnftpdcmd.d, 630
- HFS+ scripts, 370
 - hfsfileread.d, 374–375
 - hfslower.d, 372–374
 - hfsnoop.d, 371–372

Scripts (*continued*)

- HSFS scripts, 376
 - cdrom.d, 377–378
- HTTP scripts, 609
 - httpclients.d, 612–613
 - httpdurls.d, 616–618
 - httperrors.d, 614
 - httpio.d, 614–615
 - weblatency.d, 618–621
- ICMP scripts, 521
 - icmpsnoop.d, 522–525
 - icmpstat.d, 521
 - superping.d, 526–529
- IDE scripts, 250
 - ideerr.d, 257
 - idelatency.d, 252–254
 - iderw.d, 255–257
- io provider scripts, 172
 - bitesize.d, 181–183
 - disklatency.d, 175–177
 - geomiosnoop.d, 209–210
 - iolatency.d, 172–175
 - iopattern, 207–209
 - iosnoop, 187–203
 - iotop, 204–207
 - iotypes.d, 178–179
 - rwtime.d, 179–181
 - seeksize.d, 184–187
- IP scripts, 469
 - fbt provider, 470–474
 - ipfbtsnoop.d, 478–481
 - ipio.d, 475–477, 476
 - ipproto.d, 477–478
 - ipstat.d, 474–475
- iSCSI scripts, 633
 - iscsicmds.d, 643–644
 - iscsirwsnoop.d, 640–641
 - iscsirwtime.d, 641–643
 - iscsiterr.d, 644–646
 - iscsiwho.d, 638–639
 - providers, 634–638
- Java, 696
 - j_calls.d, 696–698
 - j_calltime.d, 701–704
 - j_flow.d, 698–700
 - j_thread.d, 704–705
- JavaScript, 712
 - js_calls.d, 712–713
 - js_calltime.d, 715–718
 - js_flowinfo.d, 713–715
- kernel, 932
 - cswstat.d, 932, 943–944
 - intrstat, 932–934
 - koffcpu.d, 932, 938–939
 - koncpu.d, 932, 937–938
 - lockstat, 934–937
 - priclass.d, 932, 941–943
 - putnexts.d, 932, 944–945
 - taskq.d, 932, 939–941
- LDAP scripts, 664
 - ldapsyslog.d, 664–666
- multiscripts, 666
 - nfsv3disk.d, 666–668
- MySQL, 841
 - libmysql_snoop.d, 849–850
 - mysqld_pid_qtime.d, 848–849
 - mysqld_qchit.d, 844–845
 - mysqld_qlower.d, 846–847
 - mysqld_qsnoop.d, 841–844
- NFS client scripts, 379
 - nfs3fileread.d, 383–384
 - nfs3sizes.d, 382–383
 - nfswizard.d, 379–381
- NFSv3 scripts, 576
 - nfsv3commit.d, 585–587
 - nfsv3errors.d, 588–590
 - nfsv3fbtrws.d, 590–592
 - nfsv3fileio.d, 581
 - nfsv3ops.d, 580
 - nfsv3rwsnoop.d, 578–579
 - nfsv3rwtime.d, 582–583
 - nfsv3syncwrite.d, 584
- NFSv4 scripts, 592
 - nfsv4commit.d, 595
 - nfsv4deleg.d, 597–599
 - nfsv4errors.d, 595–597
 - nfsv4fileio.d, 594
 - nfsv4ops.d, 594
 - nfsv4rwsnoop.d, 594
 - nfsv4rwtime.d, 595
 - nfsv4syncwrite.d, 595
- NIS scripts, 663
 - nismatch.d, 663–664
- PCFS scripts, 375
 - pcfsrw.d, 375–376
- Perl, 722
 - pl_calls.d, 723–725
 - pl_calltime.d, 728–731
 - pl_flowinfo.d, 725–728
 - pl_who.d, 722–723
- PHP, 736
 - php_calls.d, 736
 - php_flowinfo.d, 738
- PostgreSQL, 854
 - pg_pid_qtime.d, 856–858
 - pg_qlower.d, 855–856
- Python, 744
 - py_calls.d, 745–746

- py_calltime.d, 748–751
- py_flowinfo.d, 746–748
- py_who.d, 744–745
- Ruby, 755
 - rb_calls.d, 756–757
 - rb_calltime.d, 759–762
 - rb_flowinfo.d, 757–759
 - rb_who.d, 755–756
- SAS scripts, 259
 - mpthevents.d, 264–267
 - mptlatency.d, 267–269
 - mptsasscsi.d, 263–267
- SATA scripts, 236
 - satacmds.d, 237–243
 - satalatency.d, 248–250
 - satareasons.d, 246–248
 - satarw.d, 243–246
- SCSI scripts, 211
 - SCSI probes, 212–213
 - scsicmds.d, 218–221
 - scsi.d, 229–236
 - scsilatency.d, 221–223
 - scsireasons.d, 227–229
 - scsirw.d, 223–226
 - sdqueue.d, 213–215
 - sdretry.d, 215–218
- security scripts, 875
 - cuckoo.d, 884–886
 - keylatency.d, 882–884
 - networkwho.d, 891–892
 - nosetuid.d, 888–889
 - nosnoopforyou.d, 890–891
 - script summary, 875
 - shellnoop, 878–882
 - sshkeynoop.d, 875–878
 - watchexec.d, 886–888
- Shell, 768
 - sh_calls.d, 769–771
 - sh_flowinfo.d, 771–774
 - sh_who.d, 768–769
- socket scripts, 447
 - soaccept.d, 453–455
 - socketio.d, 457–458
 - socketiosort.d, 458–460
 - soclose.d, 455–457
 - soconnect.d, 449–453
 - soerrors.d, 465–468
 - so1stbyte.d, 460–462
 - sotop.d, 463–464
- SSH scripts, 649
 - scpwatcher.d, 661–663
 - sshcipher.d, 649–655
 - sshconnect.d, 657–661
 - sshactivity.d, 655–657
- syscall provider, 315
 - fserrors.d, 326–327
 - fsrtpk.d, 320–322
 - fsrwcoun.d, 317–319
 - fsrwtime.d, 319–320
 - mmap.d, 324–325
 - rwsnoop, 322–323
 - sysfs.d, 315–317
- Tcl, 778
 - tcl_calls.d, 779–780
 - tcl_inflow.d, 782
 - tcl_procflow.d, 780–782
 - tcl_who.d, 778–779
- TCP scripts, 481
 - fbt provider, 483–485
 - tcp provider, 482–483
 - tcpaccept.d, 486–487
 - tcpacceptx.d, 488
 - tcpbytes.d, 494
 - tcpconnect.d, 489
 - tcpconnat.d, 497–499
 - tcpfbtwatch.d, 501–503
 - tcpio.d, 491–493
 - tcpioshort.d, 490
 - tcpnmap.d, 496–497
 - tcp_rwndclosed.d, 500
 - tcpsize.d, 495
 - tcpsnoop.d, 503–516
 - script: fbt-based, 505–515
 - script: tcp-based, 515–516
 - tcpstat.d, 485–486
 - tcp1stbyte.d, 499
- TMPFS scripts, 385
 - tmpgetpage.d, 386–387
 - tmpusers.d, 385–386
- UDFS scripts, 378
 - dvd.d, 378
- UDP scripts, 517
 - fbt provider, 517
 - udp provider, 517
 - udpio.d, 520–521
 - udpstat.d, 518–520
- UFS scripts, 351
 - ufsimiss.d, 356–357
 - ufsreadahead.d, 354–356
 - ufssnoop.d, 352–354
- VFS scripts, 335
 - dnlcps.d, 346–347
 - fsflush_cpu.d, 347–349
 - fsflush.d, 349–351
 - maclife.d, 344–345
 - macvfssnoop.d, 338–340
 - sollife.d, 343–344
 - solvfssnoop.d, 336–338

- Scripts, VFS scripts (*continued*)
 - vfslife.d, 345
 - vfssnoop.d, 340–343
- XDR scripts, 529
 - xdrshow.d, 529–533
- ZFS scripts, 357
 - perturbation.d, 366–368
 - spasync.d, 369–370
 - zfsslower.d, 360–361
 - zfssnoop.d, 358–359
 - zioprint.d, 361–363
 - ziosnoop.d, 363–365
 - ziotype.d, 365–366
- scrwtop10.d script, 132–133
- SCSI probes, 212–213
- SCSI scripts, 211
 - SCSI probes, 212–213
 - scsicmds.d, 218–221
 - scsi.d, 229–236
 - scsilatency.d, 221–223
 - scsireasons.d, 227–229
 - scsirw.d, 223–226
 - sdqueue.d, 213–215
 - sdretry.d, 215–218
- SCSI virtual host controller interconnect, 221, 234–236
- sctp, 412–413
- sdt (statically defined tracing), 156, 275–276
- sdt provider, 303, 921
- sdt provider examples, 312–313
- Secure Shell. *see* SSH
- Security, 867
 - audit logs, 870
 - privileges, detection, and debugging, 867
 - HIDS, 871
 - malicious acts, 869
 - policy enforcement, 871–872
 - privilege debugging, 872–874
 - privileges, 868
 - reverse engineering, 874–875
 - security logging, 870
 - sniffing, 869–870
 - script summary, 875
 - scripts
 - cuckoo.d, 884–886
 - keylatency.d, 882–884
 - networkwho.d, 891–892
 - nosetuid.d, 888–889
 - nosnoopforyou.d, 890–891
 - shellsnoop, 878–882
 - sshkeysnoop.d, 875–878
 - watchexec.d, 886–888
- sed, 240
- segkmem, 121
- Segment driver, 121
- select-start / select-done, 838
- self->, 30, 41, 143, 228, 660, 997, 1085
- Semaphore system call, 93
- Semicolons, 23
- Send, 408, 416–419
- Send context, 550–553
- Sequential I/O, 208
- Sequential Workload, 185
- Server Message Block (SMB). *see* CIFS
- Server query status trace (simple snoop), 854
- Service time, disk I/O, 155
- setuid(), 875
- Shapiro, Mike, 1, 1003
- Shared memory, 100–101
- Shell (language), 764, 1085
 - code, 765
 - one-liner examples
 - count function calls by filename, 767
 - count line execution by filename and line number, 767
 - trace function calls showing function name, 766
 - one-liners, 765–766
 - script summary, 768
 - scripts
 - sh_calls.d, 769–771
 - sh_flowinfo.d, 771–774
 - sh_flowtime.d, 774
 - sh_syscolors.d, 774
 - sh_who.d, 768–769
- shellsnoop, 878–882
- short, 26
- Shouting in the data center, 269–273
- Show user stack trace on function call, 687–688
- Signals, 804, 813–814
- Signed integers, 26–27
- Simple snoop, 854
- Single quote mark, 24, 194, 201, 231, 880, 1021
- sizeof(), 41
- Slab allocator, 909, 913
- smb provider, 566, 572–573
- Sniffing, 869–870, 875
- SNMP Message Information Bases (MIBs), 404
- SNMP MIB event count, 424, 434–435
- snoop, 400
- Snoop process execution, 48–49
- snoop(1M), 890
- Socket accepts by process name, 422, 427
- Socket connections by process and user stack trace, 422, 428
- Socket file system, 76
- Socket flow diagram, 448
- Socket read bytes by process name, 423, 433

- Socket read (write/send/rcv) I/O count by process name, 423, 430
- Socket read (write/send/rcv) I/O count by system call, 422, 429
- Socket reads (write/send/rcv) I/O count by process and user stack trace, 423, 431
- Socket reads (write/send/rcv) I/O count by system call and process name, 423, 430
- Socket scripts, 447
 - soaccept.d, 453–455
 - socketio.d, 457–458
 - socketiosort.d, 458–460
 - soclose.d, 455–457
 - soconnect.d, 449–453
 - soerrors.d, 465–468
 - so1stbyte.d, 460–462
 - sotop.d, 463–464
- Socket system call error descriptions, 467–468
- Socket write bytes by process name, 432
- Socket write I/O size distribution by process name, 433
- Solaris, 416–418, 420, 471–474, 1076
 - 80-character maximum, 493
 - AF_INET values, 451
 - disk I/O on a Solaris Server, 176–177
 - I/O stack, 153
 - IDE driver reference, 251
 - iostat(1M), 125
 - kernel memory tools, 118
 - lower-level network stack, 533
 - netstat(1M), 125
 - performance analysis, 52
 - SAS driver reference, 260
 - SATA driver reference, 237
 - system tools, 55
 - TCP/IP stack, 401
- Solaris Auditing, 870
- Solaris Dynamic Tracing Guide*, 19, 157
- Solaris Internals*, 66
- Solaris Nevada, 298, 411, 470–471, 576–578, 592, 1011, 1025, 1038, 1086
- Solaris Performance and Tools*, 52
- sort, 992
- Sort options, 37
- specsize, 1007
- Speculations, 41–42, 1006, 1019
- SpiderMonkey, 706, 818
- spin, 812, 934, 936, 1015, 1086
- srp provider probes and arguments, 1035
- ssh, 428, 455
- SSH logins, 563, 569
- SSH scripts
 - scpwatcher.d, 661–663
 - sshcipher.d, 649–655
 - sshconnect.d, 657–661
 - sshdactivity.d, 655–657
- ssh vs. telnet, 462
- sshd (SSH daemon), 132, 189, 291, 462, 569, 649, 655–657, 661, 994–995
- sshkeynoop.d, 875–878
- Stability, 275, 806, 1086
- stack(), 40–41, 90, 92, 113, 551, 1008, 1017, 1071
- Stack backtrace counts, 929
- Stack reuse, 554–555
- Stack traces, 155, 168–171, 280–283, 312–313
- Stackdepth, 31, 698
- stackframes, 44, 1008
- stat() files, 300, 305
- Stat tools, 56
- State changes, tcp, 415
- Static probes, 4, 1085, 1086
- Statically Defined Tracing provider. *see* sdt provider
- Statistics (Analytics), 977–984
- stddev() function, 34
- STDOUT, 662, 878–880, 1014, 1019, 1086
- Stoll, Clifford, 884
- Stream Control Transmission Protocol (sctp), 412–413
- Streaming workload, 578–579
- STREAMS, 534, 944
- strftime(), 688
- String buffer, 44
- String types, 27
- String variables, 1008
- stringof(), 39–40, 1067
- strjoin(), 40, 178, 431, 1016
- strlen(), 40, 618, 687–688, 999–1000, 1016
- strsize, 44, 1008
- strtok(), 618–619, 1016
- Subroutines, 720–726, 729–730, 1014, 1087
- Subversion, 190
- sudo, 20
- sum(), 34, 118
- Sun Microsystems, 269n, 663, 973, 1079, 1083, 1084, 1086
- Sun Studio IDE, 966
- Switch buffer, 1008, 1064, 1080, 1087
- switchrate, 194, 992, 1003, 1008, 1065, 1087
- sync-cache, 225–226, 246
- Synchronous vs. asynchronous write workloads, 241
- Synchronous writes, 584–585, 595
- Synchronous ZFS writes, 242, 254
- SYS (system), 347, 942
- syscall Entry and Return, 25
- syscall provider, 60, 90–91, 126, 300–301, 404, 422, 563, 794, 919

- syscall provider examples
 - frequency count stat() files, 305
 - http files opened by the httpd server, 568
 - reads by file system type, 306
 - socket accepts by process name, 427
 - socket connections by process and user stack trace, 428
 - socket read bytes by process name, 423, 433
 - socket read (write/send/recv) I/O count by process name, 423, 430
 - socket read (write/send/recv) I/O count by system call, 422, 429
 - socket reads (write/send/recv) I/O count by process and user stack trace, 423, 431
 - socket reads (write/send/recv) I/O count by system call and process name, 423, 430
 - socket write bytes by process name, 432
 - socket write I/O size distribution by process name, 433
 - SSH logins by UID and home directory, 569
 - trace file creat() calls with process name, 304
 - trace file opens with process name, 304
 - tracing cd, 306
 - writes by file system type, 307
 - writes by process name and file system type, 307
 - Syscall provider scripts
 - fserrors.d, 326–327
 - fsrtpk.d, 320–322
 - fsrwcoun.d, 317–319
 - fsrwtime.d, 319–320
 - mmap.d, 324–325
 - rwsnoop, 322–323
 - sysfs.d, 315–317
 - sysevent provider probes and arguments, 1036
 - sysinfo provider, 58, 87–88, 90–91, 920
 - syslog(), 664–666
 - sysstat(1), 55
 - System activity reporter, 55
 - System call counts for processes called httpd, 800
 - System call time reporter, 806
 - System calls, 994–995
 - System tools, 55
 - System view
 - CPU tracking
 - analysis, 60–85
 - checklist, 57–58
 - events, 87–94
 - interrupts, 85–88
 - one-liners, 58–60
 - providers, 58
 - disk and network I/O activity
 - analysis, 128–134
 - checklist, 125
 - disk I/O, 134–141
 - one-liners, 127–128
 - providers, 126–127
 - strategy, 125
 - memory monitoring
 - analysis, 98–101
 - checklist, 96
 - kernel memory, 118–124
 - one-liners, 97–98
 - providers, 96–97
 - strategy, 95
 - user process memory activity, 101–117
 - system methodology, 53–56
 - system tools, 54–56
 - Systemwide sniffing, 881
- ## T
- Tail-call optimization, 1003
 - \$target, 32, 43, 788, 1070–1071
 - task queues, 939–941
 - Tcl (language), 774
 - code, 776
 - one-liner examples, 777–778
 - one-liners, 776–777
 - pronunciation, 774
 - script summary, 778
 - scripts
 - tcl_calls.d, 779–780
 - tcl_flowtime.d, 781
 - tcl_inflow.d, 782
 - tcl_procfow.d, 780–782
 - tcl_syscolors.d, 781
 - tcl_who.d, 778–779
 - TCP (Transmission Control Protocol), 481, 1087
 - TCP connections, 441, 446, 486–489
 - TCP event statistics, 424, 436
 - tcp fusion, 408
 - TCP handshake, 408, 482, 492–493, 514–515
 - TCP Large Send Offload, 553
 - tcp provider, 404, 425, 482–483
 - tcp provider examples
 - inbound TCP connections, 441
 - sent IP payload size distributions, 443
 - sent TCP bytes summary, 444
 - TCP events by type summary, 444
 - TCP received packets, 443
 - tcp provider probes and arguments, 1036
 - TCP scripts
 - fbt provider, 483–485
 - tcp provider, 482–483
 - tcpaccept.d, 486–487
 - tcpacceptx.d, 488
 - tcpbytes.d, 494

- tcpconnect.d, 489
- tcpconnlat.d, 497–499
- tcpfbtwatch.d, 446, 501–503
- tcpio.d, 491–493
- tepioshort.d, 490
- tcpnmap.d, 496–497
- tcp_rwndclosed.d, 500
- tepsize.d, 495
- tcpsnoop.d, 503–516
- tcpstat.d, 485–486
- tcp1stbyte.d, 499
- TCP window buffer, 552
- tcpdump, 400
- tcpinfo_t, 1042
- tcpsinfo_t, 1042
- telnet, 462
- Ternary operators, 28, 178, 195
- The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*, 884
- this->, 30, 176, 182, 997, 1087
- Thread life-cycle probes, 691
- Thread-local variables, 997–998
- Tick probe, 24–25, 1002
- tid, 31
- Time functions, 526
- Time-share scheduling code, 420
- Time stamps, 31, 807, 992–993, 995–996
- timestamp *vs.* vtimestamp, 995–996
- Timing a system call, 47–48
- Tips and tricks
 - assumptions, 1000
 - drops and dynvardrops, 1003
 - frequency count, 991–992
 - grep, 991
 - known workloads, 987–989
 - performance issues, 1001–1002
 - Perl, 993–994
 - postprocessing, 993–994
 - profile probe, 996–997
 - script simplicity, 1001
 - strlen() and strcmp(), 999–1000
 - system calls, 994–995
 - tail-call optimization, 1003
 - target software, 989–991
 - timestamp variables, 992–993
 - timestamp *vs.* vtimestamp, 995–996
 - variables
 - clause-local variables, 30–31, 998
 - global and aggregation variables, 999
 - thread-local variables, 997–998
- TLB, 798, 924–925, 937, 1087
- TMPFS scripts
 - tmpgetpage.d, 386–387
 - tmpusers.d, 385–386
- Tools, 947
 - Analytics
 - abstractions, 974
 - breakdowns, 979–980
 - controls, 983
 - datasets, 984
 - diagnostic cycle, 975
 - drill-downs, 981
 - heat maps, 979–980
 - hierarchical breakdowns, 979–980
 - load *vs.* architecture, 975
 - real time, 975
 - statistics, 977
 - visualizations, 975
 - worksheets, 983
 - Chime, 962–965
 - DLight, Oracle Solaris Studio 12.2, 966–971
 - DTrace GUI Plug-in for NetBeans and Sun Studio, 966
 - DTraceToolkit
 - installation, 949
 - script example: cpuwalk.d, 957–961
 - Man page, 959–960
 - script, 958–959
 - scripts, 949–957
 - versions, 949
 - Mac OS X Instruments, 971–972
- top(1), 55
- trace(), 37, 684–685, 799
- Trace command calls showing command name, 778
- Trace errors, 170–171
- Trace file creat() calls with process name, 304–305
- Trace file opens with process name, 304
- Trace function calls, 710, 735, 742, 766
- Trace function entry arguments, 687
- Trace method calls showing class and method, 754
- Trace PHP errors, 736
- Trace procedure calls showing procedure name, 777
- Trace program execution showing filename and line number, 709
- Trace subroutine calls, 721
- tracemem(), 39, 799, 1017
- Tracing fork() and exec(), 45
- Tracing open(2), 44–45
- Transaction group, 179
- Translation code, 162
- Translation Lookaside Buffer (TLB). *see* TLB
- Translators, 42, 1087
- Transmission Control Protocol (RFC 793), 481
- trunc(), 33–34, 36, 110, 131, 133, 581

truss(1), 869
 TS (time sharing), 420, 942–943
 Tunable variables, 1005–1010
 Type cast, 26, 1087
 Types, 26–27

U

uberblock, 641, 1088
 UDFS scripts, 378–379
 udp, 404
 UDP (User Datagram Protocol), 1088
 UDP event statistics, 424, 437
 udp provider, 404, 427, 517
 examples, 445
 probes and arguments, 1036
 UDP scripts
 fbt provider, 517
 udp provider, 517
 udpio.d, 520–521
 udpstat.d, 518–520
 UFS scripts, 351
 ufsmiss.d, 356–357
 ufsreadahead.d, 354–356
 ufssnoop.d, 352–354
 UFS *vs.* ZFS synchronous writes, 242
 uid, 31
 uint64_t, 26
 Unanchored probes, 1088
 Unary arithmetic operators, 1022
 Unary bitwise operators, 1023
 Unary logical operators, 1022
 Unary operators, 27
 Uncached file system read, 331
 Uncomment characters, 1088
 Underscore, 793*n*
 Unix File System. *see* UFS
 unrolled loop, 232, 618
 Unsigned integers, 26–27
 unsigned long, 1021
 unsigned long long, 27, 1021
 Unstable, 1088
 Unstable interface, 790
 Unstable providers, 275–278
 uregs[], 31, 677, 791, 1013
 URLs accessed, 616
 USB storage, 375
 USDT, 1088
 USDT example, Bourne shell provider, 1052–1061
 User Datagram Protocol. *see* UDP
 User-land, 1088
 User-land C, 680
 User-mode instructions, 801–803
 User-mode level-two cache misses, 803–804

User process memory activity, 101–117
 User stack trace profile at 101 hertz, 800–801
 usermod(1M), 868
 ustack(), 40–41, 90, 113, 165, 687, 872, 1008,
 1017, 1071
 ustackframes, 44, 1008–1010
 Utilities, 55

V

-v, 161
 -V, 811
 Variables, 9
 associative arrays, 29
 built-in, 31–32
 clause local, 30–31
 clause-local variables, 998
 DTrace tunable, 1005–1010
 external, 33
 global and aggregation variables, 999
 macro, 32
 operators, 27–28
 scalar, 28
 structs and pointers, 29
 thread local, 30
 thread-local variables, 997–998
 types, 26–27
 Vertical tab (\v), 1021
 vfs (virtual file system), 126
 vfs provider, 298, 303
 VFS scripts
 dnleps.d, 346–347
 fsflush_cpu.d, 347–349
 fsflush.d, 349–351
 maclife.d, 344–345
 macvfssnoop.d, 338–340
 sollife.d, 343–344
 solvfssnoop.d, 336–338
 vflife.d, 345
 vfssnoop.d, 340–343
 Video demonstration, 269–273
 vim, 343–344
 Virtual File System. *see* vfs
 Virtual host controller interconnect, 221, 234–236
 Virtual memory, 898
 Virtual memory info provider (vminfo), 297
 Virtual Network Computing (VNC), 824
 VirtualBox simulator version, 973
 VM life-cycle probes, 691
 vmem, 120, 894, 913
 vmem heap segment, 913
 vminfo provider, 96–97, 302, 308, 920
 vm_map_enter(), 105
 vmstat, 909

vmstat(1), 55–56
vm_stat(1), 55, 95, 99
vmstat(8), 55
vmstat(1M), 55, 388–389
vmstat(1M/8), 95
vmtop10.d script, 109–110
vnode interface statistics, 951
vnode_getattr(), 929
VNOP interface, 303, 338–339
Volume manager, 332, 357, 1088
VOP interface, 303, 340–343
VOP_READ_APV(), 170
vopstat, 951
VThread-local variables, 30, 997–998, 1085, 1087
vtimestamp, 31, 995–996

W

-w, 43, 1007
Wait service time, 213–215
walltimestamp, 31
Web browsers, tracking, 573–574
Web server processes, 323–324
while getopt loop, 193
Whitespace, 887, 1008
Wi-Fi *vs.* Ethernet, 462
Wiki software, 735
Wildcards, 23–24, 305–307, 690, 991
Workload, 102, 254, 270, 987, 1088
Worksheets (Analytics), 983
Write canceling, 332
write DMA extended, 242
Writes by file system type, 307
Writes by process name and file system type, 307
Writing target software, 989–991

X

-x, 43, 843
xcalls (cross calls), 91
XDR (External Data Representation), 270, 1088
 scripts, 447, 529–533
xpv provider probes and arguments, 1037
Xvnc case study
 profile provider, 829–831
 syscall provider, 824–829

Y

Youtube demonstration video, 269–273

Z

-Z, 316, 375–376, 626, 744, 756
zalloc(), 105, 916–917
ZFS, 221, 225, 241–242, 250
 I/O pipeline (ZIO), 357, 361
ZFS ARC, 303, 312–313
ZFS function calls, 688
ZFS 8KB mirror reads
 cross calls, 390–393
 vmstat(1M), mpstat(1M), and iostat(1M), 388–389
ZFS scripts
 perturbation.d, 366–368
 spasync.d, 369–370
 zfsslower.d, 360–361
 zfssnoop.d, 358–359
 zioprint.d, 361–363
 ziosnoop.d, 363–365
 ziotype.d, 365–366
zpool status, 221
Zprint, 909